VALENTINA Database Kernel

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:

	PPC	**X**	**Win32**
Valentina ORDBMS with GUI and AppleScript support	•	•	
Valentina C++ SDK	•	•	•
Valentina C SDK	•	•	•
Valentina Java SDK	•	•	•
Valentina for REALbasic.	•	•	•
Valentina for Macromedia Director	•	•	•
Valentina XCMD	•	•	•
Valentina for WebSiphon	•		
Valentina COM			•

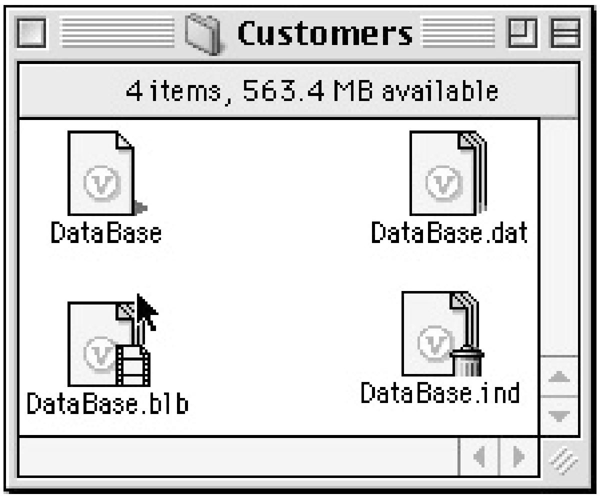
The Valentina database file format is cross-platform and consistent between all Valentina products. In other words, Valentina offers a solution to allow databases built for use with one development environment to be used with any other development environment supported by Valentina on any supported platform.

Parameters of the Valentina Database Engine

Max length of the disk file:	$2^{(64-1)}$ (2 TB)
Max number of Tables in databse:	$2^{(32-1)}$ (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 number of indexes per Table:	unlimited
Max size of String field	64KB
Max size of VarChar field	64KB
Max size of a BLOB record:	$2^{(64-1)}$ (2 TB)

Database of Valentina

Valentina has its own internal file system, so it can store many logical files in a single disk file. Valentina can be configured to store a databases as one, two, three, or four separate files:
«dbName.vdb» - description of the database structure
«dbName.dat» - custom data stored in records
«dbName.blb» - contains BLOB (Binary Large OBjects), TEXT and Picture fields
«dbName.ind» - 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”**[desc LOB, 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**
5 - **“.vdb”**[description,data,BLOB] + “**.ind”**[indexes]
6 - **“.vdb”**[description,data] + **“.blb”**[BLOB] + “**.ind”**[indexes]
7 - **“.vdb”**[description,data,indexes] + **“.blb”**[BLOB]
8 - **“.vdb”**[description] + **“.dat”**[data,indexes] + **“.blb”**[BLOB]

«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, 4, 8), 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 custom data stored in the database records.

«dbName.blb» file

Contains the contents of BLOB, TEXT and Picture fields.

«dbName.ind» file

Contains indexes, joins and any other temporary files. Some of these files will persist between sessions, while others will be removed once a session is finished.

Advantages to having a separate index file are the following:
-- Reduces the risk of disk write data corruption because part of the disk write will be to a separate index file, which can be rebuilt by Valentina. 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. Wrong indexes can result from interruption of programm execution by debugging tools in C++, REALbasic, Director, XCMD, and other languages.

Properties of Database

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

DateTime Format
Valentina uses a proprietary 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. ":"
CenturyBound	- Define behavior of auto-correction of date feature. Default is 20.

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.

Valentina provides auto correction of 1 and 2 digit date years.
Example:

```
0    => 2000
1    => 2001
19   => 2019
20   => 1920        <<<<<<<<< boundary
99   => 1999
```

If you set the century boundary to zero, then Valentina will not provide auto-correction of year values.

TimeOut
Specifies the maximum lifespan in seconds of a Cursor of this DataBase. All Cursors that exceed this time are destroyed. The default TimeOut value is -1 (infinite lifespan).

SegmentSize
Specifies the segment size for database files. Valentina allocates minimum a segment, when it is needed a new space on disk.

Mode
Specifies the number of database files.

Encrypted
Returns TRUE if the entry database is encrypted.

Elements of Database

BaseObject
In order to be valid, each database must contain at least one valid BaseObject, Valentina's unique definition of a traditional table. You can obtain the count of BaseObjects in a database, as well as individual BaseObjects by name or by index (the index is 1-based).

Cursor
The result of an SQL (structured query language) command is a Cursor. The database automatically keeps track of all Cursors. A Cursor will be destroyed once it exceeds the TimeOut value set by the user for 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. Read the description of this parameter at the beginning of this chapter. Once set during the creation of a database, the Mode parameter cannot be changed.

Segment Size
Specifies the minimum size by which Valentina will grow the disk file. 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.

The default value is 32 KB.

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

Encryption of database

You can optionally encrypt your Valentina database using an Encryption Key.

Encryption gives much stronger protection of database files then just protecting files by password. When you encrypt a database, its database files are altered to a format that is not readable. The encryption key is not stored inside the database at all, so your database cannot be hacked.

• ATTENTION: if you forget your Encryption Key of encryption you will loose this database. There is no way to decrypt data without Key.

As a trade-off for strong protection the database is a little slower, because each time Valentina reads/writes data from/to disk it must execute decryption/encryption. In fact, this slows the database by only several percentage points.

To allow you to select the best balance of performance vs. security for your needs, Valentina offers you several flexiable options of encryption:

1) encryption of the whole database
   This option provides you the most protection because ALL your data are encrypted.

2) encryption of one or several BaseObjects only.
   This option allows you to encrypt only table(s) with confidentional information and still retain excellent performance for other tables.

3) encryption of one or several fields only.
   This option allows you to encrypt only field(s) with confidentional information and still retain excellent performance for other fields.

4) encryption of database description file (".vdb") only.
   This option does not encrypt your data at all. It protects only the database structure description, therefore preventing your database from being opened by any other application using a Valentina kernel.

• NOTE • When a field is encrypted, its index file is also encrypted.

\By default Valentina uses the Blowfish encryption algorithm. BlowFish is both strong (448 bits) and fast. Its Encryption Key is a string that can be from 1 to 72 characters long, with a longer key providing stronger security.

Valentina allows you to Set/Remove encryption and change the Encryption Key at runtime even if the table being addressed contains records. Please note that runtime encryption may take a bit of time, since Valentina must modify each table record.

System Tables

YOU CAN SKIP THIS PARAGRAPH if you do not intend to use Valentina's more advanced features.

Valentina stores information about database structure in the "DatabaseName.vdb" file using system tables.

Usually RDBMS has the following system tables:

```
sysTable    { name, ...                }
sysField    { name, type, length, ... }
sysType     { value, value, ...        }     // info about enum types.
sysUser     { name, password, ...      }
sysGroup    { ID, UserPtr, ...         }
.....
```

Valentina uses only 2 system tables: "**sysItemDescription**" and "**sysItem**". Both tables have self recursion, as described bellow. In addition, the "sysItem" table has a pointer on the table "sysItemDescription".

Table "sysItemDescription" contains descriptive information about system objects and their properties. Its structure is:

```
sysItemDescription{
    Kind        as Long (indexed + unique)
    Name        as VarChar[504] (indexed)
    Type        as Byte
    ParentPtr   as ObjectPtr to sysItemDescription (RESTRICT)     // self recursion.
}
```

Valentina uses the **Name** and **Type** fields to build temporary pseudo-system tables. The **Kind** field is a unique long integer identifier of item description. You can use positive values of Kind. Valentina reserves negative values for its own use.

Note, that column 'Name' is not marked as unique. This is because different system tables must be able to have columns with the identical names. For example, study the default Valentina system table on next few pages. Both object "BaseObject" and object "Field" have property "Name".

The table "sysItem" contains 2 main fields: {**Value**, **TextValue**} and 2 ObjectPtr fields to establish relations. The structure of table "sysItem" is:

```
sysItem{
    ItemDescrPtr    as ObjectPtr to sysItemDescription (indexed, RESTRICT)
    Value           as VarChar[504]
    TextValue       as Text[256]
    ParentPtr       as ObjectPtr to sysItem (indexed, CASCADE)  // self recursion.
};
```

Note that a sysItem can store its value either in the field 'Value' (preferred for numbers and short strings) or in the field 'TextValue' (better for large chunks of text). So, when you assign a new value to a property you must choose where to store that value.

Valentina uses ItemDescPtr to link Item with its description. ParentPtr field is used to create a hierarchy of items: child items point to their parents. In fact, such a structure allows us to build virtual trees in a table. Valentina builds 2 such trees:
1) hierarchy of objects;
2) hierarchy of properties for objects.

The hiearchy of objects is shown by:

```
        database
            |------> BaseObject1
            |               |-----> field1
            |               |-----> field2
            |               |-----> field3
            |------> BaseObject2
                            |-----> field1
                            |-----> field2
```

and the hiearchy of properties is shown by:

```
        BaseObject1
            |-----> name
            |-----> fieldCount
            |-----> linkCount
```

It is interesting to note that:
• records of "Objects" in the "sysItem" table have fields Value and TextValue equal to 0.
• a property record always has at least one populated value field, and it do not have child records.

Advantages of the Valentina approach
These 2 tables can handle standard system tables mentioned above, but it is more flexiable:
1) you can add your own properties for any system object– Database, BaseObject, or Field. These properties will be stored in the vdb file, i.e. you can really adapt the database structure to needs of your application.
2) EVEN MORE: you can create your own system objects and their properties. For example you can create table "Form", "Stored Procedure", "Font maps", "Character Set".
3) older version of Valentina kernel will be able to open db files of newer version. Properties added by newer versions of Valentina are simply ignored.
4) you can build, on the fly, a temporary system table for EACH system object: "sysBaseObject", "sysField", "sysType", so Valentina will operate as usualy RDBMS.

If look deeper, you will see that the table "sysItemDescription" contain the descrption of a system table, and the table "sysItem" contains records of that system tables.

Default records of sysItemDescription table

Let's look at sysItemDescription table. Note that the RecID column is displayed just for convenience, to help you see more easily how records are related.

Note, that Names of objects and properties will never be changed or removed. At the same, time new items can be added, so your application must not depend on the number of records and their order. Your app must always retrieve the RecID of a specific item by searching for the "Kind" item that represents the key for this table (rather than the "Name" item).

```
------------------------------------------------------------------------------------------------------------
KIND        NAME                      TYPE                  PARENT
------------------------------------------------------------------------------------------------------------
// system Objects
{-1,        "Database",               0,                    0        },      // RecID = 1
{-2,        "BaseObject",             0,                    0        },      // RecID = 2
{-3,        "Field",                  0,                    0        },      // RecID = 3
{-4,        "Link",                   0,                    0        },      // RecID = 4
{-5,        "LinkBranch",             0,                    0        },      // RecID = 5
{-6,        "EnumType",               0,                    0        },      // RecID = 6
{-7,        "User",                   0,                    0        },      // RecID = 7
{-8,        "Group",                  0,                    0        },      // RecID = 8
{-9,        "Procedure",              0,                    0        },      // RecID = 9

// Database properties:
{-100,      "DateFormat",             kFBL_TypeString,      1        },      // RecID = 10
{-101,      "DateSep",                kFBL_TypeString,      1        },      // RecID = 11
{-102,      "TimeSep",                kFBL_TypeString,      1        },      // RecID = 12
{-103,      "DateCentury1",           kFBL_TypeLong,        1        },      // RecID = 13
{-104,      "DateCentury2",           kFBL_TypeLong,        1        },      // RecID = 14
{-105,      "DateBound",              kFBL_TypeLong,        1        },      // RecID = 15
{-106,      "Encrypted",              kFBL_TypeBoolean,     1        },      // RecID = 16
{-107,      "StructureEncrypted",     kFBL_TypeBoolean,     1        },      // RecID = 17
{-108,      "TimeOut",                kFBL_TypeULong,       1        },      // RecID = 18
{-109,      "Timer",                  kFBL_TypeLLong,       1        },      // RecID = 19
{-110,      "BaseObjectLastID",       kFBL_TypeLong,        1        },      // RecID = 20
{-111,      "FieldLastID",            kFBL_TypeLong,        1        },      // RecID = 21
{-112,      "EncryptedExample",       kFBL_TypeVarChar,     1        },      // RecID = 22
{-113,      "EncStructExample",       kFBL_TypeVarChar,     1        },      // RecID = 23

// BaseObject properties:
{-200,      "Name",                   kFBL_TypeVarChar,     2        },      // RecID = 24
{-201,      "Encrypted",              kFBL_TypeBoolean,     2        },      // RecID = 25
{-202,      "BaseObjectID",           kFBL_TypeLong,        2        },      // RecID = 26
{-203,      "Parent",                 kFBL_TypeVarChar,     2        },      // RecID = 27
{-204,      "InheritanceType",        kFBL_TypeShort,       2        },      // RecID = 28
{-205,      "Abstract",               kFBL_TypeBoolean,     2        },      // RecID = 29
{-206,      "EncryptedExample",       kFBL_TypeVarChar,     2        },      // RecID = 30
```

```
KIND      NAME                  TYPE                 PARENT

// Field properties:
{-301,    "Name",               kFBL_TypeVarChar,    3      },    // RecID = 31
{-302,    "Encrypted",          kFBL_TypeBoolean,    3      },    // RecID = 32
{-303,    "Type",               kFBL_TypeByte,       3      },    // RecID = 33
{-304,    "FieldID",            kFBL_TypeLong,       3      },    // RecID = 34

{-305,    "Indexed",            kFBL_TypeBoolean,    3      },    // RecID = 35
{-306,    "Unique",             kFBL_TypeBoolean,    3      },    // RecID = 36
{-307,    "Nullable",           kFBL_TypeBoolean,    3      },    // RecID = 37
{-308,    "ByWords",            kFBL_TypeBoolean,    3      },    // RecID = 38
{-309,    "Compressed",         kFBL_TypeBoolean,    3      },    // RecID = 39
{-310,    "Method",             kFBL_TypeBoolean,    3      },    // RecID = 40

{-320,    "MaxLength",          kFBL_TypeULong,      3      },    // RecID = 41
{-321,    "Language",           kFBL_TypeVarChar,    3      },    // RecID = 42
{-322,    "Segment",            kFBL_TypeULong,      3      },    // RecID = 43
{-323,    "Compression",        kFBL_TypeUShort,     3      },    // RecID = 44
{-324,    "Quality",            kFBL_TypeUShort,     3      },    // RecID = 45
{-325,    "MethodText",         kFBL_TypeText,       3      },    // RecID = 46

{-328,    "MinValue",           kFBL_TypeVarChar,    3      },    // RecID = 47
{-329,    "MaxValue",           kFBL_TypeVarChar,    3      },    // RecID = 48
{-330,    "DefaultValue",       kFBL_TypeVarChar,    3      },    // RecID = 49
{-331,    "Constrain",          kFBL_TypeVarChar,    3      },    // RecID = 50
{-332,    "EncryptedExample",   kFBL_TypeVarChar,    3      },    // RecID = 51

// Link properties:
{-400,    "Name",               kFBL_TypeVarChar,    4      },    // RecID = 52
{-401,    "LinkType",           kFBL_TypeByte,       4      },    // RecID = 53
{-402,    "Target",             kFBL_TypeVarChar,    4      },    // RecID = 54
{-403,    "DeletionControl",    kFBL_TypeVarChar,    4      },    // RecID = 55

// LinkBranch properties:
{-500,    "Name",               kFBL_TypeVarChar,    5      },    // RecID = 56
{-501,    "RelType",            kFBL_TypeByte,       5      },    // RecID = 57
{-502,    "DeletionControl",    kFBL_TypeVarChar,    5      },    // RecID = 58

// Enum properties:
{-600,    "EnumItem",           kFBL_TypeVarChar,    6      },    // RecID = 59

// User properties:
{-700,    "Name",               kFBL_TypeVarChar,    7      },    // RecID = 60
{-701,    "Password",           kFBL_TypeVarChar,    7      },    // RecID = 61

// Group properties:
{-800,    "Name",               kFBL_TypeVarChar,    8      },    // RecID = 62

// Procedure properties:
{-900,    "Name",               kFBL_TypeVarChar,    9      },    // RecID = 63
{-901,    "ProcText",           kFBL_TypeText,       9      }     // RecID = 64
```

Example of sysItem table

Now let's look on example of sysItem table. Let we have simple database with 1 table "Person" and 3 fields FirstName, LastName, BirthDate. Note, that first records describe Database object (because it was created first), then we see records that describe table and then fields.

```
-----------------------------------------------------------------------------------
RecID       ItemDescrPtr    Value         TextValue   ParentPtr
-----------------------------------------------------------------------------------
1           {1,             0,            0,          0     },    // Database

2           {10,            "2",          0,          1     },    // DateFormat
3           {11,            "/",          0,          1     },    // DateSep
4           {12,            ":",          0,          1     },    // TimeSep
5           {13,            "1990",       0,          1     },    // DateCentury1
6           {14,            "2000",       0,          1     },    // DateCentury2
7           {15,            "20",         0,          1     },    // DateBound
8           {16,            "0",          0,          1     },    // Encrypted
9           {17,            "0",          0,          1     },    // StructureEncrypted

10          {2,             0,            0,          1     },    // BaseObject
11          {24,            "Person",     0,          10    },    // Name
12          {25,            "0",          0,          10    },    // Encrypted

13          {3,             0,            0,          10    },    // Field
14          {31,            "FirstName",  0,          13    },    // Name
15          {32,            "0",          0,          14    },    // Encrypted
16          {33,            "VarChar",    0,          15    },    // Type
17          {41,            "504",        0,          16    },    // MaxLength
18          {42,            "English",    0,          17    },    // Language

19          {3,             0,            0,          10    },    // Field
20          {31,            "LastName",   0,          19    },    // Name
21          {32,            "0",          0,          19    },    // Encrypted
22          {33,            "VarChar",    0,          19    },    // Type
23          {41,            "504",        0,          19    },    // MaxLength
24          {42,            "English",    0,          19    },    // Language

25          {3,             0,            0,          10    },    // Field
26          {31,            "BirthDate",  0,          25    },    // Name
27          {32,            "0",          0,          25    },    // Encrypted
28          {33,            "Date",       0,          25    },    // Type
```

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.

Functions to modify BaseObject properties

You can use the next functions (direct API) to modify properites:

```
VBaseObject.SetProperty(
	inPropertyName as String,
	inPropertyValue as String )
```

Assign a new value for specified property of VBaseObject. If such property not exists it is created. If you want delete a property then specify NULL for the second parameter.

```
VBaseObject.GetProperty(
	inPropertyName as String ) As String
```

Read and returns the specified property of VBaseObject.

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
Medium	3 bytes	-8388608..8388607
UMedium	3 bytes	0..16777215
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	$\pm (3.4^{-38}..3.4^{38})$
Double	8 bytes	$\pm (1.7^{-308}..1.7^{308})$
Date	4 bytes	any date between -2^{22} ..2^{22} years
Time	4 bytes	any time
DateTime	8 bytes	conjunction of Date and Time
String	1..65535 bytes	string of fixed length
VarChar	1..65535 bytes	string of variable length
FixedBinary	1..65535 bytes	binary data of fixed length
VarBinary	1..65535 bytes	binary data of variable length
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.

Functions to modify Field properties

You can use the next functions (direct API) to modify properites:

```
VField.SetProperty(
        inPropertyName as String,
        inPropertyValue as String )
```

Assign a new value for specified property of VField. If such property not exists it is created. If you want delete a property then specify NULL for the second parameter.

```
VField.GetProperty(
        inPropertyName as String ) As String
```

Read and returns the specified property of VField.

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.

For example if you know that in field will be stored numbers in range 0..1,000,000 the best choice is UMedium type, because UShort type is not enough to handle1,000,000. ULong type also can be used but in this case you loose 1 byte per record.

Note, that Boolean type of Valentina uses really just one BIT per record, other DBMS use 1 byte or, in best case, they pack into byte 1-8 bits.

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 name of language as string. For the list of available languages look below.

The default «Language» is "ASCII", 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. If this method is not acceptable for your language, then you must specify the language. 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 of the field, because order of records most probably will be other.

List of languages:

Default
ASCII

Afrikaans
Arabic
Armenian
Bangladesh
Bulgarian
Byelorussian
Catalan
Croatian
Czech
Danish
Dutch
English
Estonian
Farsi
Finnish
French
French Canadian
German
Greek
Hebrew
Hindi
Hungarian
Icelandic
Indonesian
Italian
Japanese
Korean
Latvian
Lithuanian
Norwegian
Pakistani
Persian
Polish
Portuguese
Romanian
Russian
Serbian
Simplified Chinese
Slovakian
Slovenian
Spanish
Swedish
Thai
Traditional Chinese
Turkish
Ukrainian
Vietnamese

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

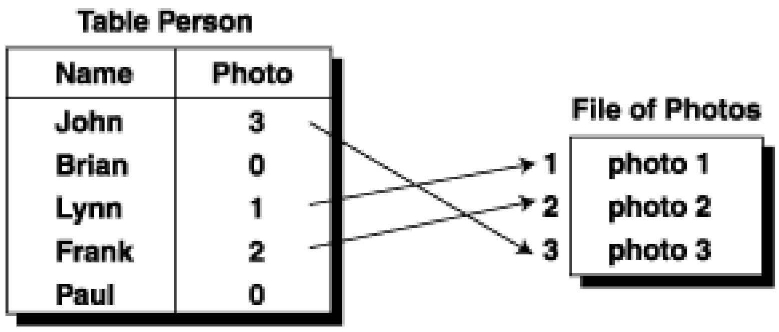
Example: ‘(?i)I AM CASE INSENSITIVE(?-I)I am Case Sensitive’

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.

Compression of BLOB field

Valentina allows you compress a BLOB field. You can make a BLOB field compressed as easy as set its flag 'Compressed' to TRUE.

• On default Valentina uses ZIP compression.
• You can set up level of compression in range 0..9.
  0 - fastest compression.
  9 - best compression.

Valentina allows you change the flag 'Compressed' at runtime, i.e. you can change it for table that has records. As you understand, this is a long time operation, because Valentina must compress/uncompress BLOB field of each table record.

If a BLOB field has index (for example TEXT field) then index is not compressed and still can be successfuly work.

In fact the main advantege to have compression of BLOB field on the database kernel level is ability to have an INDEXED compressed TEXT field. Really, for not indexed general BLOB developer can implement own compression and store to BLOB already compressed data. But this way will not work with TEXT, because Valentina will not be able to built index. When Valentina do compression self it can build index with no problems.

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;
};
```

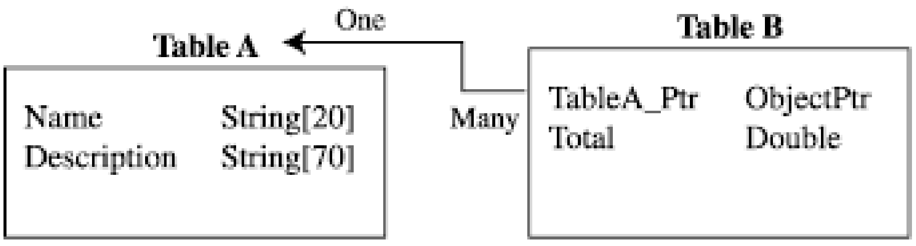
If you have a JPG file created, for example, by PhotoShop, you can store this file directly into Pictrure field. For this you need create and fill in RAM header of picture record, which contains 4 + 4 + (2 + 8) = 18 bytes. Write this header into Picture field and then append context of JPG file (this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

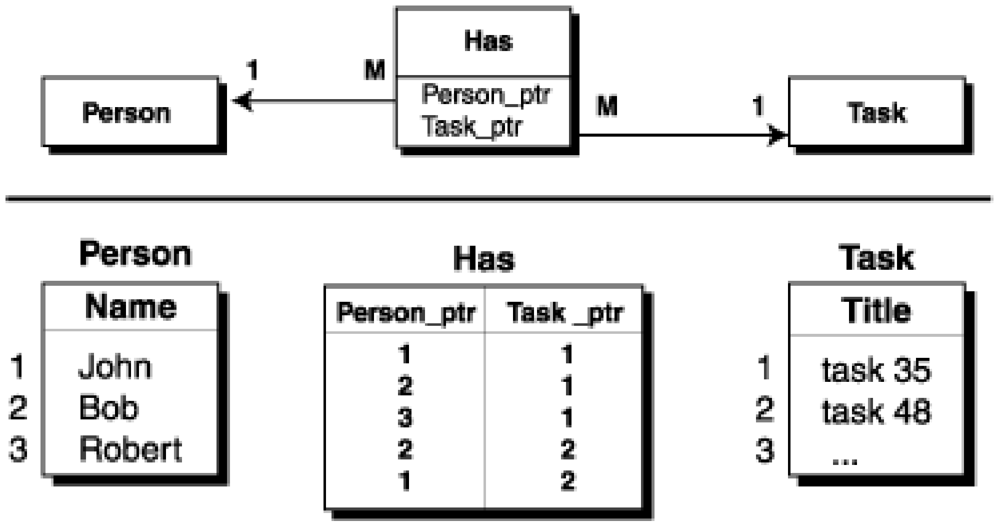
CASCADE – the One record is deleted and all related with it Many records are deleted also. If a Many record also have related Many records in the third Table(s) then they are also deleted. In other words we have cascade deletion.

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:



Person

	Name
1	John
2	Bob
3	Robert

Has

Person_ptr	Task _ptr
1	1
2	1
3	1
2	2
1	2

Task

	Title
1	task 35
2	task 48
3	...

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 of 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.

Using of BaseObject Methods

BaseObject Methods significantly expand flexibility of database kernel. They can be used for the following:

1) Having several indexes for the same field

Let you have field “f1” and you want have 2 or more different indexes for its data. To do this you define INDEXED BaseObject Methods. Method do not store own values on disk, but its index will be stored on disk, so in fact we have one field column and several index files.

You can use this if you need:
-- have index for a field as in original so in upper case.
-- have index by words and in the same time NOT by words to be able sort strings.

2) Compound index

i.e. index which is built using data of SEVERAL fields. To do this you need define INDEXED Method that calculate its value using data of 2 or more fields.

3) Optimization of index

Let you have a string field "F1" with length 150 bytes, but want index only first 7 bytes. To do this you define Method = "left(f1, 7)" and make it indexed. Field "F1" must not be indexed.

4) Case-insensitive work with strings

Very often you need be able search on strings ignoring case. To do this database must have an index for field in UPPER or Lower case. So you define a Method "M1" as = "upper(f1)" and make it indexed. Now in query you use M1 for search and F1 for displaying data in original case.

```
s = "test STRING"
sUpr = toUpper( s )
SQLstring = "SELECT f1 FROM T WHERE m1 = '" + sUpr + "'"
```

Note, that the match string must be converted to upper case before you will use it in SQL query.

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.

Creation of Cursor

When you create a cursor using VDatabase.SqlSelect() method you need specify 3 parameters:

CursorLocation	- can be { kClientSize = 1, kServerSide = 2 }.
LockType	- can be { kNoLock = 1, kReadOnly = 2, kReadWrite = 3 }.
CursorDirection	- can be { kForwardOnly = 1, kRandom = 2 }.

CursorLocation parameter specifies where will be located this cursor. The client side cursor copy all found records on

1) Client side cursor can be read-only. i.e. you cannot modify its records. Client side cursors should be used only for selections with small amount of records. Because Server send all found records into RAM of Client computer, and this can take some time.
This cursors are good for WEB development.
Note, that you can modify selected records using SQL's commands INSERT/UPDATE/DELETE, but your cursor will not reflect this changes. You need rebuild cursor to see that changes.

2) Server side cursor can modify its records if you have specify kReadWrite parameter, and cursor satisfy usual rules of Updatable cursor (e.g. it is built for single table).
Server side cursor keeps selected records on the Server RAM and/or HDD, Only one record at time is moved to the RAM of Client computer. This cursors are good for Intranet development where you have fast network.

IMPORTANT!!!
If you do not specify this parameters, then on default Valentina creates:
Client side, read-only, Forward-only cursor.
This combination of parameters create cursor that is the most friendly to other users in multi-user environment.

Therefore, if you port your existed stand-alone application to be Client, you need to make sure that for ALL cursors which you use to modify records, you have specify kServerSide.

LockType parameter specifies what record locks you want obtain for this cursor.

1) if you do not want lock records and be affected by locks of other cursors then you can specify kNoLock.
NOTE: kNoLock is similar to transaction isolation level 0 "DIRTY READ".
Use this kind of locks very carefully! When you exactly know what you do.

2) If you want only read records using this cursor then you need use kReadOnly.
Several different cursor can set Read Only lock on some record. This is why this kind of lock also is named SHARED LOCK.
NOTE: kReadOnly is similar to isolation level 1 "UNCOMMITTED READ".

3) If you want modify records of this cursor then you need specify kReadWrite.
Only one cursor can set Write lock on a record. This is why such kind of lock also is named as EXCLUSIVE LOCK.
NOTE: kReadWrite is similar to isolation level 3 "REPEATABLE READ".

In the case Valentina cannot obtain lock for at least one record of cursor, it return error kCannotSetLock and cursor is not created.

CursorDirection parameter is optional. If you know that you will iterate the cursor only once in forward direction then you can specify this to Valentina. As result Valentina will be able unlock a record as soon as you move to the next record, because Valentina know that you will never return to that record using this cursor. Again, this is just optimization. This parameter DO NOT change semantic of operations.

Access of BaseObject Methods from Cursor

When you use 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»:

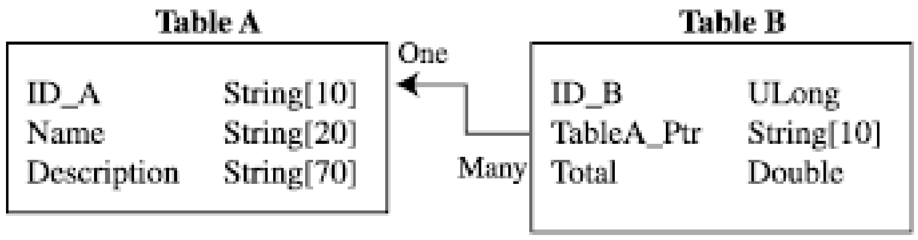


Table A must have a field «ID» (indexed, unique, required), and Table B must have a field with the same attributes - a pointer to the record in Table A. The related records have the same values in the fields «Table A.ID» and «Table B.Table_A_Ptr». As a result, we have a relation [Table A : Table B] of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

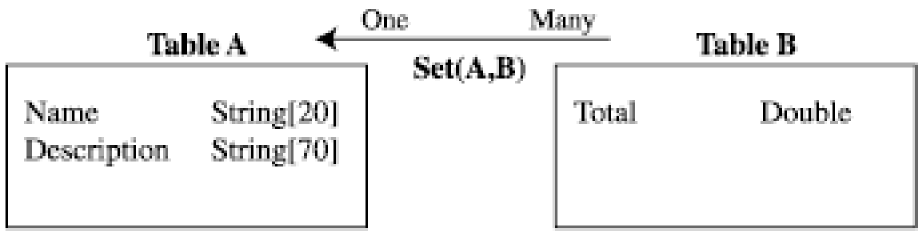
Drawbacks of the Relational Data Model
A database developer must provide values for the unique identifier of Table A. There are 2 traditional ways to do this: 1) as an identifier which is used in some real world value (such as «social security number»), or 2) a special, artificially generated number.

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.

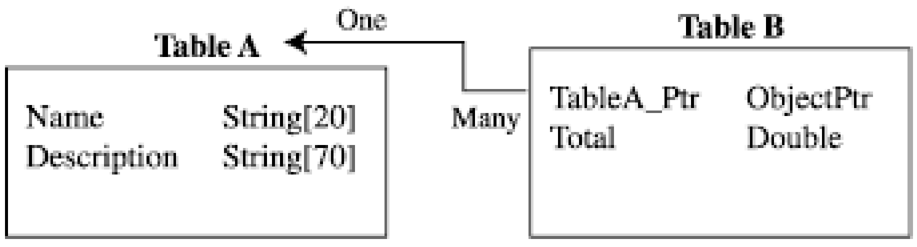
Drawbacks:
The internal representation of the record depends on the database structure. If you add a new set, then the system must add additional space where the pointers will be stored to each record. Sets do not allow for fast random or sorted access to the records, because there are no indexes.

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	
kFBL_WrongEncryptionKey	= 326	
kFBL_DatabaseIsNotOpened	= 327	
kFBL_WrongStructureEncryptionKey	= 328	
		// 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	
kFBL_LanguageNotFound	= 350,	
kFBL_CorruptedVarChar	= 351,	
		// 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 layer 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
errSQL_NotEnoughParameters         = 619;
errSQL_Expected_TABLE              = 620;
errSQL_Expected_WHERE              = 621;
errSQL_Expected_INTO               = 622;
errSQL_Expected_VALUES             = 623;
errSQL_Expected_SET                = 624;
errSQL_Expected_ASSIGN             = 625;
errSQL_Expected_DOT                = 626;

errSQL_Invalid_FieldLength         = 627;
errSQL_Invalid_FieldValue          = 628;
errSQL_Missing_FieldValues         = 629;
errSQL_ExpectedStringConstant      = 630;

// semantic errors:
errSQL_FieldNotFromSelectedTable = 640;
```