DRUID ESSENTIALS

Andrea Carboni

Rev 4, 29-nov-2003

Contents

Chapter 1

Getting started

1.1 Requirements

Druid is a java application so it needs a java virtual machine to work (JVM). You can download the JVM from one of the following sites:

- Sun (http://java.sun.com/j2se)
- Blackdown (http://www.blackdown.org)
- IBM (`www.ibm.com`)
- Bea (`www.bea.com`)
- Kaffe (`www.kaffe.org`)
- Sable (`www.sablevm.org`)

Druid works with JRE 1.4 so it must be installed and the java executable must be in your path. Once the JVM is installed you may install the Druid package. Make sure you have at least 32 MB of free memory. If you plan to work with big databases (more than 100 tables) 64 MB of free memory are a requirement.

1.2 Installation

Issue the command:

```
java -jar druid-x.x-install.jar
```

to start the installer and follow the instructions. On some platforms, you can just point on the jar icon and click to install the package.

1.3 Starting Druid

Issue the command:

```
java -jar <druid directory>/druid.jar
```

You don't need to be into the Druid's directory and you don't have to setup the classpath. On some platforms, you can just point on the druid.jar icon and click to run the program. Many platforms (Windows, KDE etc...) let you associate the jar extension to the JVM. This way you can run Druid with a mouse click.

1.4 JDBC Drivers

Druid is able to connect to a DBMS via JDBC to extract/rebuild the database schema. If you plan to use this feature you have to download and install the proper jdbc driver. The driver must be:

- A pure java type 4 thin driver
- Compliant to the JDBC 2.1 API specifications

Usually, each DBMS has associated a JDBC driver, so you can download it from the same site. A list of drivers is available on the java home page. If the driver is not compliant to the JDBC API Druid can still use it but you could not be able to do some operations.

1.4.1 Installing a JDBC driver

If you want to install a new JDBC driver follow these steps:

- Open the menu and select "Options ▷ Jdbc drivers" to open the driver dialog
- Press the "new" button to Open the file dialog
- Browse your file system and select the jar of the driver you want to install

If there are no errors Druid will add the driver to its list.

1.5 License

The product is released under the GPL license. I want to give the product to the community hoping that it will be useful, so you can use any part of it as you want (obviously without any warranty). The full license is located in the "docs/COPYING.TXT " file.

Chapter 2

Overview

2.1 Introduction

Many firms have plenty of data which needs to be stored somewhere. This data can be subdivided into "groups" (called tables) where each group contains data of the same type (for example a group of users, another of patients, another of employees and so on). With all this data to manage arises the figure of the Database Administrator (DBA) : a person with strong knowledge of data managment. Recently, the work of the DBA is increased, because databases have become very large with hundreds of tables causing many headaches to the DBA. To solve this problem software tools have appeared, each with its own pros and cons, to help the DBA.

2.2 What Druid is

Druid is a cross platform tool written in java, which simplifies many repetitive tasks making database creation / maintanance as fast as possible. With Druid you can create your own db, add tables, add fields to a table, move fields and tables from a place to another and generate the sql-script that builds your db (that is, creates all the tables). The main purpose of Druid is not just to manage tables and generate scripts. An important aspect of a database is its documentation. Druid lets the DBA specify all relevant information for tables and fields and then it generates full html/pdf output describing the purpose of each table and field. Last, but not least, Druid has many facilities for the developer of database applications: data dictionaries, summaries, class generation and much more.

2.3 Features

Here is a list of features present in Druid:

2.3.1 Basic features

- **DBMS independence**

  Druid pushes DBMS independence to its limits. Datatypes are completly customizable and are not built into the Druid's core. If you need a particular datatype, present only in a specific DBMS, you can add it to your project. The only thing you must tell Druid is if the datatype is constant or variable (see chapter 3 for more information). You can also customize field attribs (like primary keys, unique, not null) and add attribs present only in specific DBMS (like the less used sql's DEFERRABLE attrib present in Oracle). This lets you use the full power of your DBMS (see chapter 4 for more information). If this is not enough for you, Druid lets you add your own custom sql code for tables and the databases in general.

- **Strong datatypes**

  Druid's datatypes can have a domain. The user can specify a range or a set of values the datatype must belong to. Domain checks are added to the sql-script file to offer the best protection against data faults. Other operations include datatypes merging, remapping and usage. Serial types are also handled.

- **Several databases in the same project in hierarchical view**

  You can create and edit as many databases as you want. In the project view, are listed all the project's databases and can switch from one to another with a simple mouse click. All the project's elements are listed in the project view and are grouped hierarchically. Druid lets you use folders to group tables and other folders. This is very useful when the database becomes large.

- **Cut, Copy and Paste for every database's entity**

  When editing a database, it is common practice to move a field from one table to another or to add fields with the same structure. It is also common to add a new table that is very similar to an existing one. Druid lets you cut, copy and paste every database's entity thereby speeding up database both creation and maintenance.

- **Data generation**

  Druid can generate a rich set of documents: Data dictionary, summary files, database documentation(in both html and pdf format). Futhermore, Druid creates files or classes (in Java, C, C++) to store a fields' length. The developer can also define variables for a table and let Druid generate the code to include in the table's class (table vars are described in TABLES AND FIELDS).

- **Command line interface**

  Druid can be run by a command line interface to generate data without using the GUI.

- **Handled objects**

  Druid handles tables, fields, views, indexes, procedures, functions, sequences, field constraints, triggers and lets you add revisions. You can add DBMS specific features too.

- **Full plug-in architecture**

  Each Druid's module is separate from the main program. Developers with particular needs can easily write new modules by themselves.

2.3.2 JDBC features

- **Advanced driver management**

  Drivers are installed with a simple file dialog. Druid takes care of loading each driver and testing if it is a jdbc driver. When you connect to a database you don't have to specify a driver. Druid scans all loaded drivers and automatically chooses the best driver.

- **All database information is retrieved**

  You can view all database information, from general info (datatypes, result-sets, keywords) to specific info (table structure and references). Almost all objects are handled (tables, views, synonyms, procedures, functions, schemas, sequences).

- **Execute queries**

  Druid has a sql navigator that lets you create several queries and execute them. Results are displayed in a grid fashion and can be saved into a file. Druid can save data in several file formats (eg. tab delimited).

- **Edit table's records**

  Druid has a powerful record editor. You can insert / modify / delete records using a simple grid. It is possible to edit a record individually to edit large data. Results can be saved to a file in various formats or can be exported in a special file format (the Druid data format) that can be reimported later. This works for all drivers, whether they have updatable result-sets or not.

- **Structure exchange / reverse engineering**

  It is possible to import the full database structure into Druid for reverse engineering[1]. You can also send your Druid project to the jdbc db to rebuild it "on the fly" and perform some basic operation on the jdbc db (drop objects, refresh).

[1] Druid's ability to retrieve all the database schema depends on the JDBC driver. If the driver is fully compliant to the JDBC 2.1 API then Druid can retrieve all the information.

2.3.3 E/R features

- **Use of the IDEF1X notation**

  You can draw a particular E/R diagram to represent relationships between entities (a table or a set of tables). Each entity can be documented and fully customized. The created diagrams can be saved to disk or printed and become part of the final documentation.

Chapter 3

Datatypes

3.1 Types, sweet types

Datatypes are a fundamental concept that a DBA must know and master. Each piece of information stored in a database must have a type. Even though is not mandatory to use several types (it is possible to create a db with only a string type), the use of them gives one more dimension to the db structure allowing the DBA to classify the data. Common sql types are : integer, date, char and varchar, decimal. In addition to these types, several DBMS implement some types which are not compliant with sql-92 in order to fill specific cases. Examples of this include blob, clob and counter. With all these types, a tool for handling tables should implement them all, because the tool's developer doesn't know which DBMS will be used.

3.2 Types supported by Druid

As stated before, Druid is completely DBMS independent regarding datatypes because it uses an original approach. Consider all types you know : int, smallint, date, varchar, blob etc... As you can see, these types can be subdivided into two classes. The first class, called by Druid "Constant Types", contains types like int, smallint, blob, date. These are types that are auto-contained, that is, the type is concrete and be instanciated. Note that the word "constant" doesn't refer to the type itself. A blob is a type of variable size, but Druid considers it constant. The second class, called "Variable Types", contains types like char, varchar, numeric and decimal. As you can see, these types are abstract because you can't use them "as is". You must supply one or more values to make them concrete. For example, you cannot use a varchar, but you can use a varchar(24). You cannot use a numeric, but can use a numeric(10,4) (however, in some DBMSs if you don't supply a value a default one is choosen).

All types I have found until now can be classified as constant or variable, so giving the user the possibility of adding types for both classes leads to a tool that is datatypes independent. The Druid's GUI for datatypes has a tree structure with two branches, constant size and variable size, that represent the type's class. Adding a child to the constant branch, makes the type constant. The same goes for the variable branch. In figure 3.1 you can see the datatype panel with some

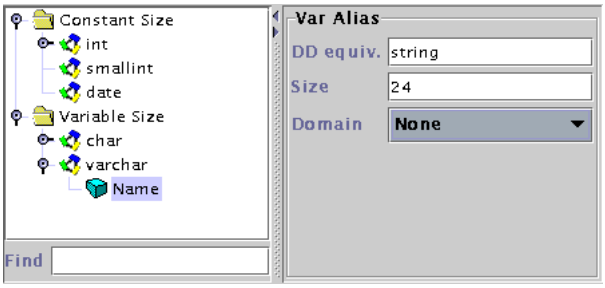

Figure 3.1: An example of datatypes

types defined.

3.3 Aliases

An alias is another name for a datatype. The use of aliases is convenient for constant types and is mandatory for variable types. The best approach to explain how aliases work is to start with an example.

Suppose you want to create a DB to store information about people. Consider these fields:

Field	Description
code	Used as primary key to identify the person
name	the person's name
surname	the person's surname
birthdate	obvious
height	the person's height (in inches , feet or whatever you want)

The first thing to do is to estimate a maximum size for name and surname fields (in chars). Let it be 24. To add this type in your Druid project do these steps (I assume you have already created a database):

- Select the database node (in the project view, on the left of the window)
- Select the "datatypes" tab in the right panel
- Right click in the "variable size" folder and choose "Add basic type" from the popup menu
- Click on its name and change it from "Unnamed" to "varchar". At this point you have created your first variable type!
- Right click on it and choose "Add alias"

- Click on its name and change it from "Unnamed" to string
- Put the value 24 in the size's box

Now you have a variable type (varchar) and an alias (string). This alias will be used for both name and surname fields. Note that you cannot use the "varchar". When Druid generates the sql-script it will resolve the alias (string in this case) this way:

- It takes the basic type's name (varchar in this case)
- It combines this name with the alias size (including the two braces) building "varchar(24)"
- It substitutes the alias name (string) with "varchar(24)"

At this point we have built a type for both name and surname. Let's build a type for height. The first step is to choose a sql type to store it. Suppose we have a height with decimal values (like meters). In this case a "numeric(2,2)" sql-type is appropriate, let's build it with Druid. The steps are the same for the string type. The difference is that the basic type is now "numeric" and the alias is, for eg., "height" with it's size "2,2". With the previously mentioned resolution rule, Druid will substitute the "height" type with "numeric(2,2)". Note that Druid doesn't care about what you type in the size box. If you type "2,3,4" Druid will substitute "height" with "numeric(2,3,4)". This gives you the maximum flexibility, even though the type check is demanded by the DBMS.

Now, let's talk about constant types. We want to define the "date" type. The steps are similar to those for the variable types:

- Right click in the "Constant size" folder and choose "Add basic type" from the popup menu
- Click on it's name and change it from "Unnamed" to "date"

For the code field we will use a different approach to show how constant aliases work. The basic type for the code field is an "int". This is usual for primary keys. Repeating the steps performed to add the "date" type we can add the "int" type. Now we can use this int type, but let's go further. In many cases, when creating a database, we are forced to attach several meanings to the same datatype (called semantics). Take the sql int type for example. You can use the int type for : primary keys (in this case the int is a serial for you), bits, a custom datatype packed into an int or to store numbers in general (like height, width, weight and so on). If you could give a different name to these uses of the int type the database would be more readable. This can be accomplished with constant aliases and in this case we can perform the following steps:

- Right click on the "int" type and choose "Add alias"
- Rename the alias to "Serial" (don't confuse this with Postgresql' serial type)

At this point we have the "Serial" type that can be used for the code field. Note that both "int" and "Serial" types can be used because they are concrete.

The resolution rule for constant aliases is (for constant basic types the type's name is used):

- Take the basic type's name ("int" in this case)
- Substitute the alias name (Serial) with "int"

Again, Druid doesn't care about what data you add. The datatype check is demanded by the DBMS.

3.4 Domains

A datatype is usually associated with a domain. From a math point of view, given a variable, a domain is the set of values that the variable can assume. These values represent a constraint for the variable and specialize the variable's type. For example, suppose you want to define a datatype to represent the age of people. A good choice is the int type but, without constraints, a user can insert a negative age (which should not be allowed). In this case we can define a domain telling the DBMS to check if the value is inside (or outside) the range.

Druid lets you specify one of five domains for a datatype. These are:

- **None**

  This domain doesn't perform any check on the datatype and can be used if the domain is not important or not necessary. It is commonly used to text and dates.

  There is no check rule.

- **Lower case text**

  This domain is used only for text types and checks the case of the input text. If the user tries to insert (or update) text which contains upper letters the DBMS raises an error.

  The check rule is : field = LOWER(field)

- **Upper case text**

  Same as above, but for uppercase text.

  The check rule is : field = UPPER(field)

- **Range of values**

  Usable for ordered datatypes (like int, decimal, numeric, dates etc...). The user can define a range and ask the DBMS to perform a check if the value falls either outside or inside the range. In figure 3.2 you can see an Age datatype using this kind of domain. The range is 0..120 (including both 0 and 120). If you need to check the range outside the given one,

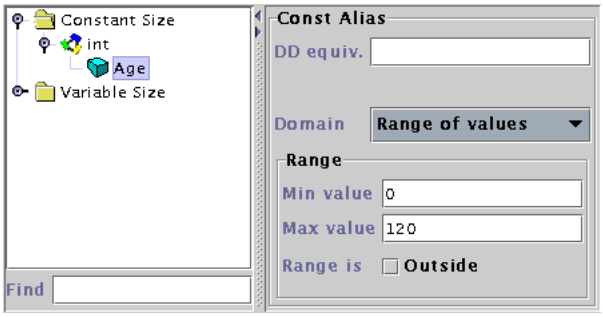

Figure 3.2: An example of the range domain

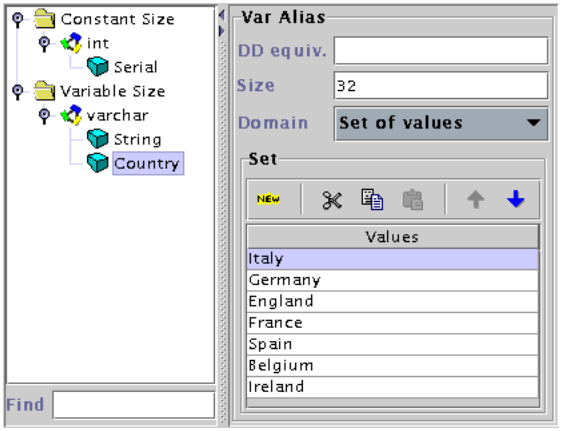

Figure 3.3: An example of the set domain

then mark the "Outside" checkbox. In this case, if the checkbox is set, the DBMS raises a check error if the value is inside the range 0..120 (including both 0 and 120).

The check rule is : field [NOT] BETWEEN min AND max

- **Set of values**

  This domain lets the user specify a set of values the value must belong to and it is used for text types (like char and varchar). The DBMS raises a check error if the value the user is trying to insert doesn't belong to the given set. In figure 3.3 you can see an use of this domain. Suppose you want a datatype to represent european countries. The type is a varchar, but if you want to be sure that all records contain valid countries you must list them. Note that Druid automatically adds the two single quotes.

  The check rule is : field IN('value-1', 'value-2', ... 'value-n')

3.5 Datatypes operations

There are some operations on datatypes that Druid can perform. These operations automate some long tasks and let the database designer to easily adjust the schema.

3.5.1 Remapping

It may happen that a datatype is placed in the wrong class (constant instead of variable or vice versa) or in the wrong subclass (basic instead of alias or vice versa). This may be due to several reasons:

- A wrong database design
- A change in the project requirements / specs

Druid lets you move a datatype from one class to another, remapping it if needed. This operation is transparently performed when you use the cut and paste facility. When you cut a constant type and then paste it into a variable type, Druid transforms the type into a variable alias. Note, however, that the remapping is a lossy operation because the destination type might not have all the attribs of the source type. So, you can safely remap a constant type to a variable alias, but if you remap a variable alias to a constant alias you lose the 'size' attrib.

This feature is also very useful when importing a db structure into Druid via jdbc. If Druid doesn't map the type correctly, you can remap it manually.

3.5.2 Merging

In some cases you may want to scan all table fields and change a certain datatype into another. This may be due to the fact that you want to remove the source datatype or because the datatype doesn't fit a new need (for example, you have a clob type and you need to store binary data in it). To change the datatype you have to scan the fields of all tables and, when you find the target datatype, change it into the new type. When you work with a big database this operation becomes a tedious and time consuming task.

To solve this problem Druid lets you change all instances of one datatype into another. This feature is activated by selecting a datatype and choosing "Merge with..." from the popup-menu. Druid displays a dialog showing all datatypes and asks you to select a target one (with the right mouse button). When the operation is completed, the source datatype is not used anymore and may be safely removed.

3.5.3 Usage

When selecting a datatype (other than the basic, variable type) you may select the 'Usage...' option from the popup menu. This feature lets you know all tables and fields that use the selected datatype. Note, however, that the list doesn't include fields that are foreign keys.

3.6 The "DD equiv." box

In this box you may put a string that represents the type (like "integer", "string", "date"). This string will be used when Druid generates the data dictionary (see chapter 6 for more information).

3.7 Examples

Here are some examples of useful datatypes:

Field	Description	Domain	Basic type
Serial	A datatype to indicate a serial counter	None	int
TDate	A custom date represented as an int	None	int
Byte	A datatype to store a single byte	Range (0..255)	numeric(1)
Quantity	A generic quantity type	None	numeric(10,4)
Char	A single char	None	char(1)
Bool	A boolean type	Set ("y", "n")	char(1)
Name	A generic uppercase name	Upper case	varchar(24)
Memo	A type to store notes	None	varchar(4000)

Chapter 4

Field Attributes

4.1 Introduction

Simply speaking, attribs are those "keywords" you put after a field when you create a table. Common attribs are "NOT NULL", "UNIQUE", "PRIMARY KEY", "DEFAULT" and they may refer to either a single field or a group of them. The main issue with attribs is that if you want to develop a tool like druid you have to make a big choice between remaining compliant with the SQL-92 standard or specializing the tool for a particular DBMS. If you choose to develop a general tool, users cannot use some advanced features of their DBMS. If you specialize your tool, another question needs to be answered: which DBMS should I choose? Both choices have their pros and cons.

This problem has been addressed allowing custom attribs. If you look closely at them, you can figure out that all attribs have the same structure, so it becomes easy to build a structure to handle them all. This way, if you need to use a particular attrib of your DBMS you may add it to Druid (one example is the sql's "DEFERRABLE" attrib used by Oracle).

Furthermore, users can create custom attribs for special purposes in order to have custom data in the data dictionary or to create indices.

4.2 Druid's attrib structure

A druid's attrib is composed of the following parts:

- **Name**

  This name is used by the GUI to refer the attrib. It should be short because it is used as a table's column header.

- **Sql name**

  This name is used by the sql-script generator and represents the attrib's name in standard sql. For a primary key you may use "PrKey" for the name, and "primary key" for the sql name. Note that you can use this field to create an attrib which represents a complex

sql option that uses a special feature of a DBMS. For indexes, this field is used to store a custom template for the index name.

- **Type**

  For attrib types look at section 4.2.1.

- **Scope**

  For scope types look at section 4.2.2.

- **Description**

  This is simply a description to remind you the purpose of the attrib.

- **Use in data dictionary**

  When you create a new attrib you may or may not want it in the generated data dictionary. If this flag is set the data dictionary generator adds the attrib's column (which can be a boolean value or a string depending on the attrib's type). For example, you may wish to define a custom attrib that is present only in your data dictionary, but not in your sql-script.

- **Use in summary**

  Same as above.

- **Width**

  This is a GUI parameter and specifies the width of the attrib's column in the table editor. Some attribs (for e.g. bool attribs, like unique and not null) may require less video space than others (for eg. the default attrib, which must contain a value).

Note that Druid has no concept of primary keys. For Druid a primary key is just an attrib like all the others, but to maintain future compatibility I recommend you put it at the first position.

The order of attribs you create is reflected in the table editor.

4.2.1 Attrib types

An attrib can be of type bool, int or string. The type specifies if the attrib takes a parameter or not. For example, the "unique" attrib has no parameters, while the "default" attrib must be followed by a string or an int. Here follows an explanation of attrib types:

- **bool**

  Is used for attribs that don't take parameters. Examples are "UNIQUE", "NOT NULL", "PRIMARY KEY", "DEFERRABLE" and indices. This type is represented with a check-box in the field editing GUI. When a field has an associated attrib of this type (that is, the checkbox is set), the attrib's sql-name is added after the field's declaration.

Name	Type	PrKey	Unq	NotN	MUnq	Def	Idx1
code	Serial	☑	☐	☐	☐		☐
surname	UpperString	☐	☑	☐	☑	'DOE'	☑
name	String	☐	☑	☐	☑	'john'	☐
birthDate	date	☐	☑	☐	☑		☐
isDeceased	Bool	☐	☑	☐	☐	'n'	☐
age	Age	☐	☐	☐	☐		☐

Figure 4.1: An table with several field attribs

- **String and Int**

  These types are used for attribs that take parameters (an example is the "DEFAULT" attrib) and are represented with editable text in the field editing GUI. When a field has an associated attrib of these types (that is, the editable text is not empty), the attrib's sql-name and the given value are added after the field's declaration. An int type is added "as is", while a string type is added with single quotes. These two types are basically the same and the difference is that a check is performed on the int type to see if the given data is a proper int.

As stated before, the attrib's type changes the attrib's appearance in the GUI. Figure 4.1 shows the table editing GUI with the bool and string types in action.

4.2.2 Scope types

A scope is a concept used to classify an attrib. Depending on the scope, the attrib can influence a single field, a group of field or the entire table. Druid defines the following scopes:

- **Field**

  This scope is used for attribs that affect only a single field. Examples of such attribs are: "unique", "not null", "default". These are attribs that follow the field's declaration (with or without parameters). An attrib that belongs to this scope can be of any type (bool, int or string).

- **Table**

  Used for attribs that affect groups of fields. Examples are: "primary key" and "unique" for multiple fields. When several fields have set one attrib of this scope, the sql generation puts together the attrib's sql-name and the field names to form something like "primary key(field1, field2)" or "unique(field1, field2)". Attribs of this scope are generated after all field declarations. As you can imagine, an attrib that belongs to this scope can be of bool type only.

- **Index**

  This scope is used to define new table indices. Here the sql-name can be used to supply a template for the index name. A usefull template is {table}IDX{cnt} where table will

Name	Sql name	Type	Scope	In DD	In Sum	Width	Descr
PrKey	primary key	Bool	Table	☐	☑	60	Primary key fo...
Unq	unique	Bool	Field	☐	☑	60	Unique for a si...
NotN	not null	Bool	Field	☐	☑	60	
MUnq	unique	Bool	Table	☐	☑	60	Unique for sev...
Def	default	String	Field	☐	☑	60	Field's default ...
Idx1		Bool	Index	☐	☑	60	An index

Figure 4.2: Field attribs with several types and scopes

be replaced with the table's name and cnt is a generic counter. See 5.5.6 for a complete list of substitutions. The type must be bool. With this template, given a "Patients" table, the sql generator will generate the following indexes: "PatientsNDX1", "PatientsNDX2", "PatientsNDX3" and so on.

- **Unique Index**

  Same as above, but for unique indices. A custom template replaces the standard one.

- **Custom**

  This is not a proper sql scope. It refers to attribs used for special purposes. For example, when developing an application that requires some data to be attached to each field. The usage of this scope is well explained with an example.

  Suppose you are developing a client-server application. The server may supply an operation like "add values v1,v2,v3 in the table X". A problem arises when table X contains fields that must not be modified from outside (that is from clients), because they are for internal use. In this case, the server must check if each input field can be modified. To do this, it must know if a field can be modified or not for each table. Druid solves this problem by letting you define an attrib like "can be modified", with type = bool and scope = custom. Then, in the field editing GUI you can set this attrib for each field. At last, you let Druid generate a data dictionary for each table in the database containing the rows for "table-name", "field-name" and "access modality". At this point your server can load this data dictionary and implement its access policy.

Figure 4.2 shows some field attribs with different types and scope.

4.3 Summary

The following table summarizes the attrib's structure. Each column is a type and each row is a scope. Each cell explains the attrib's properties for the given type and scope.

	Bool	**Int**	**String**
Field	Attribs for a single field that may or may not follow the field declaration (like "NOT NULL", "UNIQUE", ”DEFERRABLE”)	Like the string type but for numbers	Attribs for a single field that may or may not follow the field declaration, but that require a string value after them (like "DEFAULT")
Table	Attribs for groups of fields (like "PRIMARY KEY" and "UNIQUE")	(meaningless)	(meaningless)
Index	Specify an INDEX	(meaningless)	(meaningless)
Unique Index	Specify a unique INDEX	(meaningless)	(meaningless)
Custom	Custom properties. For example, you can define the "canInsert" and "canUpdate" attribs to declare the attrib’s access restrictions. Then your server app can load the generated data dictionary and use this info to restrict a client’s access.	Custom number to associate to a field	Custom text to associate to a field

4.4 Examples

Name	Sql Name	Type	Scope	Description
PrKey	primary key	bool	table	The primary key of a table
PrKeyS	primary key	bool	field	A primary key composed of only one field (rarely used)
Unq	unique	bool	field	Unique attrib for a single field
UnqM	unique	bool	table	Unique attrib for a group of fields
NotN	not null	bool	field	The not null attrib
Dfr	deferrable	bool	field	Deferrable attrib used in Oracle
Def	default	string	field	The default string value of a field
Idx	{table}IDX{cnt}	bool	index	A simple index on a group of fields with a classic template for its name.
IdxU	{table}_{fields}	bool	index-u	A unique index using a different template. {fields} will be replaced with all field names separated by an underscore '_'.
Ins	(meaningless)	bool	custom	A bool attrib to store custom data. In this example "Ins" means insertable with a bool type of yes/no

Chapter 5

Database objects

5.1 Introduction

Druid has no limits to the complexity of the database. You may add as many entities as you want (tables, fields, etc...) and when the database complexity raises you may use folders to organize your entities. Regarding entity names you have complete freedom (that is druid makes no checks) but a few rules should be followed in order to obtain consistent data:

- Don't use spaces to separate names. For example the name "Patient Data" is not valid because the sql-script that gets generated is not valid.
- Some names (like "Data") could be considered as sql keywords.
- More entities with the same name can cause problems to some modules (like the XHTML one) because they use the entity's name to create files on the file system.

5.2 Documentation

Almost every druid's object can be documented. After selecting it (a table, a field, a folder etc...) you should be able to see the Docs tab in the right side (see figure 5.1). Here you can write all docs you want, change the font, color, alignment, add images and more. However, keep in mind that the doc editor have some bugs. For example, if you add an image to an empty doc the image is not shown.

5.3 The Database

5.3.1 Revisions

When working with a database, it is useful to keep track of changes. A facility offered by Druid is the project build, which is incremented during each save, but the user has no control over it.

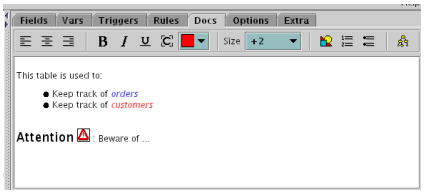

Figure 5.1: An example of the docs editor

Revisions have been introduced to fit this purpose. They are like comments the user adds to keep track of the current database development and don't affect the project.

Basically a revision has this structure:

- **Version**

  Here you can store a string representing a database version. Common strings are: "0.1", "v 0.3 alpha", "0.9 pre1" and so on.

- **Date**

  The date of the revision. This field is added automatically by druid, but you can modify it. This field is a string and is not restricted to dates.

- **Description**

  A description of the current database status. Here you can indicate what was added / removed / changed with respect to the previous revision.

5.3.2 Extra sql fields

Druid is a database independant tool but in most cases the user will want to use some special features of his DBMS. To accomplish this, two sql fields have been introduced:

- **Pre sql**

  The sql code you put here is added to the sql script BEFORE the sql that creates the tables.

- **Post sql**

  The sql code you put here is appended to the sql script, that is AFTER the sql that creates the tables.

With some DBMSs (like PostgreSQL) you can place "BEGIN WORK;" in the pre-sql field and "COMMIT;" in the post-sql field. Using these commands the sql script is executed as a whole or not at all. This way, if the script fails you don't have to delete all created objects manually (works greatly on PostgreSQL because PostgreSQL's serial type creates a sequence, which must be deleted separately).

5.4 Folders

Folders are what their name indicate and can contain any kind of objects. Their purpose is to:

- Keep related tables together
- Structure a big database hierarchically so that tables can be easily found

5.5 Tables

5.5.1 Fields

Table's fields can be edited using the tree view on the left side or, more easily, using the Fields tab of the table node. Beside the field's name there is its datatype and all field attribs you have defined in the database. To change the datatype simply click on it to rise the Datatype selector, from which you can choose another type or a foreign key to another table.

5.5.2 Table vars

Table vars are a feature used mostly by developers. To explain what they are and how they work let's consider the following scenario:

Scenario: You have to design a client (that can be an application, an applet or a web page) that accepts user input and sends it to a DBMS. One data that the client can accept is the user's name, which is represented as a varchar(24) in the database. If the user enters a text longer than 24 characters the DBMS rises an error and you don't want this to happen. The only way to prevent this (I suppose that the textfield you are using doesn't let you specify a maximum size) is to add a few lines of code that perform a check between the text's lenght and 24. However, hardcoding 24 is not a recommended way to do this because this value may change. The best solution is to use a constant that is auto-generated and that follows the database changes.

Given this scenario it should be clear what table vars are. When you create a table which has fields of variable size (that is its datatype is of variable class) you can ask druid to generate a file or a set of files (depending of the programming language you have choosen) which contain(s) the constants.

Example : If you create the table 'Users' with the fields:

- code (int)
- surname (varchar(32))
- name (varchar(24))

Name	Type	Value	Descr
GLOBAL	Int	0	a global administrator
LOCAL	Int	1	a local administrator
USER	Int	2	a simple user
OTHER	Int	3	other (for example a guest)

Figure 5.2: Constants for a generic "Users" table

Druid will generate the following code (I suppose that the java language has been selected):

```
public class Users
{
    public static final int SURNAME_SIZE = 32;
    public static final int NAME_SIZE = 24;
}
```

Obviously, there is no constant for the 'code' field because its size is constant.

In addition to these vars (that are automatically generated) the developer may manually add some vars to accomplish specific tasks. Let's consider this scenario:

Scenario: You have the previously created table 'Users' with one more field: userType (int). This field contains the type of the user and may contain the following values:

0	For a global administrator
1	For a local administrator
2	For a simple user
3	Other (for example a guest)

When you write your client application it is not a good idea to hardcode these numbers because they may change and because your code gets not readable to others. The best solution is to use constants. Druid let you define several variables for a table and puts their code together with the auto generated vars. To add a variable simply select a table, select the 'Vars' tab and press the 'new' button. Figure 5.2 illustrates the four variables (or constants) of our scenario.

Each variable has the following attribs:

- **Name**

  This is the name of the variable. You refer to the variable using this name.

- **Type**

  The variable's datatype. Druid supports the 'bool', 'string' and 'int' types

- **Value**

  This is the value you want to give to the variable

- **Description**

  This description will be used as a comment during code generation.

Referring to the previous scenario, druid will generate the following code:

```
public class Users
{
    public static final int SURNAME_SIZE = 32;
    public static final int NAME_SIZE = 24;

    //--- a global administrator
    public static final int GLOBAL = 0;

    //--- a local administrator
    public static final int LOCAL = 1;

    //--- a simple user
    public static final int USER = 2;

    //--- other (for example a guest)
    public static final int OTHER = 3;

}
```

5.5.3 Triggers

5.5.4 Rules

5.5.5 Sql commands

It may happen that a vendor ships an application that uses a DBMS to work. Applications like this one must create their working database and it is common, after the creation, to fill the database with default data. This data is usually shipped together with the sql script in the application's package. When the application is under development, these data should reflect every change made in the db. Druid lets you define these data as INSERT(...) commands. In the Extra ▷ Sql commads panel of a table node you can add as many INSERT statements as you need. During the sql script generation, these commands will be appended at the end of the table's schema.

5.5.6 Template names

If you select a table node, in the Optisons ▷ General tab you can see the Template for names panel. Here, and in each sql generation module, you can define the templates for the names to generate. You can see three textfields:

Primary keys template for primary keys

Foreign keys template for foreign keys

Other template for other table constraint (like unique)

If the template is empty it won't get generated. A tipical primary key constraint name is <table-name>_PK. To use such template simply type:

```
{table}_PK
```

Other recognized keywords are:

{table} will be replaced with the table's name.

{fields} will be replaced with the names of the fields that form the index (separated by "_").

{cnt} will be replaced with a counter that represents a progressive number.

5.5.7 Multiple FKeys

Support for multiple foreign keys is automatic. The general rule is:

Rule: Druid merges a table's fkeys when its fields refer different fields of the same target table. These fields must be one after the other in the table's schema.

This is difficult to explain in words. Maybe looking at the algorithm may help:

- Druid scans all fields of a table and collects all fields that refer to other tables (retrieves all foreign keys).
- Then Druid scans all collected fields and sees if the current field and the next one refer to the same table but to different fields. If this is the case then the two (or more) fields are merged together.

Let's look at some examples:

Example	FKey generated	Notes
Table B (code field1 A(code) field2 A(code))	fkey (field1) refer A(code) fkey (field2) refer A(code)	Both field1 and field2 refer to the same field. No merging required.
Table B (code field1 A(code1) field2 A(code2))	fkey (field1,field2) refer A(code1,code2)	Field1 and field2 refer to different fields of the same table. Merging occurs.
Table B (code field1 A(code1) field2 A(code2) field3 A(code1) field4 A(code2))	fkey (field1,field2) refer A(code1,code2) fkey (field3,field4) refer A(code1,code2)	Field1 and field2 are merged but field3 refers to a field already referenced. In this case field3 starts a new foreign key group. Field4 is merged with the previous group (formed only by field3).
Table B (code field1 A(code1) field2 A(code1) field3 A(code2) field4 A(code3))	fkey (field1) refer A(code1) fkey (field2,field3,field4) refer A(code1,code2,code3)	Field2 starts a new group. Field3 and field4 can be added to this group because they reference different fields.

5.6 Views

Views are supported, but without GUI facilities. Adding a view is similar to adding a table: just right click on a database or a folder and select "Add view" from the popup menu. At this point, you can see your view's node in the project view (on the left) and you can edit the sql code in the work panel (on the right).

During sql generation, views are added at the end of the database tables and a ";" is added if it is missing.

5.7 Procedures and functions

Support for these objects is limited and is the same as for views. To add a procedure do a right click on a database or a folder and select "Add procedure" from the popup menu (same as for functions).

5.8 Sequences

Sequences are objects that generate progressive numbers, often used for primary keys. Druid handles the following aspects of a sequence:

- **Increment**

  The value to add to the current value to obtain the next

- **Min value**

  Minimum value allowed

- **Max value**

  Maximum value allowed

- **Start**

  The initial sequence value

- **Cache**

  Specify how many values must be kept in memory for faster access

- **Cycle**

  Indicates that the sequence must continue to generate numbers when it reaches it maximum or its minimum value

- **Order**

  Specify that sequence numbers are generated in order of request (less used)

Usually all options can be omitted and some (like cache, order) are not accepted by all DBMSs.

To add a sequence do a right click on a database or a folder and select ”Add sequence” from the popup menu.

5.9 Notes

The purpose of a Notes object is to contain the documentation that don’t fit into a particular object. It is not used during normal operations and contributes only when generating docs. Each note has a type that classify the note itself. The type can be info, alert or danger and you choose between them depending on your needs.

To add a note do a right click on a database or a folder and select ”Add notes” from the popup menu.

Chapter 6

Data generation

6.1 What gets generated ?

Once you have developed the database schema, you certainly need to create this schema into your DBMS. Usually this work is done by a sql script that creates the schema objects one after another using sql commands. At this point the main work is finished but there are several secondary works that are part of the database and that can help during developing. Druid can automatically generate the following kind of files:

- **The sql file**

  This is the file I talked about. This is an ascii file with sql commands that create the database objects. You have several modules to generate the sql script, one for each database you use.

- **Documentation**

  The documentation is an important aspect of the database. It explains the purpose of tables and fields, why a field exists, various relationships between objects and so on. You can generate the documentation in several formats, depending on your needs.

- **Code**

  An important aspect that Druid takes into account is the generation of programming aids. Druid can generate source files / classes in several programming languages. To see how this feature can be used see section 5.5.2 or the MODULE REFERENCE MANUAL.

- **Summary**

  As its name suggest, a summary is a short description of your database schema. You can use it as a fast reference.

- **Data dictionary**

  This is an ascii file that contains the database data dictionary, that is a list of table names and fields. It is mainly used for programming and its format depends on the module that

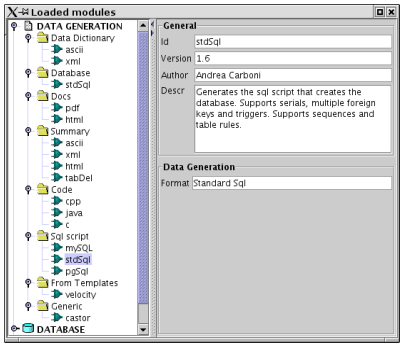

Figure 6.1: The data generation modules

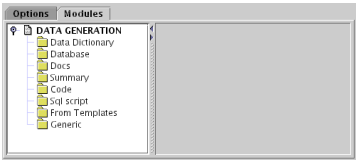

Figure 6.2: The generation module panel

generated it. The standard ascii module generates a file that can be easily parsed and stored into a database table.

If you select Config ▷ Modules from the main menu, you open the Loaded modules dialog that show all loaded modules. In the Data Generation folder are located all modules that concern data generation. You should see a dialog like the one in figure 6.1.

Druid has a plug-in architecture for modules. This means that you can download and install modules after the Druid main program has been installed. Every Druid package comes with all modules available so you don't have to download them separately. In any case, to know more see the MODULE REFERENCE MANUAL.

6.2 Adding modules to a project

To generate a kind of data file you need to add the proper module to your project. To do this, select the database node and then the Generation tab. You would see a panel with two tabs: Options and Modules. Select the second one and you will see something like figure 6.2. Here you have a set of folders, one for each kind of module. To add a module to your project simply do a right click on a Folder, choose Add module and the module you want to add.

Once you have added a module you can configure it using the panel on the right, which is divided into two parts. The top part is common to all modules and contains or a path or a filename (it depends on the module) where to generate data. The bottom part contains all options specific

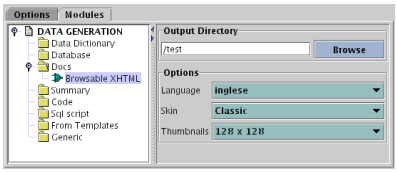

Figure 6.3: The sql generation module

to a module so it can or cannot be present. Figure 6.3 shows the xhtml docs generation module with its options at the bottom of the right panel.

6.3 Data generation

Data can be generated from the Data generation panel (see 6.2). You can do a right click on a node and choose Generate to activate all modules that are children of the selected node. For example, referring to figure 6.3 if you do a right click on the xhtml module you will activate only that module. If you do a right click on the Docs node, you will activate the xhtml module and all other docs modules. If you do a right click on the Data Generation node, you will activate all modules added to the project.

6.4 Customization

Some modules can be easily customized in what they generate. For example the html docs generation module uses some templates that you can change. See the MODULE REFERENCE MANUAL for more information.

Chapter 7

The command line interface

7.1 Introduction

When several developers / dbas work on the same project each person handles only one part of it. It is a common common practice that a DBA maintains the database, modifing tables and fields, while other developers use the new database structure supplied by the DBA. They may receive the sql-script that generates the database or the database itself in another format (for example a Druid file). In the last case developers must run the program used to create the database and regenerate the script file. Druid simplifies this task supplying a command line interface which allows developers to drive some Druid's action without running the GUI. This feature is best used in scripts.

7.2 Parameters

Druid takes several parameters, which may have an arbitrary order. Each parameters has the following structure (note the "-" prefix):

```
-<PARAM>[:<VALUE>{,<VALUE>}]
```

So, common parameters have this form:

```
-PARAM
-PARAM:VALUE
-PARAM:VALUE,VALUE
-PARAM:VALUE, ...  , VALUE
```

Druid accepts the following parameters:

-proj Indicates a Druid project and must be supplied.

Example: `-PROJ:DATABASE.DRUID`

-gen If given, this parameter tells Druid to generate the database data. Some values must be supplied in order to specify what must be generated. Accepted values are:

sql	generates the sql-script
docs	generates docs
datad	generates the data dictionary
summ	generate the summary
code	generate the code classes / file
generic	generate data related to generic modules
template	generate data from templates

Remember that the output files / dirs are those specified in the Druid file.

Example: `-GEN:SQL,HTML,SUMM`

-db A Druid file can contain several databases. This parameters indicates the database to work on. The value must be an integer starting from 0 (for the first database). If the Druid file contains n databases then 'value' must be in the range 0..n-1. If this parameter is not given, Druid performs its operations on all databases.

Example: `-DB:2`

-help Shows a brief summary of commands.

7.3 Starting with a specific project file

If only the -proj parameter is supplied, Druid starts the GUI and opens the given project. Example:

```
java -jar druid.jar -proj:myproject.druid
```

Chapter 8

Ant integration

Druid can be run as an ant task. Here is an example of BUILD.XML file that calls druid:

```
<project name="test" default="test">
   <taskdef name="druid"
         classpath="/my/src/druid-project/druid/libs/ant-task.jar"
         classname="druid.AntTask" />
   <target name="test">
      <druid command = "-gen:sql -db:0 -proj:/.../database.druid" />
   </target>
</project>
```

As you can see, the taskdef tag defines the task named druid. Then, the druid tag is used to run druid. command is the only accepted attribute and must reflect the full command line that would be issued in a shell.

Note This druid version doesn't accept paths that contain spaces in the command attrib.