VALENTINA
Database Kernel

Paradigma (www.paradigmasoft.com)

Acknowledgments:

**Andy Bachorski , Andy Fuchs, Bill Mounce, Brian Blood, Craig A. Berry,
Dan Waylonis, David A. Bayly, Frank J. Schima, Guillermo Zaballa,
Hideaki Iimori, John Roberts, Lynn Fredricks, Paul Shaap, Robert Brenstein.**

Contents

Contents

Introduction

Valentina is an extremely fast, powerful and unique **Object-Relational** database engine. It is the result of concentrated, independent research started in 1993. The underlying technology combines the best features of two industry standard database models, «**relational**» and «**network**», and adds some new unique features to create one of the fastest and most flexible database solutions available today.

Valentina implements the **Object-Relational (OR) data model** (a theoretical development of Paradigma Software), which is an extension of traditional **Relational data model**. The OR data model contains the Relational model as an exact subset. This means that everything that works in a Relational Database Management System (RDBMS) must work in Valentina (we can compare this with the programming languages C and its object-oriented extension C++). This means that you can smoothly switch from the familiar RDBMS model to the modern model of Valentina.

Paradigma Software offers different solutions based on the Valentina database

- Valentina ORDBMS with GUI and AppleScript support	Mac
- Valentina C++ SDK	Mac/Win
- Valentina for REALbasic.	Mac/Win
- Valentina for Macromedia Director	Mac/Win
- Valentina XCMD	Mac
- Valentina for WebSiphon	Mac
- Valentina COM	Win

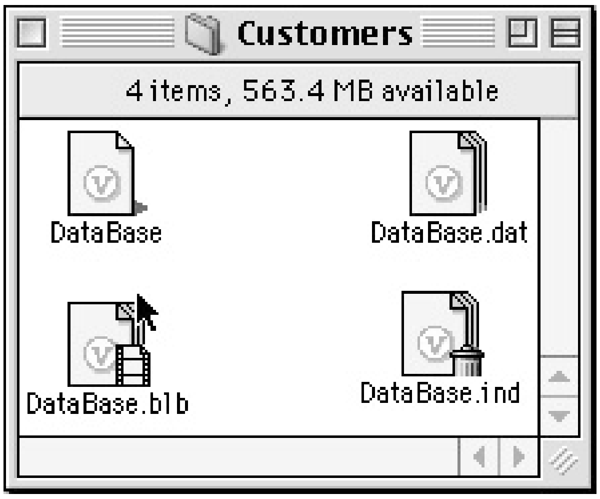
The Valentina database file format is cross-platform and consistent between all Valentina products. In other words, Valentina offers a solution to allow databases built for use with one development environment to be used with any other development environment supported by Valentina on any supported platform.

Characteristics of Valentina Database Engine

Max length of the disk file:	$2^{(32-1)}$ (2 GB)
Max number of Tables:	$2^{(32-1)}$ (2,147,483,647)
Max number of Fields in the Table:	2^{16} (65,535)
Max number of records in the Table:	2^{32} (4,294,967,295)
Max size of a BLOB record:	$2^{(32-1)}$ (2 GB)

Database of Valentina

Valentina has its own internal file system, so it can store many logical files in a single disk file. Valentina can store databases in one, two, three or four files:
«dbName.vdb» - description of the database
«dbName.dat» - major data of the database (Tables, Relations)
«dbName.blb» - contains BLOB (Binary Large OBjects), TEXT and Picture fields.
«dbName.ind» - temporary data of the database (indexes and any other temporary files)



The number of files used must be specified by the parameter «Mode of Database.Create()». The choices are:
1 - **“.vdb”**[description, data, BLOB, indexes]
2 - **“.vdb”**[description] + **“.dat”**[data, BLOB, indexes]
3 - **“.vdb”**[description] + **“.dat”**[data, BLOB] + **“.ind”**[indexes]
4 - **“.vdb”**[description] + **“.dat”**[data] + **“.blb”**[BLOB] + “**.ind”**[indexes]   // default

«dbName.vdb» file

Contains description of the entire database: tables, fields, relations, layouts, users, passwords and so on. This file usually is very small, just several KB. If the description file is separate (in modes 2, 3, and 4), it can moved to another computer, opened by Valentina, and a new, empty database will be created. Valentina will automatically generate the other files.

The extension .vdb for this file is optional in MacOS. If you are doing a cross-platform development, you need to specify this extension to allow the use of database files between platforms.

«dbName.dat» file

Contains the values of the records of the database.

«dbName.blb» file

Contains the contents of BLOB, TEXT and Picture fields.

An advantage to having a separate file for BLOB fields is that you can have a larger database: 2GB for «.dat» file and 2GB for BLOB fields.

«dbName.ind» file

Contains all temporary files of the database such as indexes and join tables. Some of these temporary files, in particular indexes, stay valid between sessions of database use. Valentina will remove others when the database is closed.

Advantages to having a separate index file are the following:
- Increase the upper size limit for a database, because now you can have 2GB for data, 2GB for BLOB and 2GB for indexes.
- Reduced risk of data corruption because part of the disk write operations will be performed on the other Index file.

This file can be deleted and the database will not be corrupted since Valentina will rebuild indexes when they are needed. This is useful if there was a crash while working with a database.

During development, if you begin to see incorrect results of a search and/or sort, you may consider deleting the index file and re-trying your application. You may get out of date indexes if you are developing with C++, REALbasic, Director, XCMD or any other language which has debug tools which allow you to interrupt execution of the application.

Properties of Database

Name
Each database must have a unique name in the scope of application. Name of database is case-insensitive.

DateTime Format
Valentina supports own format for Date and Time. It specifies how Date and Time fields will be converted to and from strings. This conversion affects any string operation of the kernel, including Import/Export and SQL queries.

A Database has the following properties:

DateFormat	- specify order for date: 0 - M/D/Y, 1 - D/M/Y, 2 - Y/M/D
DateSep	- separator for date, e.g. ‘/’
TimeSep	- separator for time, e.g. ‘:’

By default, the DateTime format of a Database is the same as the system format of the host computer. However, it can be changed on the fly at any time. For example, if you want to import a file with different settings for Date fields, then you can change the DateTime format, import that file and change the format back.

TimeOut
Specifies the maximium time (in seconds) of life for a Cursor of this DataBase. All Cursors that exceed this time are destroyed. On default the TimeOut is -1 (indefinite).

Elements of Database

BaseObject
- Each valid database must have at least one valid BaseObject.
- You can get the number of BaseObjects in a database.
- You can get a BaseObject from a database by name or by index (index start from 1).

Cursor
The result of an SQL (structured query language) command is a Cursor. The database automatically keeps track of all the Cursors. A Cursor will be removed when its lifetime exceeds the TimeOut value of the database.

Creation of Database

Two optional parameters can be specified when creating the database:

Mode
Specifies the number of disk files for a new database. The Mode of existing database cannot be changed.
1 - **“.vdb”**[description, data, BLOB, indexes]
2 - **“.vdb”**[description] + **“.dat”**[data, BLOB, indexes]
3 - **“.vdb”**[description] + **“.dat”**[data, BLOB] + **“.ind”**[indexes]
4 - **“.vdb”**[description] + **“.dat”**[data] + **“.blb”**[BLOB] + “**.ind”**[indexes]   // default

Segment Size
Specifies the minimum size at which the disk file will grow. It does not limit the database size to the specified value, but controls the overall number of segments in the database (e.g., a 1 MB database with 32 KB Segment Size would contain 34 segments).

In most cases, this parameter will not need to be changed. However, if the items in the database are small and numerous (e.g., a game with a database for the players, or a address book), a smaller size may be specified. It is not recommend to use a segment size greater than 64 KB.

The default value is 32 KB.

Note: If the description file is stored separately, then the Segment Size will always be 4KB.

BaseObject (Table)

Each database must have at least one BaseObject. The maximum number of BaseObjects for database is 2 Billion.

The term «BaseObject» is used instead of the traditional RDBMS term «Table» because Valentina uses Object-Relational model. In this model, BaseObjects can inherit attributes and data (i.e., one BaseObject can be implemented as several related tables that are invisible to the User and Developer). The result is that BaseObject appear to be a single Table. In the simplest case, when BaseObject has no parent, it is exactly the same as a Table. The BaseObject can, however, do more than a Table. For simplicity sake, the terms can be used interchangeably.

Properties of a BaseObject

Name
Each BaseObject has a unique Name in the scope of database. The Name is case-insensitive.

Identifier
A BaseObject has its own unique identifier inside of a database. In most cases, this property is used internally by the Valentina kernel.

Elements of a BaseObject

A BaseObject can be considered as a Table with Fields (columns) and Records (rows).

Fields
- Each valid BaseObject must have at least one Field.
- You can get the number of Fields in the BaseObject.
- You can get a Field from a BaseObject by name or by index (starting at 1).
- Each BaseObject has an internal virtual Field named «RecID» (this name is reserved by Valentina, so you must not use it for your fields) and type ULong (4 bytes). This field can only be accessed by name.

Records
- Each BaseObject can have between 0 and 4 billion records.
- Each record has a unique «RecID» (starting at 1).
- A «RecID» of 0 indicates an undefined record.
- Deleted records are marked as such and their space will be reused.

Field types

Valentina supports the following type of fields:

Type	Size in Table	Range
Boolean	1 bit	0,1
Byte	1 byte	0..255
Short	2 bytes	-32768..32767
UShort	2 bytes	0..65535
Long	4 bytes	-2147483647..2147483647
ULong	4 bytes	0..4294967295
LLong	8 bytes	$-2^{(64-1)}..+2^{(64-1)}$
ULLong	8 bytes	$0..2^{64}$
Float	4 bytes	$3.4^{-38}..3.4^{38}$
Double	8 bytes	$1.7^{-308}..1.7^{308}$
String	1..65535 bytes	string of fixed length
VarChar	1..65535 bytes	string of variable length
Date	4 bytes	any date between $-2^{22}..2^{22}$ years
Time	4 bytes	any time
DateTime	8 bytes	conjunction of Date and Time
BLOB	up to 2 GB	any binary data
Text	up to 2 GB	text
ObjectPtr	4 bytes	RecID of pointed record

Properties of a Field

Name
Each field must have a name (up to 32 bytes) unique in the scope of BaseObject.

Indexed
If this property is TRUE then Valentina will maintain an index for this field. This property can be changed at runtime.

When Valentina needs to search or sort on a field, it automatically builds the index for that field (if needed) and sets the flag ‘indexed’ to TRUE. If you set this flag to false, then Valentina will to remove the index data for that field from disk.

Indexing requires a more time for some operations (e.g., Add/Update/Delete), but it increases the speed of searching and sorting. Also, indexing requires additional disk space.

Unique
If the flag ‘unique’ is TRUE, then Valentina will not add to the BaseObject a new record with duplicate values for this field.

If the flag Unique is changed at runtime and a table is not empty, then the index will be automatically rebuilt. This occurs because Valentina uses different formats for indexing unique and non unique fields. Unique index has more compact format because it don’t need store for each value count of records (for unique field it is always 1).

Nullable
The flag ‘nullable’ defines if the field accepts NULL values. By default, the flag is FALSE. If it is TRUE, then fields can store NULL values. This feature adds additional information on disk – 1 bit per record of the nullable field.

NULL is not the same as an empty string “” or zero. NULL means – «value is not defined» or «value is not known», so a unique field can have many records with a NULL value. On sorting, records with a NULL value are placed first.

This property can be changed at runtime. If there are records in the table in this case, they will automatically be considered NOT NULL.

IsMethod
This flag is true if the field is a Method. In this case, the Field is virtual and does not use disk space. Its value is calculated on the fly when needed. For real fields of a BaseObject that stores values on disk, this flag is false.

It is not possible convert a virtual field into a real field and vice versa for an existing field.

Numeric Fields

Valentina supports many different numeric types. They differ on the size and on the range of the supported values. You should choose the smallest type of database field which supports your range of values. The benefits are that you reduce the disk space requirements, and increase the overall database performance.

Date, Time and DateTime Fields

Date and Time fields use 4 bytes per record.
DateTime fields store both date and time and use 8 bytes per record.

The internal format of these fields is independent of the system settings. However, the formatting strings for these fields must be consistent for the field in the database.

String and VarChar Fields

Valentina offers 3 field types for storing text:

String	- stores strings of **fixed** length with a range of 1-65,535 bytes.
VarChar	- stores strings of **variable** length with a range of 1-65,535 bytes.
Text	- stores strings of **unlimited** length (up to 2GB).

• For strings with a length of 1 byte, it is better to use a Byte field.
• For strings with length 2..10 bytes, it is better to use a String field.
• For strings with length 10..1024 bytes, it is better to use VarChar or String fields.
• For strings with a length of more than 1K bytes, it is typically more effective to use a Text field. However, if you know that the maximum length is 3 KB and the average length is 100-200 bytes, it might be better to use a VarChar type.

String field – implements fixed length strings. It always use N bytes on disk, even if you store an empty string.

VarChar technical notes

Since a VarChar field can store strings of variable length, the internal implementation of a VarChar file is based on pages.

The minimal size of a page is 1,024 bytes. A page must be able to contain at least 2 strings of maximal length. If you set the MaxLength of a field to 20K then pages with a size of about 40KB will be used.

This is important to know because a page is an atom of read/write operations. Performance will be decreased if you specify unreasonably large MaxLength while storing small strings.

Each page has a header of 8 bytes. For each string on a page are used:
(length of the string + 4) bytes.

Note that if the value of a VarChar is an empty then 8 bytes are used:
4 in the Table and 4 in a page file, i.e. VarChar field has 8 bytes overhead per string

This means that if you use a MaxLength in the range:

1.. 504	size of page is 1,024 bytes
505..1016	size of page is 2,048 bytes

The default MaxLength for a VarChar field is 504 bytes. This is the maximum number that allows us to work with 1,024 byte pages. Indeed:
8 bytes of header + 2 records * (504 + 4) = 1024 bytes.

You can specify lower values for MaxLength (e.g., 20), but 1,024 byte pages will still be used. The only advantage is that you will truncate strings longer than 20 bytes.

Question: Is a VarChar field slower than a String field or faster?

1) Random access speed of a VarChar is lower of the access speed of a fixed length String, because Valentina must at first locate the page of a string, then locate a string on the page.

2) Since the size of a VarChar can be several times less than the size of a String with the same maximal length, operations such as Indexing, Export, and RegEx search should be faster for a VarChar.

3) Update of an existing record for VarChar is slower. This is caused by having to rearrange the whole page.

Properties of String and VarChar Fields

VarChar and String fields have absolutely the same API, they differ just on internal storage format. These types have additional properties that help to manage working with strings.

MaxLength

This property is used for String and VarChar fields (Text field has no limit).
It defines a maximum string length that can be stored in this field. If you try to store a string that is longer, then it will be truncated. When using a VarChar field, there is no benefit (in terms of speed or disk space) to using a value of less than 504 bytes because it stores characters in logical pages.

• Changing of MaxLength is similar to ChangeType operation, because it requires a transformation of disk files.

IndexByWords

When this flag is TRUE, the index is built for each word of the string. This is very useful for large strings (50 characters and more).

For example, let’s say we have one record with the value «aaa bbb ccc ddd». If the field is indexed by words, then search conditions like «bb», «ccc», or «d» will return that record. If the field is not indexed by words, then said conditions return nothing. Only conditions like «a», «aa», «aaa bb» (i.e., which match to the beginning of the string) will return the record.

• If a field is indexed by words, then it can not be sorted, because there is no index on the beginning of the string field.

• On creation of a String and a VarChar fields Valentina set this flag in TRUE if you specify the length more than 255 bytes. You can set this property in FALSE right after creation of a field if you don’t want index the field by words.

• Changing of the IndexByWords property will require rebuilding the index.

Language

The string field has a «Language» property that can accept short integer values. For the MacOS, this is the ID of the corresponding «itl2» resource of the system file.

The default «Language» value is –1, which corresponds to a byte-to-byte method of string comparison which sorts based on the ASCII value of the characters. This is the fastest method, but it can be used only for some single-byte languages (e.g., English). If this method is not acceptable for your language, then you must specify the language code for it. Valentina allows for fields with different languages in the same table.

If you use a single-byte language, then a String can store up to 65,535 characters. If you use a 2-byte code language, then it can accept up to 32,787 characters. The 'length' property of the string field determines the size of the field in bytes (not in characters). Note that if you are going to use your database internationally, languages like Chinese, Korean and Japanese require 2-byte language support.

• Changing of the language property will require rebuilding the index.

RegEx search

String, VarChar and Text fields can be searched using a regular expression (RegEx) search. The syntax of RegEx can be found in the folder «Syntax of RegEx». To search by RegEx, you should use the «LIKE» keyword in the SQL query.

A RegEx search never uses the index of a field because it needs to check all records. Search time by RegEx is proportional to the number of records in a Table.

RegEx search can be case insensitive. For this you should use the switch
(?i) to set case insensitive search ON for the following string
(?-i) to set case insensitive search OFF for the following string

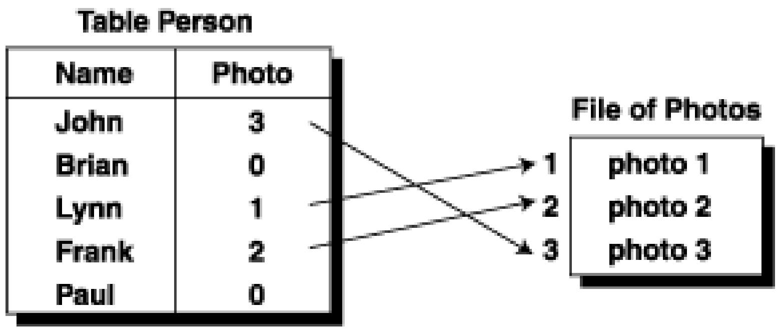
Example: '(?i)I AM CASE INSENSITIVE(?-I)I am Case Sensitive'

Note, that if you need to use in your search characters which are reserved for RegEx, then you should prepend a backslash '\'. For example:

'Company \(A\)'

BLOB Field

This type of field is intended for storing large data objects (e.g., pictures, text, movies). When a BLOB field is defined in a table, 4 bytes in each record are reserved and an additional file is created to store the data. A reference (4 bytes) to the BLOB file is stored in the table record. If a table record does not have an associated BLOB record, then the reference is 0.



When Valentina loads a record of the table into memory, it does not load the associated BLOB record. This allows Valentina to quickly loop through records in a table without the overhead of loading the BLOB data. The data can be loaded into memory using one of the special BLOB field methods.

• With the exception of a Text field, a BLOB field cannot be indexed, unique or nullable.

Properties of BLOB Field

SegmentSize

This parameter is used only at creation time – it cannot be changed at runtime. Choose a value other than the default if you know something about the size of the BLOB data that would allow it to fit within a segment.

For example, if you have a fixed size of 12 KB for an image, then you must set SegmentSize = 12 KB + (overhead of 1 KB). Or if you know that the maximum size of your text field which you will store as a BLOB is 3 KB, then set SegmentSize = 3 KB + 1 KB. By doing this, you can save 10 - 4 = 6 KB on each record.

• If you want to store 100KB BLOBs, then the segment size is not required to be 100 K. It can be 10KB, 30 KB, or even 200KB.
• Do not confuse the SegmentSize of a BLOB field with the SegmentSize of a Database. The SegmentSize of a Database affects allocation of disk space for database files. The SegmentSize of a BLOB field defines the logical structure of a BLOB file.
• Different BLOB fields can have different SegmentSizes.
• The default SegmentSize is 1KB.
• Minimum SegmentSize is 128 bytes.

Text Field

Don’t confuse this field with the String field! This is a special version of a BLOB field that is intended to store large amounts of text. The main advantage of this field is that it can be indexed and searched by index. This field can also be searched by regular expression.

The Text field is very similar to the String field, the only difference is in the internal implementation of the storage mechanism. A String field always has a limit on its length, which must be specified for each field. A Text field has no such limit. On the other hand, a Text field is always slower than a String field because its content is stored outside of the table.

Properties of Text Field

Language
The Text field supports specification of a language, which must be used to index the text.

IndexByWords
When this flag is TRUE, the index is built for each word of the string.

RegEx search

String, VarChar and Text fields can be searched using a regular expression (RegEx) search. The syntax of RegEx can be found in the folder «Syntax of RegEx». To search by RegEx, you should use the «LIKE» keyword in the SQL query.

Picture Field

The Picture field is a special BLOB field, which can store pictures in different formats. By default it stores Pictures with JPEG compression. You can choose a rate of compression by parameter quality with range 0 to 100.

Advanced information:

1) This field must get and returns back:
- on MacOS PICT handle.
- on Windows DIB handle.

2) Picture field is the regular BLOB field which stores data in the next format:

Parameter	**Length**	**Offset**	
PicType	4 bytes	0 bytes	// defined in FBL.h file
PicSize	4 bytes	4 bytes	
Picture itself.		8 bytes	

Picture is stored in the format of QuickTime PICT with JPG compression i.e., it has the following header:

```
struct Picture {
	short		picSize;
	Rect		picFrame;
};

struct Rect {
	short		top;
	short		left;
	short		bottom;
	short		right;
};
```

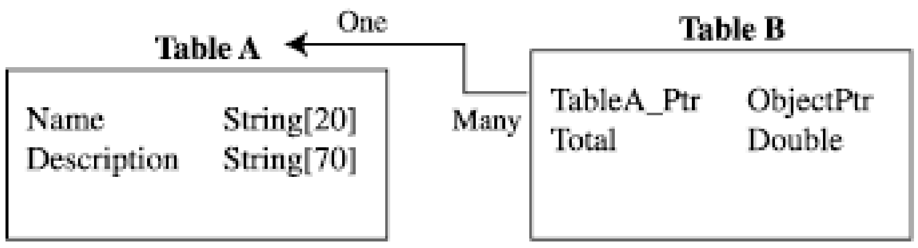
If you have a JPG file created, for example, by PhotoShop, you can store this file directly into Pictrure field. For this you need create and fill in RAM header of picture record, which contains 4 + 4 + (2 + 8) = 18 bytes. Write this header into Picture field and then append context of JPG file (this context must start from 0xFFD8 marker).

ObjectPtr Field

The field of type **ObjectPtr** is intended to establish a relation, «many to one» [M:1] between two tables (BaseObjects).

It stores references to the related parent record (one record). This reference is an unsigned number (4 bytes, ulong) and it is a physical number of the parent record. In other words, ObjectPtr field of Table B stores values of internal virtual field «RecID» of Table A (see picture).

Valentina can use physical number of a record because records are never moved in the table, so it is constant for a record all its life (after deletion of a record its physical number will be reused).



«Parent» Table,
«One» Table

«Child» Table,
«Many» Table

Properties of ObjectPtr

target BaseObject
The ObjectPtr field must know the referenced object (parent object).

deletion control
This parameter maintains the database integrity. It regulates a record deletion in the Many table when a record is deleted in the One table. This parameter can be changed runtime. This is just a RULE, which defines behavior of the deletion of record. There are three ways deletions are handled:

SET_NULL – from the database only the One record is deleted, the ObjectPtr of the related many records is automatically set to NULL. In other words we delete a parent record and leave its child records.

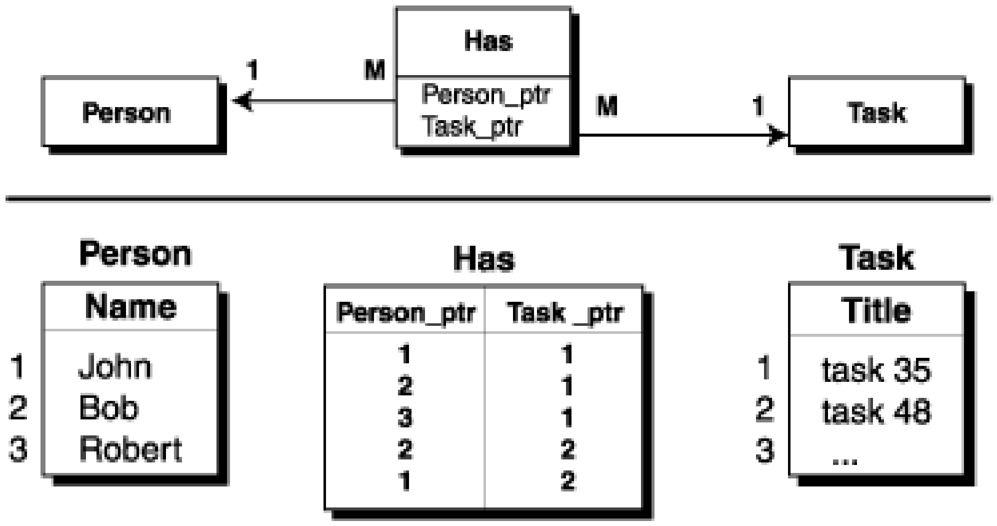
CASCADE – the One record is deleted and all related with it Many records are deleted also. If a Many record also have related Many records in the third Table(s) then they are also deleted. In other words we have cascade deletion.

RESTRICT – the deletion of the One record is not allowed if there is at least one related with it Many record.

The ObjectPtr field can be used to establish MANY to ONE relation.

It also can be used to establish ONE to ONE relation by specifying the ObjectPtr field as unique. Valentina can use this information to optimize query resolving.

Besides using ObjectPtr field, you can establish MANY to MANY relation between 2 tables. For this you need create additional third table - Link as shown on the picture:



BaseObject Methods

Methods of BaseObject are virtual Fields that calculate their own value using values of other fields (real or virtual) of the same record of BaseObject.

This is similar to Calculated fields in FileMaker. In SQL, in a SELECT statement you can specify expressions to calculate the value of a new column, for example:

```
SELECT Name, Upper(Name) FROM Person
```

Methods of BaseObject don't use the disk space to store their values. Instead, they are calculated on only when needed. This is why they are called "virtual Fields". A Method is represented as a modified Field of a BaseObject, so you can use Methods everywhere you can use Fields and in ALMOST the same way. ALMOST because there are some limitations for Methods. In particular you cannot assign a value to a Method, so you can consider them to be a read-only fields.

• It is similar to any computer language that contains VARIABLES and FUNCTIONS. You can read the value of a variable and assign a new value to it:

```
k = var
var = k
```

but you can only read the value of a function:

```
k = sin(x)
sin(x) = 0.12456        // ERROR
```

In Object-Oriented languages, DATA MEMBERS represent what a class KNOWS, and METHODS represent what a class CAN DO. You should think of BaseObject Methods as a Function/Method/Code attached to a BaseObject and which can be called to calculate a value.

Since Methods and Fields are elements of a BaseObject, they share the same name space. You cannot have Fields and Methods with the same name in the same BaseObject.

Method Results

Method result can be any supported Field type (except BLOB types). When you specify the result type of Method, you force Valentina do type casting.

Example: if you specify:

«sin(fld)» returns Short

then you will get only the integer values -1, 0, 1.

If the result size of a calculation is bigger than the specified result type, the result will be truncated.

Indexing of Methods

Methods can have the flags ‘Indexed’ , ‘Unique’ and ‘IndexByWords’. This allows you to create several different indexes for one real field, although it will look like one Field has one Index.

For example you can have the field «Name». If you set an index for this field then the index will be case sensitive (i.e., it will consider Jon and JON as different names). Very often you may want to have a case-insensitive index for this field. To resolve this task, you can define an indexed Method with the name, for example, ‘NameUpr’ and specify its text as: «Upper(Name)».
As a result, an index will be built which gets data from the real field «Name» converted to uppercase. So you can do a case-insensitive search by this index while retaining the original data as case sensitive.

Note that a Method can use several fields of a BaseObject for calculation of a new value. This means that Methods allow you to have COMPOUND indexes also (i.e., an index composed of more than one field).

You can use Methods to create optimized indexes. For example, if you have a String[50] but want search only by the first 4 letters, then you can make an indexed Method: «LEFT(fld,4)».

- An Indexed Method represents 100% of functionality of this feature of standard SQL or the **SET INDEX** feature of FoxPro**.** At the same time, a Method has more power because we can use it to read the values that comprise the index.

NULL Values and Methods

If the value of a real Field is NULL, then the result of the Method is also NULL. The result will also be NULL if a calculation meets some prohibited case, such as division by zero.

Methods can have the flag ‘Nullable’ like real fields. In this case, information about if a value is NULL will be stored on disk - taking 1 bit per record. On the other hand you get the ability to do a very fast search of NULL or NOT NULL values without recalculating all of the records.

This is another advantage of Methods over the RDBMS feature SET INDEX.

Access of BaseObject Methods

Since a Method looks like a COLUMN of a Table when we display it, logically we work with them like with BaseObject Fields. So real Fields and virtual Fields are kept in one common array. The order of Fields and Methods in this array is the order of creation. You can specify any order using a Cursor.

BaseObject.GetFieldCount(kAll | kFields | kMethods)
Returns the count of all Fields in the BaseObject (kAll), the count of real Fields only (kFields), or the count of Methods only (kMethods).

BaseObject.GetFieldByIndex(inIndex)
Returns a real or virtual Field by its index. You can recognize if a field is real or virtual by using the Field.IsMethod property.

BaseObject.GetFieldByName(inName)
Returns a real or virtual Field by its name. You can recognize if a field is real or virtual by using the Field.IsMethod property.

Cursor

Cursor provides the result of an execution of SQL's SELECT statement for a database. Valentina offers a cursor with random access to records (i.e., you can go directly to Nth record of a Cursor).

Cursor is a very powerful mechanism of querying your database. In the background Valentina uses the smart query optimizer and query resolver. You can do queries to search on several fields from different tables and sort result on several fields.

Cursor itself can be considered as a new temporary Table that keeps records of a result. It looks like a Table (it has fields and records) but it also looks like a selection of all records in that Table, because you can go to Nth record of cursor.

Note that navigation methods of BaseObject don't have random access to Nth record, they have only next/prev methods. Since a Cursor keeps a selection of records it uses RAM – 4 bytes per record.

Cursors are good choices for implementing multi-user application. Each Cursor can be executed in separate thread of your application and represent one user.

Each cursor has independent memory buffer, so you can have many cursors at the same time for the same BaseObject, which point on different records even in the same thread.

Properties of Cursor

FieldCount
How many columns this cursor has.

RecordCount
The number of records found as a result of SQL query.

CurrentPosition
Keep current position in the record. Position in the Cursor is not the same as CurrentRecord in BaseObject. For example first record of the Cursor can be 125th in the BaseObject.

ReadOnly
TRUE if the records of the Cursor can be read only; otherwise it is false.

• Cursor is read only if it is designed as a result of join of several BaseObjects with at least one MANY BaseObject.
• Cursor built on single BaseObject can be modified.
• Cursor built as a result of join of only ONE BaseObject can be modified.

In other words Cursor is read only if there is an ambiguity in the execution of such operations as Add/Update/Delete.

Elements of Cursor

Fields
Cursor contains the fields (in order) that you have specified in the SELECT statement of an SQL query.

Records
Cursor can contain zero or more records found as a result of an SQL query.

Access of BaseObject Methods from Cursor

Inside of a Cursor, BaseObject Methods look and work like real fields. You can use the same functions to access Methods:

```
Cursor.GetFieldCount()
Cursor.GetFieldByIndex()
Cursor.GetFieldByName()
```

Relations between Tables

To establish a relationship between two tables, Valentina offers a special mechanism - fields of type **ObjectPtr**. We will call this type of relation an **ObjectPtr-link**, while the old RDBMS way is called a **RDB-link.**

A developer can choose either the traditional RDBMS or modern Valentina technique to establish a relation between tables; furthermore, he can mix both techniques to get the best solution.

To understand how an ObjectPtr works, let's quickly consider traditional techniques of the **relational** and the **network data models**.

Relational Data Model

In the Relational Data Model, in order to establish a relation between two tables, a relational database uses a "Pointer by Value»:

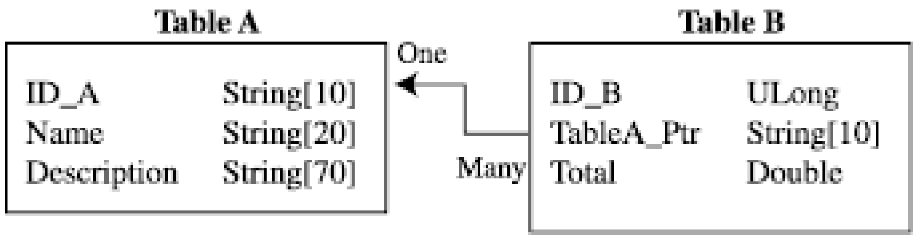


Table A must have a field «ID» (indexed, unique, required), and Table B must have a field with the same attributes - a pointer to the record in Table A. The related records have the same values in the fields «Table A.ID» and «Table B.Table_A_Ptr». As a result, we have a relation [Table A : Table B] of one to many [1 : M] - one record of Table A is related to many records of Table B.

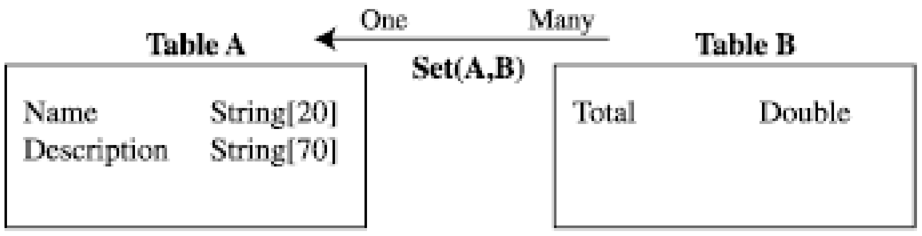
Drawbacks of the Relational Data Model
A database developer must provide values for the unique identifier of Table A. There are 2 traditional ways to do this: 1) as an identifier which is used in some real world value (such as «social security number»), or 2) a special, artificially generated number.

A field of any type can be used as the identifier of the record. So if a string[20] is used, then Table B must store an additional 20 bytes in the record and an additional 20 bytes must be stored in the index of the field «Table_A_Ptr».

Access speed from the record of Table B to the related record of Table A is slow, because the database must perform 4 steps:
1. Get the ID value from the field «Table_A_Ptr».
2. Search the index of the field «Table A.ID», using this value to get some internal ID of the record.
3. Search the primary index of Table A, using the internal ID to get the address of the record.
4. Load the record of Table A.

Network Data Model

The other, much less commonly used database technology is the pointer-based navigational Network Data Model. The same database structure for the network model looks like:



As we can see, the Network Data Model does not need ID fields in the tables, so there is less work for the developer. Instead there is an additional concept, the «Set». The system automatically links related records in the list. A record of Table A that is a parent record, has pointers to the first and the last related records of Table B (4 + 4 = 8 bytes). Each related record of Table B has pointers to the previous and the next related records and a pointer to the parent record (4 + 4 + 4 = 12 bytes).

The direct pointer is a direct address of the record. As a result, the system can find the parent record in the shortest time.

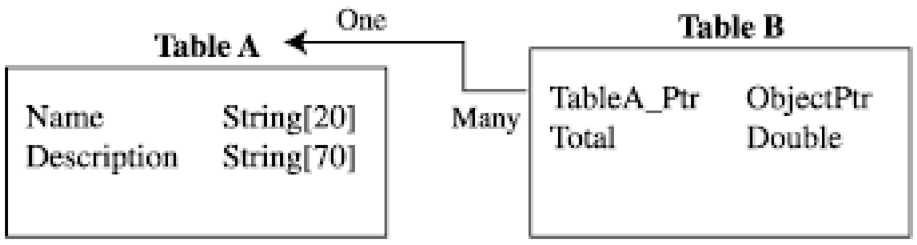
Drawbacks:
The internal representation of the record depends on the database structure. If you add a new set, then the system must add additional space where the pointers will be stored to each record. Sets do not allow for fast random or sorted access to the records, because there are no indexes.

Although at first look the network model is simpler than relational, this is not true. Working with sets adds a significant amount of work to the management of database.

The main problem of Network DBMS - they can't be interactively queried via language like SQL. Queries must be programmed with such languages as C/C++.

Valentina Data Model

Valentina offers a hybrid of the relational and network database models. It stores direct pointers to the related records as normal fields of the table:



With the Valentina Data Model, there no need for key in the tables. There is also a special type of field called an **ObjectPtr**. This field is intended for storing RecIDs of the related records (ulong, 4 bytes).

- A pointer of type ObjectPtr points to the Table and not to the field of the Table as in the relational data model. In other words an ObjectPtr field does not depend on the structure of the related Table. As a result we get a more flexible database structure, because you can change a parent table and the change will not affect other tables.

With the Valentina Data Model, you get all of the benefits of a relational database, plus:
- The database structure is simpler and more flexible.
- Required disk space is lower because there are no ID fields in the tables and there are no corresponding indexes, the size of the pointer is always 4 bytes.
- The parent record can be found in a single logical disk access.

An ObjectPtr field is 2-5 times faster than a RDB-link based on a Long field (i.e., 4 bytes). The ObjectPtr field will be even faster in comparison to a RDB-link based on a field type larger than 4 bytes (for example, in the RDBMS world we very often see tables related by a String[10-20]).

As was mentioned at the beginning of this section, you can mix both techniques in his database. Why would you prefer to use the old slower RDB-links? There are several possibilities:

a) You already have a RDBMS database and want to transfer it to Valentina format. The easist way to do this – use RDB-links. Later you can convert a RDB-link into an Object-Ptr link for an existing database.

b) If data for the database comes from an external source, then typically a special unique ID will mark the records. In this case again it may be easier to use RDB-links.

- You can export records of tables related by ObjectPtr-links into text files which can be imported into another RDBMS. For this export the field «RecID» of the related BaseObject, it will play the role of an unique ID field (i.e. key field) of that table.

The Database Cache

To improve the efficiency of disk access, Valentina uses a Database Cache. The minimum cache size is 512 KB. Maximum size is limited only by available RAM. The optimal choice for cache size is 50% of the available RAM, but it must not be any larger than that.

All information read from disk goes through the Cache, and all information, which you write to the Database, goes through the Cache. If you add/update many records, the cache will be contain new information not yet stored on disk. The risk is that if at this moment a crash occurs, all of the changes will be lost.

To force Valentina to store any new Cache info on disk, you can use the method Flush(). This method will cause all changes to be written to disk. A subsequent crash will not lose data. As a developer of your database application, you must choose a compromise between safety and efficiency:

1) You can call Flush() after any database change. This is the safest way, but efficiency of Add/Update will significantly decrease. This way usually is used if you get one record for Add/Update via the GUI of your application.

2) You can call Flush() after some amount of added/updated records (e.g., 100 or 1000 records). This way gives you the best possible performance but increases the risk of losing data. Usually this method is used by applications which generate new records in a loop by some algorithm.

• You can call Flush() on idle in your application if this is possible.

Valentina automatically flushes the cache in the following cases:
- The cache becomes full. Valentina will write to disk the oldest information of the Cache and remove it from Cache. Imagine that you have a 3 MB Cache and write, in a loop, 100 MB of records. In this case, after the loop the first 97 MB will be already stored on disk and the last 3 MB are still present in the Cache. To save them on disk, call Flush().
- When the database is closed.

RAM requirements for sorting

Valentina’s ‘sort on fields’ algorithm must have enough RAM in order to be successful. In particular it must have 2 * (4 * N) bytes, where N is the number of records in the Selection.

Number of records	**RAM**
10,000	80 KB
100,000	800 KB
1,000,000	8 MB

Note that this is a single operation, which must have enough RAM to be successful. Otherwise, you’ll get the message: «Not enough RAM to complete operation».

All other operations of Valentina can be successful in any amount of RAM. However, they may be slower if the amount of available RAM is low.

Appendix A: Valentina error numbers

The errors numbers of MacOS and Windows - are negative numbers and 1..100
The errors specific to Valentina - are positive numbers in range 300..700

The following is the list of all error numbers which can be raised by Valentina:

kFBL_TimeOut	= 666	
kFBL_Error	= 300	
kFBL_DataFileError	= 301	
kFBL_TableError	= 302	
kFBL_FieldError	= 303	
kFBL_IndexError	= 304	
kFBL_ParserError	= 305	
kFBL_CacheIsNotPresent	= 306	
kFBL_AssertionFail	= 307	
kFBL_AssertionNULL	= 308	
kFBL_CacheIsSmall	= 309	
kFBL_WrongParameterType	= 310	
		// DataFile errors:
kFBL_CantOpenNewVersion	= 320	
kFBL_CantReadDomain	= 321	
kFBL_EmbFileNotFound	= 322	
kFBL_ExpectedBONotFound	= 323	
kFBL_ExpectedFieldNotFound	= 324	
kFBL_WrongPassword	= 325	
		// Field errors.
kFBL_FieldNotIndexed	= 340	
kFBL_FieldIsConstant	= 341	
kFBL_FieldIsCounted	= 342	
kFBL_FieldIsComposed	= 343	
kFBL_FieldIsUnique	= 344	
kFBL_IndexNotSorted	= 345	
kFBL_FieldNameNotUnique	= 346	
kFBL_FieldIsNotMethod	= 347	
kFBL_InvPageVarChar	= 348	
kFBL_CannotCreate	= 349	
		// Table errors.
kFBL_TableIsEmpty	= 360	
kFBL_TableIsCorrupted	= 361	
kFBL_RecordNotFound	= 362	
		// Index errors.
kFBL_SXCorrupted	= 370	
kFBL_FailToBuild	= 371	
kFBL_InvPage	= 372	
		// Query parser errors.
kFBL_NeedValue	= 380	
kFBL_NeedLeftBound	= 381	
kFBL_NeedRightBound	= 382	

```
                                                // Parameter errors.
kFBL_WrongDataBaseRef             = 390
kFBL_WrongBaseObjectRef           = 391
kFBL_WrongFieldRef                = 392
kFBL_WrongCursorRef               = 393
                                                // OBL layear errors:
kOBL_Error                        = 500
kOBL_BaseObjectError              = 501
kOBL_RelationError                = 502
kOBL_DatabaseError                = 503
                                                // Database errors:
kOBL_WrongDBName                  = 510
                                                // BaseObject errors.
kOBL_WrongBaseNumber              = 520
kOBL_WrongChildType               = 521
kOBL_ObjectIsNotEmpty             = 522
kOBL_CorruptedChain               = 523
kOBL_ExpectedBONotFound           = 524
kOBL_ExpectedFieldNotFound        = 525
                                                // Relation errors.
kOBL_CantRelate                   = 550
kOBL_RelationIsNotEmpty           = 551

                                                // SQL errors:
errSQL_SELECTexpected             = 600
errSQL_FROMexpected               = 601
errSQL_TableNotFound              = 602
errSQL_FieldNotFound              = 603

errSQL_SyntaxError                = 604
errSQL_CantBeUpdated              = 605
errSQL_NullExpected               = 606
errSQL_LeftBracketExpected        = 607
errSQL_RightBracketExpected       = 608
errSQL_ExpressionExpected         = 609
errSQL_WrongFieldType             = 610
errSQL_NotStringField             = 611
errSQL_ValueExpected              = 612

errSQL_WrongLink                  = 613
errSQL_AmbigiouseLink             = 614
errSQL_MissingLink                = 615
errSQL_WrongExpression            = 616

errSQL_MixAgregativeAndNormal= 617
errSQL_CommaExpected              = 618
```