

Vitalware Documentation

Release Notes: Vitalware 2.1.01

Document Version 1

Vitalware Version 2.1



Contents

Here you will find collected together the Release Notes for Vitalware 2.1.01, alongside all documents referenced in the notes. These release notes and documents are also available on the [KE Vitalware website](#).

This PDF document brings together a number of individually published documents: please note that page numbering below refers to this combined PDF document and not to the page numbers printed at the bottom of pages, as each individual document, e.g. Record Recall, has its own internal numbering:

Release Notes: Vitalware 2.1.01	5
Statistics documentation	24
Record Recall	63
Record Templates	73
XSLT processing of XML import files	105
FIFO Server	118
KE Vitalware Configuration	137
Range Indexing	169

Release Notes: Vitalware 2.1.01

Release Date: 24 July 2009

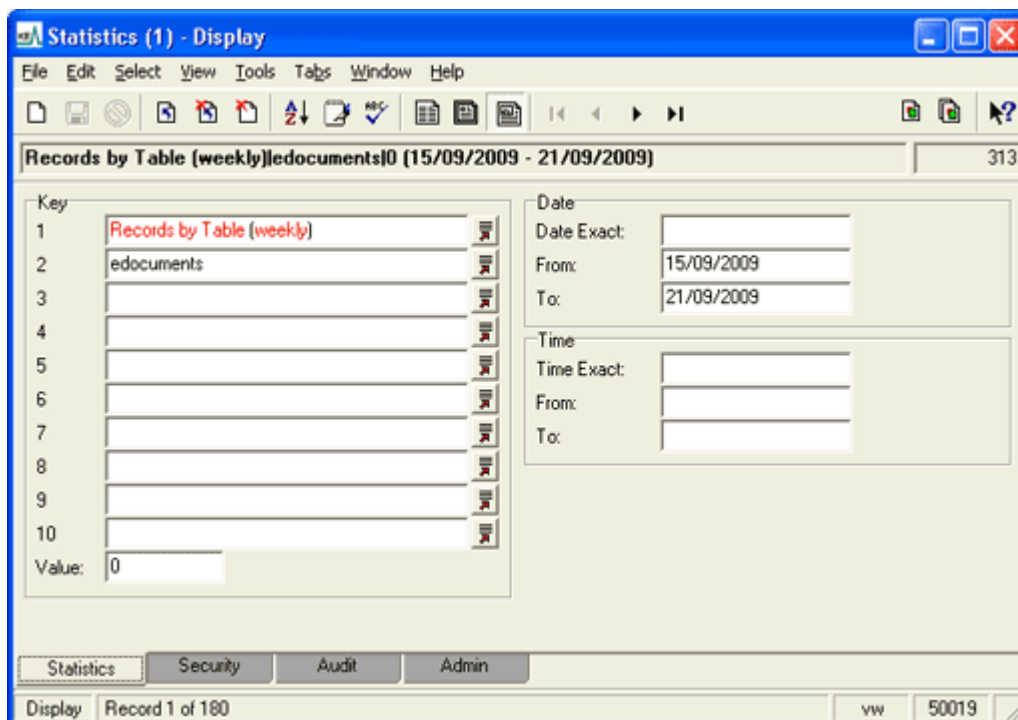
Requirements

- For Windows 2000, XP, 2003, [Microsoft Windows Services for UNIX](#) (version 3.5)
- [KE Texpress 8.1.020](#) or later
- [KE TexAPI 3.1.016](#) or later
- [Perl 5.8](#) or later

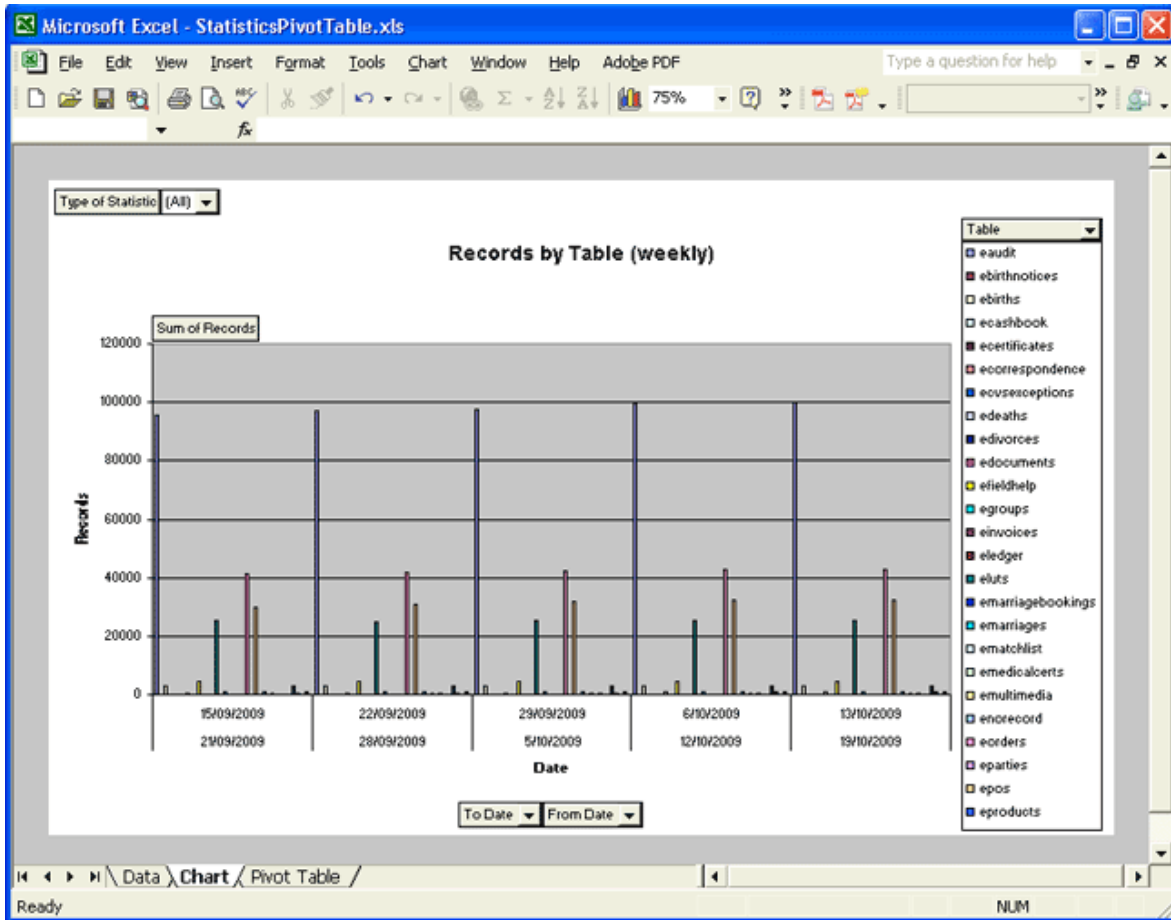
Updates / New Features

- **Statistics Module:** A facility has been added that allows statistical information to be gathered and stored at regular intervals. The information is maintained in a new module called *Statistics*. Tasks are executed on the Vitalware server to populate *Statistics* records with a numeric value. The resulting records may be used to produce numeric and graphical reports. Statistical data may be generated:
 - hourly
 - daily
 - weekly
 - monthly
 - at user defined periods (e.g. fortnightly, annually, etc.)

Statistical data may also be added manually, if required. The image below shows the structure of a *Statistics* record:



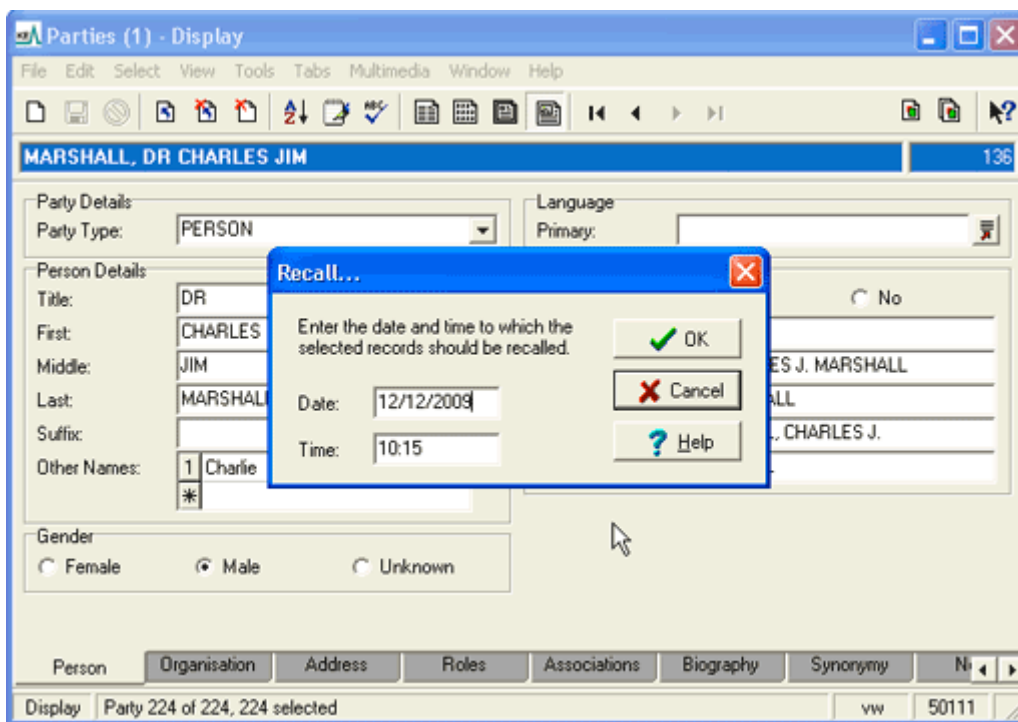
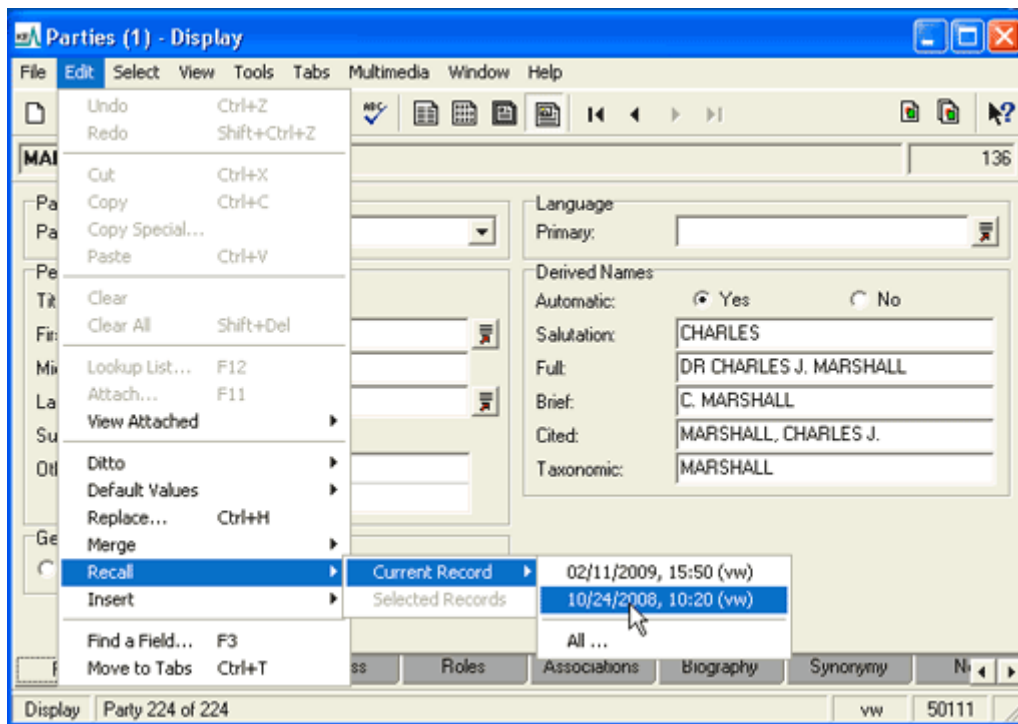
A sample report is shown below:



System Administrators may add new tasks to generate site specific statistics.

A complete description of the new module and periodic tasks can be found in the Statistics module documentation.

- Record Recall:** The addition of record level auditing in [Vitalware 2.0.03](#) provided the possibility for restoring records to previous versions. A new facility has been added that allows a record to be restored to the values it contained at a specified date and time. A batch command allows a set of records to be reset. Users require the **daRecall** operational privilege to be able to use the facility. The images below show the new commands:



A complete description of the new facility can be found in the Record Recall documentation.

- **Record Templates:** The new Record Templates facility allows an existing record to be used as the basis for producing one or more new records. A wizard is provided to streamline the record creation process. A number of features are available:
 - Data from the existing record may be copied into the new records
 - Static values may be added to the new records
 - Users may be prompted for values that are added to the new records

- Starting values for incrementing fields may be supplied, along with an incrementation value
- Variable data may be added to the new records (e.g. record creation number)
- Blocks of IRNs may be preserved

Record Templates

Choose how many records to create.

Select the number of records to create and a starting IRN and click Next, or accept the defaults and click Next.

Number of records to create:

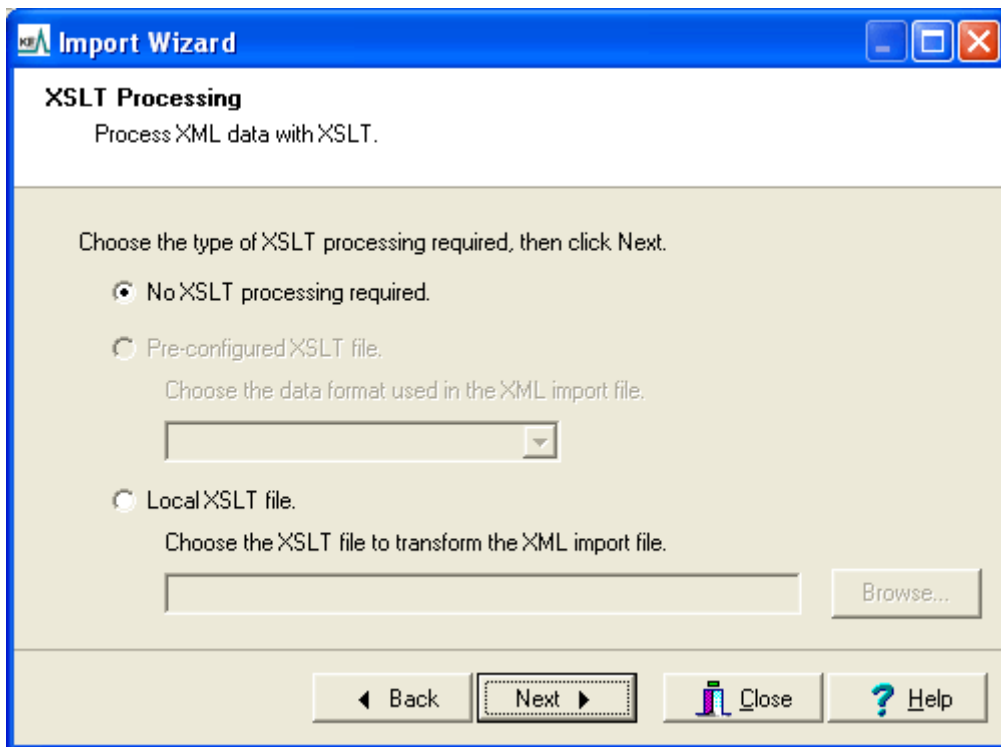
Starting IRN: (empty = next available IRN)

◀ Back Next ▶ Close ? Help

The Record Template facility allows a number of duplicates to be made of an existing record. It provides a useful means for reserving a block of numbers (e.g. Registration Numbers, Stock Numbers, IRNs, etc.) that will be completed at a later date.

A complete description of the new facility can be found in the Record Templates documentation.

- **Import XSLT Processing:** The Import facility has been extended to allow XSLT pre-processing before XML data is imported into Vitalware. A series of pre-configured templates may be used or users may define their own templates. The Import wizard has been extended to allow templates to be tested before importing data. A complete description of the extension may be found in the XSLT Processing documentation.



- **FIFO Service:** A generic framework has been added to the Vitalware server allowing complex calculations to be performed with the results used to populate data values. The framework removes the old system() scheme leading to significant improvements in speed and flexibility. A series of plugins is used to perform the calculations. The new service is used for:
 - calculating turn-around times for product insurance
 - determining multi-lingual values for system settings

A complete description of the new server can be found in the FIFO Server documentation.

- **Multimedia Upgrade:** Replacement of the old multimedia components with new versions is complete. All controls where media is attached and viewed now use the same sub-system. The new sub-system provides support for a large selection of audio, video and [image formats](#). In particular improved support is available for the following formats:
 - AV
 - MPEG
 - WMV
 - MP3
- **Multimedia Drag and Drop:** A multimedia file (image, audio, video, documents, etc.) or URL may be dragged from the Windows environment and dropped on any control displaying multimedia. A new Multimedia record is created for the dropped item with the resulting record attached automatically to the control. Allowing multimedia to be added via drag and drop may streamline the data entry process, however the Multimedia record created will only contain minimal data (default values, extracted metadata, generated resolutions, etc.). If extra data exists, the multimedia record may need to be updated manually.

- **System Tuning:** New tools are available to provide more optimal indexes for searching. In general, the Vitalware server will configure itself to provide near optimal searching performance for most data sets. The new tools allow fine-tuning of the indexes to provide even faster retrieval and to minimise the amount of storage required. System Administrators may configure the system manually to achieve optimal performance for irregular data sets. A complete description of the new tools may be found in the Configuration and Range Indexing documentation. Note that System Tuning requires Texpress 8.2.
- **XSLT Report Viewer:** The XSLT Report viewer has been rewritten. The previous version wrapped the generated XML with a stylesheet processing instruction, requiring the browser to perform the transformation from XML to HTML. Since web browsers provide different implementations of the XSL standard, the generated HTML may vary from browser to browser, resulting possibly in erroneous views. The new version of the viewer performs the transformations internally (rather than using the browser) using the MSXML engine. Page consistency is guaranteed, regardless of the browser employed, as a single engine is used.
- **Image Resolution:** A new Registry entry has been added that allows the resolution (dots per inch / cm) and resolution unit (inch / cm) to be defined for derived images. Using this entry, derived images can be altered to contain a resolution suitable for embedding directly in Crystal reports via their OLE Graphic object. The format of the new Registry entries is:

```
System|Setting|Multimedia|Metadata|Set|Property|format|Re
solution|x:y
System|Setting|Multimedia|Metadata|Set|Property|format|Re
solution Unit|unit
```

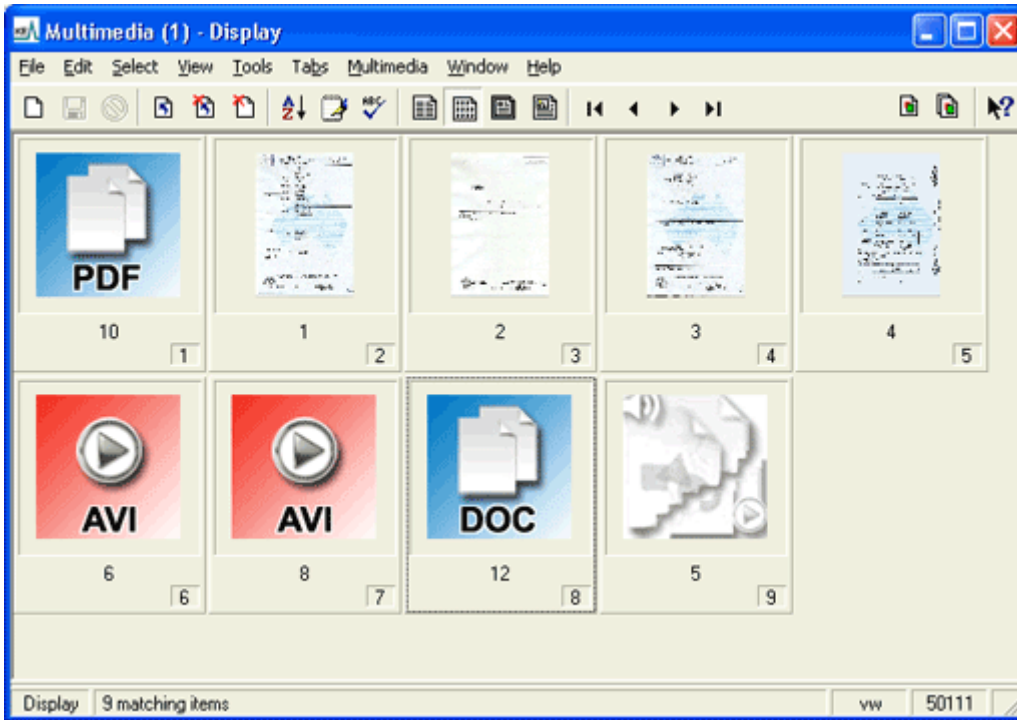
where:

format is the image format for which derived images will have the resolution set. A value of **Default** may be used to indicate the resolution should be set for all derived images. Individual image formats may also have an image width and height specified to further restrict the setting of the resolution. For example, a *format* of **90:100:JPG** would indicate that only 90 x 100 (width x height) pixel JPEG images should have the resolution set.

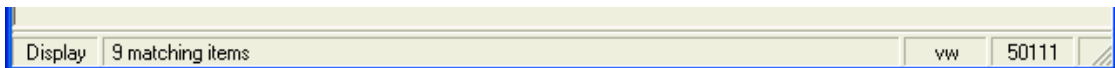
x:y specifies the resolution (*width:height*) to place in the derived image. The resolution unit is defined by the second Registry entry where the following values are available for *unit*:

- 0 - Clear any existing resolution
- 1 - Resolution unit is DPI (dots per inch)
- 2 - Resolution unit is DPCM (dots per centimetre)

- **Media Thumbnails:** The thumbnails displayed where multimedia is embedded in a tab have been upgraded to provide consistent colouring and detail. The image below shows some of the new thumbnails:



- **Crystal Viewer Upgrade:** The Crystal Reports viewer has been upgraded to Crystal Reports XI Release 2. A number of Crystal Reports issues have been resolved in the latest version. Support for French, German, Italian, Spanish and Swedish is available in the new viewer. To install the new viewer a **Standalone** or **Network Client** installation is required on all user computers.
- **User identification:** The status bar, located at the bottom of each module, now includes the user name of the person logged in and the service (port number) to which they are connected. The information provides an easy way to determine the EMu server to which you are connected, particularly if you are logged in more than once:



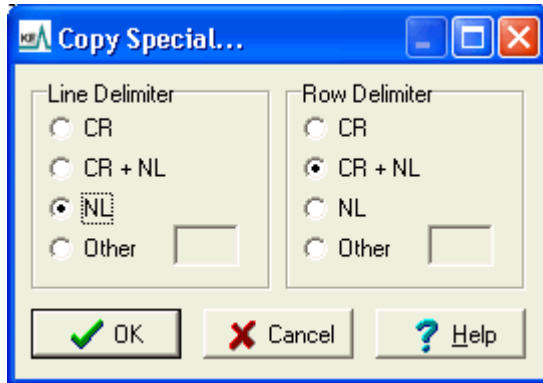
- **Image Quality:** The Registry entry used to determine what resolutions are derived when an image is added to the Multimedia repository has been extended to allow the image quality to be defined. The quality is used by *lossy* formats (e.g. JPEG) to determine the level of image loss tolerated. The value is between 1 and 100, where 1 indicates a complete loss of quality and 100 represents no loss of quality. The format of the Registry entry is:

```
System|Setting|Multimedia|Resolutions|format|Resolution|resolution;...
```

where *resolution* consists of up to six colon separated values:

- image width in pixels
- image height in pixels
- image type (e.g. JPEG, TIFF, etc.)
- scale image, that is maintain aspect ratio when resizing (TRUE or FALSE) [default: **TRUE**]
- enlarge image, that is allow derivative to exceed size of original image (TRUE or FALSE) [default: **TRUE**]

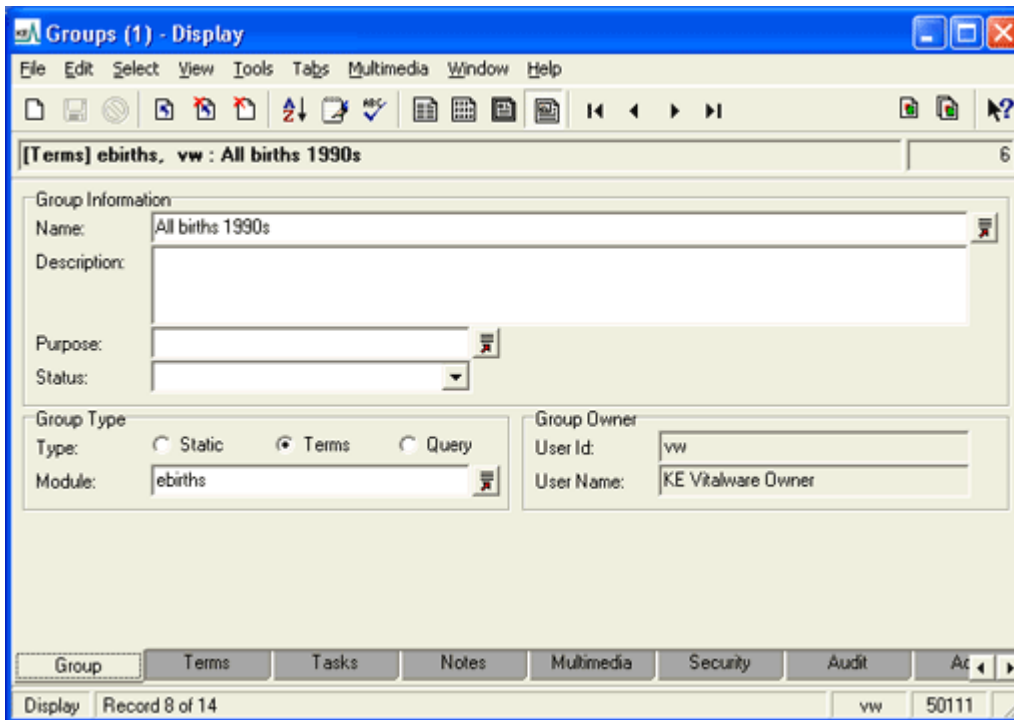
- compression to apply (e.g. NONE, FAX3) [default: **NONE**]
 - image quality (value between 1 and 100) [default: **100**]
- **Copy Special:** A Copy Special command has been added under the Edit menu to allow the end of line (used within text to end a paragraph) and end of row (used between records) delimiters to be defined. The delimiters to use may vary depending on the application into which records are being pasted. The image below shows the delimiters available:



- **Groups Module:** The Groups module has been extended. A number of new tabs have been added:
 - Tasks
 - Multimedia
 - Notes
 - Legacy Data

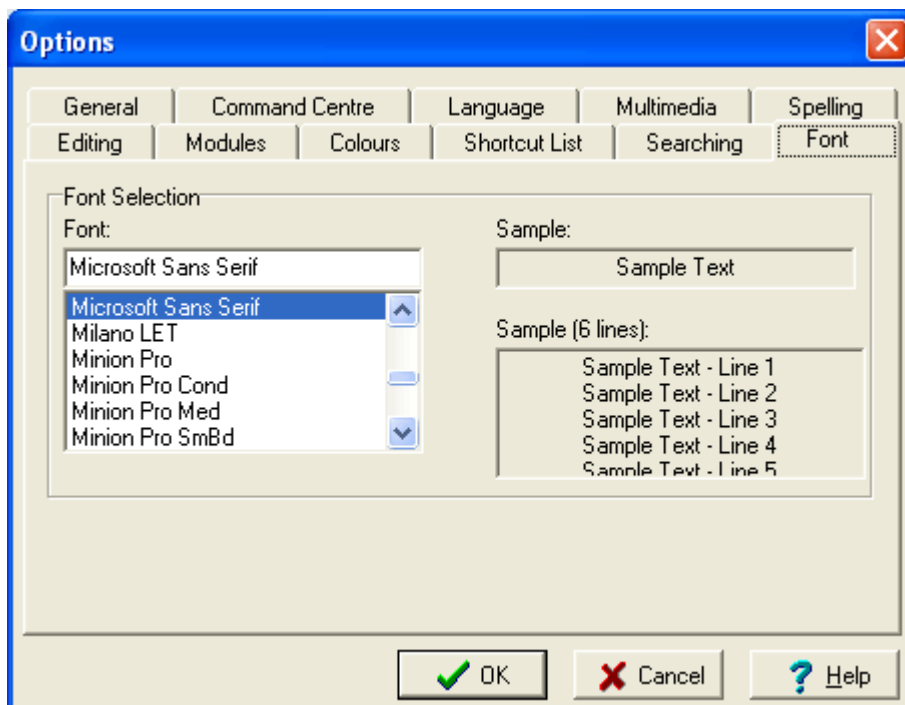
Three new fields have also been added:

- Group Purpose
- Group Description
- Group Status



The new fields and tabs allow full documentation of a group, rather than just recording the group name.

- **Font Setting:** A new option allows the font used to display data to be set. The option is useful for LCD screens where the standard Windows font may be difficult to view. The image below shows the new Font tab option:



The default font has changed from **MS Sans Serif** to **Microsoft Sans Serif**.

- **Security Extension:** The `Security` Registry entry has been expanded to allow the name of the user to be embedded in security values. The variable `$user` will be

replaced with the user name of the person running EMu. Using this variable, security may be adjusted on a per user basis depending on the user name stored in the data. The entry below grants editing privileges based on the user name stored in the *AdmTrustedUser* column in the Catalogue module:

```
Group|Default|Table|ecatalogue|Security|Edit|AdmTrustedUser=$user
```

- **Exact Lookup Match:** A new Registry entry has been created requiring that Lookup List values must match exactly against values entered, that is, the character case and punctuation must be the same as the Lookup List entry. The format of the Registry entry is:

```
Group|group|Table|table|Lookup Exact|colname|true
```

where *colname* is the name of the column for which Lookup List entries must match exactly the value entered.

- **Improved Media Download:** The mechanism used to transfer media files from the EMu server to the client machine has been replaced with a more efficient protocol. The new mechanism has resulted in download time improvements of between 300% and 700%.
- **Multimedia re-arrangement:** Significant speed improvements have been made when re-arranging attached multimedia. The improvement is significant if a large number of multimedia attachments exist.
- **Image Display:** The Image Display Registry has been extended to allow testing for empty and filled fields. The new Registry entries are:

```
Group|group|Table|table|Image Display|colname|NULL|image
Group|group|Table|table|Image Display|colname|NOT
NULL|image
```

where *NULL* matches an empty field and *NOT NULL* matches a filled field.

- **Barcode Stock functionality:** Management of Security Stock (from creation of stock records through to distribution, return, etc. of Security stock) can be managed using a Barcode Scanner.
- **Receipt Printer:** Added ability to use more than one type of receipt printer. Where additional receipt printer types are used they are specified using the *Receipt Printer Name* Registry entry. The format of the entry is:

```
Group|Default|Table|epos|Receipt Printer Name|Name
```

where *Name* is the name that Windows uses to refer to the printer.

- A facility was added to retire products in the Products database that are no longer offered to customers but appear on historic records. This is done by setting the Status field to *Retired*.
- Added check so that records which are restricted from printing cannot be attached to certificate orders.
- Altered processing of the removal of an attached event for a certificate so that if a certificate had been printed, the removal of the event requires supervisor authorisation.
- Added the ability to reset all link grids to the top left position when inserting a new record. This functionality is controlled by the *Reset LinkGrids* Registry entry. The format of the entry is:

```
Group|Default|Table|Default|Reset LinkGrids|boolean
```

where *boolean* is *True* (the cursor will move to the top left of a link grid when a user

enters Insert / New mode) or `False` (the cursor will remain in the same position in a link grid when a user enters Insert/ New mode).

Issues Resolved

Issue	Resolution
<ul style="list-style-type: none">The Audit facility introduced in the previous release requires each record created to have a unique IRN (Internal Record Number). If a record is deleted, the number should not be re-used, otherwise audit trail data for the new record may display information about a deleted record.	The default value for autokeyreuse has changed from <code>yes</code> to <code>no</code> . The change guarantees IRNs are never re-used, ensuring audit trails refer to the current record only.
<ul style="list-style-type: none">The error message Column operation performed before row has been accessed may appear after sorting a matching set of records or retrieving a set of records via the Groups facility. After the error is shown, the data for the matching records may be blank (although the record count is correct).	The error no longer appears and the records are displayed correctly.
<ul style="list-style-type: none">If the Security tab is resized the Add and Remove buttons may not be positioned correctly after the tab layout is adjusted.	The Add and Remove buttons are now positioned correctly.
<ul style="list-style-type: none">A number of controls on query tabs do not allow multiple values to be entered, hence restricting OR based queries to additional searches.	All query controls now allow multiple values to be entered, enabling alternative terms to be entered in the same search.
<ul style="list-style-type: none">An invalid number format or Invalid query specified error is shown if a value greater than 2,147,483,648 is entered into a numeric (integer) field.	The maximum allowable value for a numeric field has been increased to 9,999,999,999.
<ul style="list-style-type: none">An Access Violation error may occur when extracting XMP metadata where multiple RDF xmlns attributes are defined.	Multiple xmlns attributes are now handled correctly.
<ul style="list-style-type: none">If many columns are defined for viewing via Shortcuts, moving between records may be slow, even if Shortcuts are disabled. The columns are loaded from the Vitalware server regardless of whether Shortcuts are displayed.	When Shortcuts are disabled, the columns are no longer fetched from the Vitalware server. The change results in faster movement between records.
<ul style="list-style-type: none">An Access Violation error may occur in the Multimedia repository for audio, video and document based media where the Documents table is empty. Under normal operation the table always contains the master media entry; however if data is	An error is no longer displayed if the Documents table is empty.

loaded from external sources, the table may be empty.

- An **Access Violation** error may occur when moving through records where a read-only grid changes from having no entries to having one entry. An error is no longer displayed where a read-only grid changes from having no entries to having one entry.
- If controls have been removed from a tab (as a result of sub-classing), the remaining controls may not be laid out correctly when the tab is resized. The controls in sub-classed tabs, when resized, are laid out correctly.
- Under rare circumstances it is possible to have the same IRN assigned to two different records. The sequence of events required to produce the error involves a complex interaction between a series of users. This sequence of events no longer causes an error.
- An **End of File** error may be produced when performing a search and displaying matches in either List or Contact Sheet mode. The error is displayed only if the results contain a "false" match. "False" matches are handled correctly in List and Contact Sheet modes.
- An image in a report may not be displayed if the file name contains an ampersand (&) character. The image is displayed correctly if the file name contains an ampersand.
- Some multimedia helper applications may not start correctly when Launch Viewer is selected. Older Windows based programs may not accept the slash character as a valid path separator (e.g. MS Excel). All paths to multimedia files are now translated to "old" Windows based paths (using the back slash character) before launching the associated viewer.
- An **Access Violation** error may display when using drag and drop to re-arrange the order of a large number of multimedia attachments. The error message no longer appears when re-arranging multimedia attachments.
- The Undo command may not rewind changes made to the order of multimedia attachments. An incorrect order may result where a large number of multimedia attachments exists. The Undo command resets the order of multimedia attachments correctly.
- The Vitalware help file does not load correctly under Windows Vista. Microsoft has modified the mechanism used to load The Vitalware help file now loads correctly under all versions of Windows.

help files under Vista.

- The Summary Data may not be displayed in bold when moving between records under some circumstances. The Summary Data is always displayed in bold.
- A module may be created off the viewable screen area if the Save Last Position option is enabled and the viewable size of the monitor is reduced (either by moving from dual monitors to a single monitor or moving from a large monitor to a smaller monitor). As the new module cannot be seen it may be difficult to move it to a viewable area. All modules are now created within the viewable area of the monitor regardless of the Save Last Position setting.
- When re-attaching media to an existing Multimedia record the original version of the media may not be removed correctly where multiple storage areas exist on the Vitalware server (as defined by the **ServerMediaPath** Registry setting). Previous versions of media are cleaned up correctly when multiple storage areas are specified.
- The Paste and Paste (Insert) commands are not enabled when the clipboard contains some invalid data for pasting into grids. A better solution is to allow valid values to be pasted with an error displayed for invalid values. The Paste and Paste (Insert) commands are now enabled even if invalid values are part of the clipboard data.
- A report consisting of image and non-image based multimedia within the same record, where a resolution has been specified for the images, may not display the images. The images are now displayed correctly for reports where the record contains image and non-image based multimedia.
- The layout of controls on a tab may not be correct if a new module is opened and the Save Last Position and Save Last Size options are enabled and the module is maximised. The controls are laid out correctly if a new module is opened and maximised.
- The time required to save a new record may be increased if another instance of the same module is open and a large number of matching records are displayed. The time taken to insert a new record is not lengthened if another instance of the module is open.
- Incorrect data may be displayed when performing a spell check on a text field that is part of a nested form. The data displayed is for another row in the form, rather than The correct data is displayed when performing a spell check in a nested form.

the row with the bad term.

- Only the first row of a multiple row Paste or Paste (Insert) may be added to a grid if multiple rows are extracted from the same attachment module. All rows are pasted correctly where multiple columns are extracted from the same attachment module.
- The metadata values (EXIF, IPTC, XMP) are copied into a new record when Ditto All is selected. The metadata values are no longer copied when Ditto All is selected.
- The What's this Help? dialogue contains information about the control selected. It would be useful if the information could be copied and added to the clipboard for use by other applications. The field help information may be selected and copied to the clipboard.
- The resizing of CMYK images may result in the C (Cyan), M (Magenta) and Y (Yellow) planes being resized correctly, however the K (black) plane may not be resized, causing a distortion in the final image. CMYK images are resized correctly.
- The Audit facility does not handle back slash characters correctly. The character is removed from audit trail data. Back slash characters are handled correctly and kept in audit trail data.
- The tabs located at the bottom of the module window may become disproportionately large when the Windows Big Font feature is enabled. The tabs increase size in proportion to the rest of the module.
- The Attach Media button on the Multimedia tab may be disabled the first time the tab is visited, even if media could be attached. The Attach Media button is enabled if media may be attached when the Multimedia tab is first visited.
- A **Primary Key is not assigned** error may appear when completing one insertion and starting the next if the New Record command is entered before the saving of the previous record has completed. A new insertion cannot commence before the saving of the previous record is complete.
- The Date Modified and Time Modified fields are not updated when a record is changed in the Field Help module. The Date Modified and Time Modified fields are updated when the record is saved.
- The ownership of created reports may not be correct if a Registry entry exists for a report without a title (the entry can only be entered The ownership of created reports is correct even if a bad Reports Registry entry exists.

- A report containing images may not adhere to the report resolution settings when the reporting module is not the Multimedia module. The master image may be used rather than the resolution specified. The correct image resolution is used.
- An Admin Task may not stop executing when the Abort button is clicked. The Admin Task is terminated when the Abort button is clicked.
- The Audit facility does not generate the correct column name for double nested grids of latitude, longitude, date and time columns. The correct column name is generated. The upgrade process to Vitalware 4.0.01 corrects any bad column names.
- The Audit facility may add redundant empty values to audit records where a column computes to an empty value. Empty values are not added to audit records. The upgrade process to Vitalware 4.0.01 removes any redundant empty values.
- The Audit facility may place the client specific catalogue name as the table name in audit records, rather than the generic ecatalogue table. The generic ecatalogue table name appears in all catalogue audit records.
- The Program field in the Audit module may display the time value rather than the program name. The program name is shown in the Program field.
- The column widths configured for List View mode may not be preserved correctly if the total width of all columns exceeds the width of the module. The column widths are preserved correctly.
- An incorrect attachment query may be generated from Search mode if a grid control already contains attachments which are cleared after which the attachment button is clicked. The attachment query is no longer generated if the grid has been cleared.
- An incorrect query may be generated where two or more query controls contain attachment queries (that is records have been attached to query control). An **OR** based query is generated rather than the required **AND** query. The correct query is generated where multiple controls contain attachment queries.
- The Update Resource command in the Multimedia module may truncate IPTC metadata where the metadata contains a **null**. All **null** characters are removed from IPTC data when the metadata is extracted. A **null** character is not valid

character (that is, a non-existent character at code point zero) under the IPTC standard.

- A module may start with the maximum screen size, but without the maximise button enabled if the **Save Last Size** option is selected and the module was last closed when maximised. The module starts maximised with the button enabled.
- The values in calculated fields may not display correctly if a non-standard colour is set for calculated data. Calculated values display correctly, even when a non-standard colour is used.
- An **Index out of bounds** error may be displayed if rows in a grid are deleted from the top down while in Query mode. The error message no longer appears.
- An **Access Violation** error may appear if the **Abort** button is clicked when sorting records where a summary is requested. The error message no longer appears.
- An **Access Violation** error may appear when right clicking an image thumbnail and viewing the Launch Viewer sub-menu. The error message no longer appears.
- The Audit Trails and Groups modules may not display module specific documentation when help is requested. Module specific help is displayed.
- It was possible for a certificate to print even though payment for it was pending. The certificate is no longer printed.
- When changing the status of an order it was possible to receive an **Index out of range** error. The status may now be changed without error.
- When inserting a POS transaction it was possible for the date and time inserted values to be incorrect. The date and time values are now correct.
- When printing the cheque run it was possible to use the same cheque number more than once. Cheque numbers cannot be re-used.
- On occasions the applicant and refund details did not auto-populate from the delivery details. The applicant and refund details are now populated.
- When rolling back ledger records due to The refund ledger record is now rolled

- failure to save a POS transaction, a refund ledger record may not have been rolled back.
- When an invoice was paid the Till menu state still reflected that no Till was signed on. The Till menu state shows the Till as signed on.
- When a **Cannot focus disabled window** message was displayed as a result of failing to save a POS record, record roll back was not performed. Record roll back is now performed.
- When resizing the POS module the payment group boxes did not resize. The payment group boxes resize.
- On rare occasions an empty ledger record could be inserted. Empty ledger records are no longer inserted.
- On occasions when delivery details were dragged and dropped from Parties, other changes on the POS transaction could be lost. All changes are correctly retained.
- On rare occasions when selecting the reprint menu options an additional ledger record could be created. Additional ledger records are no longer created.
- After a lookup table error it was possible for an under-payments ledger record to be created instead of a refund. A refund ledger record is now created.
- A product was indicated as not valid for reversal after its name had recently been changed. Check for validity was altered from product name to product code to cater for product name changes.
- When back filling was turned on and multiple search columns were in use, on occasions the back fill of a field would be empty. The back filling code now checks all columns to try to find a value for use in the field.
- When back filling from a column that was a nested table, the back filled details could overwrite values in other rows in the POS record. No other values are overwritten.
- After making a change to the status on the Sales tab and then swapping to the Single Sale tab, the changed status was not. The status is correctly updated on the Single Sales tab.

reflected.

- When transmitting records from one Vitalware environment to another, the same record could be transmitted more than once. The record is only transmitted once.
- When printing certificates, the same certificates could be printed more times than was ordered. The certificate is only printed the number of times it is ordered.
- When module caching was being used, after viewing a record's history, a user's module permissions may no longer have been correct. The user's module permissions are now correct.
- Some registration menu options which require edit privilege would be shown as enabled even though a user did not have edit permission. The registration menu options are disabled when a user does not have edit permission.
- When adding a user note to a record that was already being edited, the client application would hang. An error message is now displayed indicating that the record is being edited.
- When double keying a richedit box that spanned multiple lines, the comparison function could at times fail even though the two values were the same. The comparison function correctly identifies the two keyings as the same.
- When double keying a record and a value was entered for the second keying where no value was entered for the first, an access violation could be shown. The access violation no longer appears.

Vitalware Documentation

Statistics

Document Version 1.0

Vitalware Version 2.1



Contents

SECTION 1	Statistics Facility	3
	Overview	3
	Statistics Module	4
	Reporting	7
	Periodic Tasks	11
	vwperiodic	12
	Tasks	14
	Creating a new period	19
	Regenerate missing data	20
SECTION 2	Appendix A - KE::Statistics perl module	21
	Name	21
	Synopsis	22
	Description	23
	KE::Statistics::Session	24
	Methods	25
	KE::Statistics::ResultSet	27
	Methods	28
	KE::Statistics::Date	29
	Methods	30
	KE::Statistics::Statistics	33
	Methods	34
	Bugs	36
	See Also	37

SECTION 1

Statistics Facility

Overview

As institutions continue with their Vitalware implementations, the question of statistical analysis of system operations and data content inevitably arises. System administrators and managers require reports showing the number and type of operations performed on a per user basis, e.g. the number of insertions into the Catalogue module on a daily basis for the past month listed by user. The answer to this request is found in the records in the Audit module. In order to produce the information in a reportable format it is necessary to perform a number of searches of the Audit information and collate the results into a spreadsheet, which can then be graphed or tabulated. The process may be quite time consuming and tedious, and if the same information is required again at a future date, the same steps need to be repeated to get the same results.

Vitalware 2.1.01 introduces a Statistics facility that allows statistical information to be generated on a regular basis (hourly, daily, weekly or monthly) and stored in the Statistics module for later use. System administrators and managers need only search the Statistics module to locate the information they require and then produce a report (Excel Pivot table) from which tables and graphs may be generated.

The Statistics facility consists of two parts:

- **Statistics Module**

The Statistics module contains records with computed statistical values. Each record contains one value, a floating point number, that represents the result of a statistical criteria. For example, a value of 10 may indicate the number of records inserted by user `james` into the Births module on 17 February 2009. A standard Vitalware module interface is provided to the Statistics module. An Excel report is supplied that presents the records in a Pivot table for further manipulation.

- **Periodic Tasks**

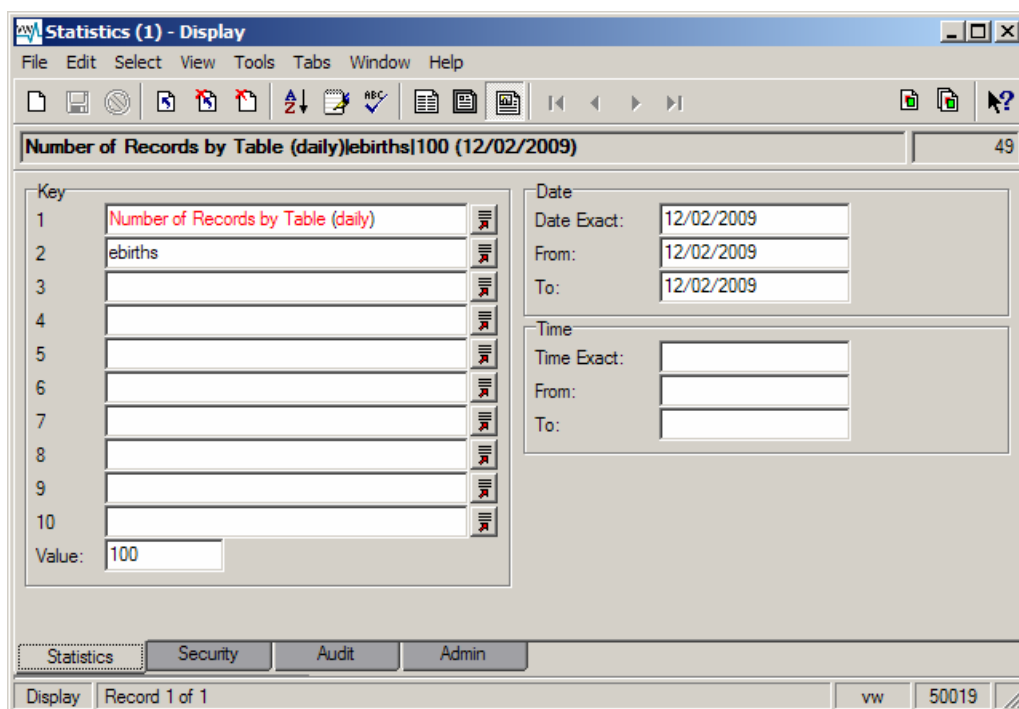
In order to provide useful statistical information it is necessary to have statistic records generated at regular intervals, removing the need for information to be obtained manually. The Periodic Tasks facility implements a framework in which individual tasks (scripts) can be placed and executed on a regular basis. It is the purpose of the tasks to generate statistical records by examining the various system reports and data within a Vitalware implementation. Periodic tasks can be run on an hourly, daily, weekly or monthly basis. It is possible to add new periods (e.g. fortnightly) if required.

Statistics Module

Vitalware 2.1.01 sees the addition of the Statistics module. Designed to hold statistical data, the module stores one statistical value per record. The value is computed by a task, which is charged with creating the record. Administrators can search the module to retrieve sequences of records used to produce reports.

The module consists of a Statistics tab that contains all the information about the statistical data. The other tabs are:

- Security - controls access to the data.
- Audit - lists auditable operations performed on the record.
- Admin - contains record creation and modification dates/times.



The Statistics tab stores three discrete pieces of information:

- **Keys and Value**

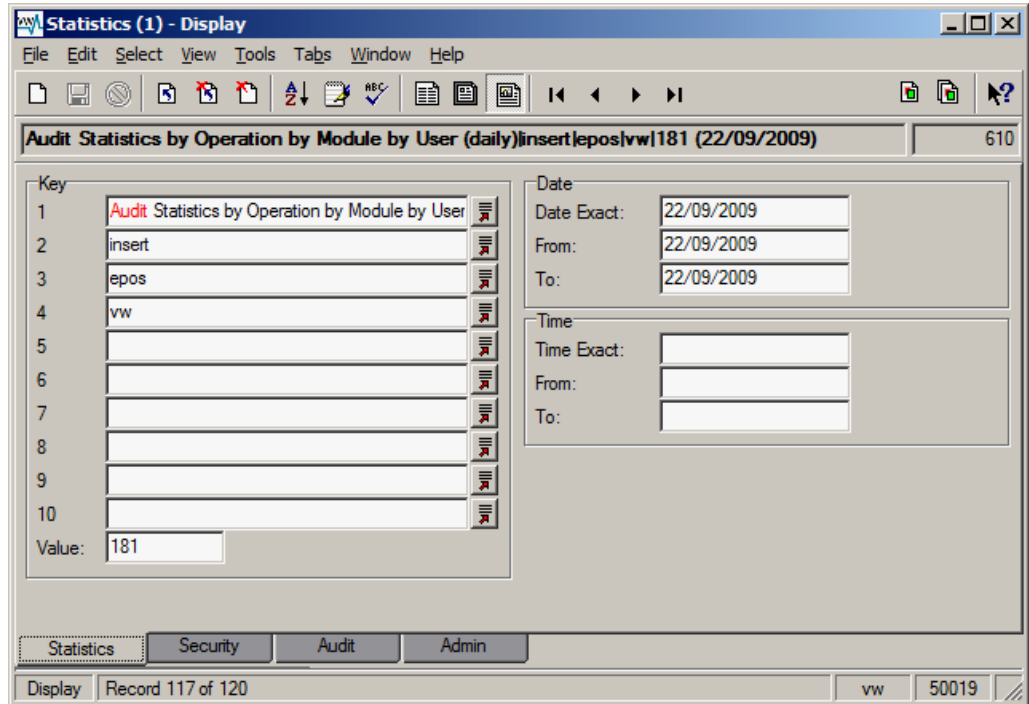
The Keys describe the type of statistical value stored in the record. A record consists of a number of hierarchical keys in which each level defines a variable piece of information for the statistic generated. The top level is reserved for the type of record. In the image above the first *Key* has a value of Number of Records By Table (daily). Three pieces of information are contained within the title:

- The *Value* of the record is a record count (Number of Records).
- The record count is generated on a per table basis (by Table).
- The *Value* is generated daily (daily).

The second *Key* (ebirths) indicates the table for which the record count applies. Thus, the record above contains the number of records in the ebirths table generated daily. In general, the title of the record should use the word *by* to indicate what variables are contained within the record. For example, a title

of Audit Statistics by Operation by Module by User (daily) would indicate that the record contains a count of the number of audit operations (insertions, edits, deletions, etc.) on a per table basis for each individual user. The *Value* represents the number of operations on a daily basis. Given this title, *Key 2* would contain the audit operation type, *Key 3* the table name and *Key 4* the user name.

The *Value* is a floating point number containing the numeric value defined by the Keys. In most instances the *Value* is an integer, however if averages are computed, the fractional part may be required.



• **Dates**

Three dates are provided: depending on the period of the statistical record, some or all of them may be filled:

- *Exact* - filled for data that is gathered within a single day (daily and hourly).
- *From* - the commencement date for the period. A commencement date should always be supplied.
- *To* - the completion date for the period. A completion date should always be supplied. If the period is a day or less, the commencement and completion dates are the same as the *Exact* date.

The date fields are used to define the day or range of days covered by the statistical value. The values are very useful when performing searches to gather statistical information for reporting.

- **Times**



Three times are provided: depending on the period of the statistical record, some or all of them may be filled:

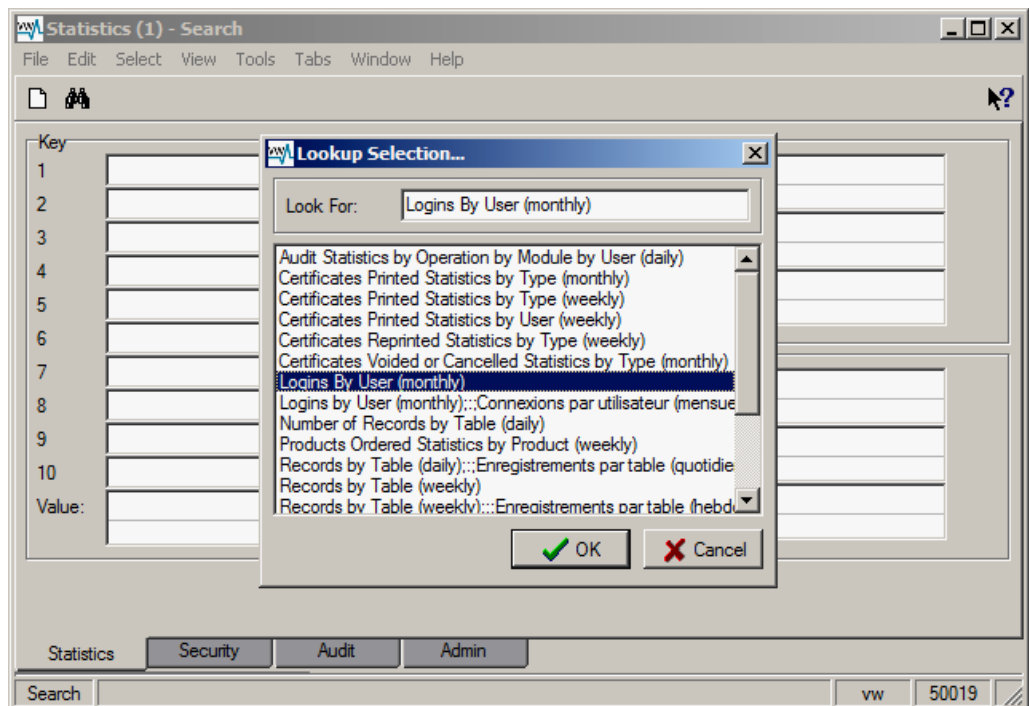
- *Exact* - filled for data that is gathered at a single point in time. Some hourly records represent a value at a fixed point in time, e.g. the number of users accessing the system. As this value represents the count at a fixed point in time, the *Exact* time field should be filled.
- *From* - the commencement time for the period. A commencement time should be supplied for tasks that are within a day (e.g. hourly).
- *To* - the completion time for the period. A completion time should be supplied for tasks that are within a day.

The time fields are used to define the point in time or range of time covered by the statistical value. If the value period is a day or longer, the time fields should be left empty. The values are very useful when performing searches to gather statistical information for reporting.

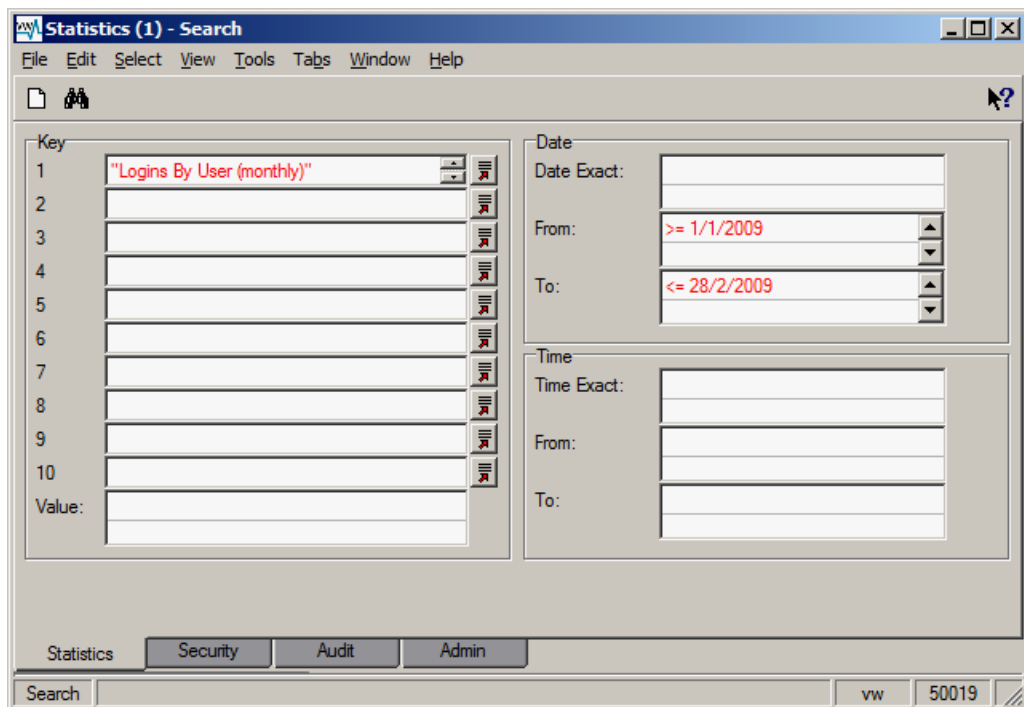
Reporting

The main reason for gathering statistical information is to produce reports. Reports may be tables of data, or more graphical representations such as charts may be used. The Statistics module provides one report only, the Excel based Statistics Pivot Table report. Before we can produce a report, it is necessary to retrieve the data on which to report. The steps below outline the process required to produce a statistical report:

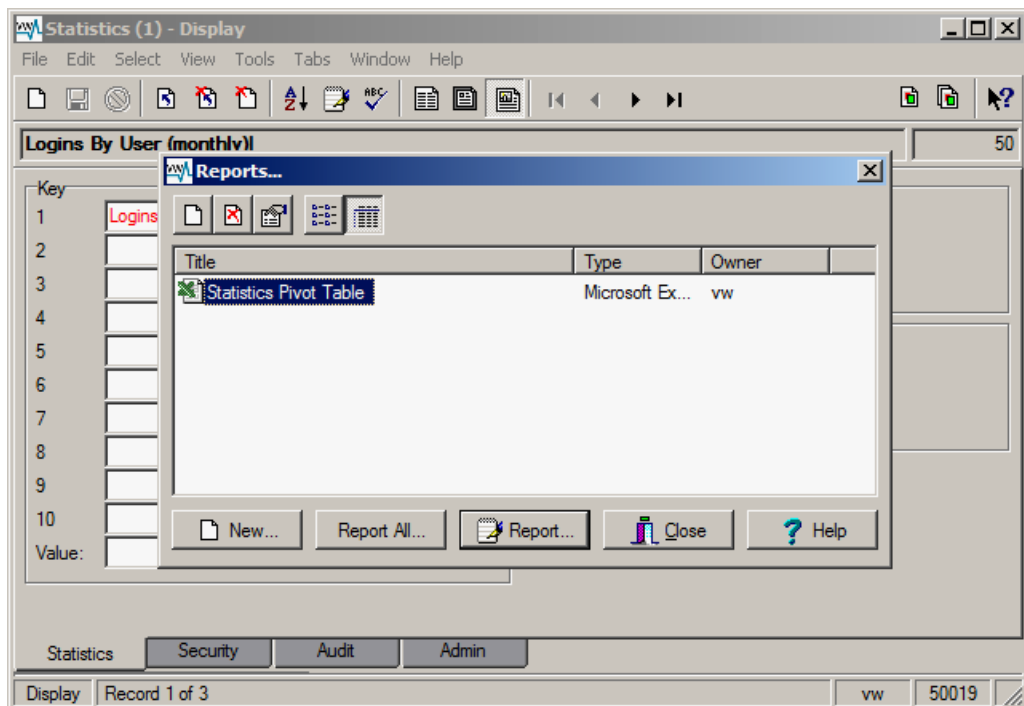
1. Open the Statistics module by selecting the **Statistics**  button in the Command Centre.
2. Select the **Lookup List**  button for *Key 1*. A list of all the statistical data maintained by the system is displayed:




3. Select the entry for the report type to be produced, e.g. **Logins by User (monthly)**.
4. If reporting on a single user or list of users as opposed to all users, the *Key 2* Lookup List could be used to select the required user names. In general, if a specific value or list of values is required for any given Key, the associated Lookup List can be used to select the values. If all values are to be reported, the Key should be left empty. In this example we want to report on the number of logins on a user basis for all users, so we leave *Key 2* empty.
5. Specify the date range on which to report. In general this requires specifying a *From* date and a *To* date. In this example we want all records for January and February 2009:

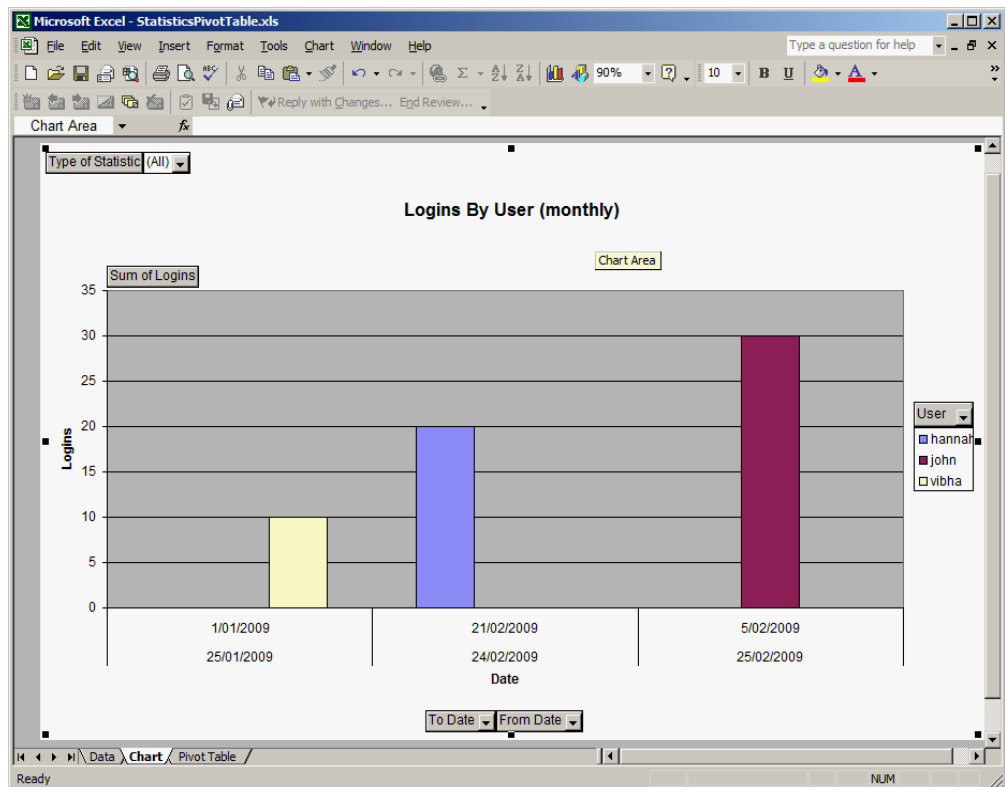


6. Perform the search to retrieve the required statistical records.
7. Select **Tools>Reports** from the Menu bar to display the Reports dialogue box and select the **Statistics Pivot Table** report:



8. Select **Report All**  to generate the report.
An Excel report will display.

 The report requires macros to be enabled so that a graph of the data can be produced.



Various tables and graphs can be produced as the data is now in a pivot table, based on the different *Key* values supplied.

	A	B	C	D	E	F	G
1	Type of Statistic	(All)					
2							
3	Sum of Logins		User				
4	To Date	From Date	hannah	john	vibha		
5	25/01/2009	1/01/2009			10		
6	24/02/2009	21/02/2009	20				
7	25/02/2009	5/02/2009		30			
8							
9							
10							
11							
12							

Once all the statistical information has been added to the Excel pivot table it is possible to manipulate any of the statistical variables by either restricting values or enabling all values. Pivot tables are extremely powerful and provide a very convenient mechanism for the production of reports with multiple statistical variables.

While Excel is the recommended tool for manipulating statistical data, it is possible to use any other reporting mechanism. If specialised output is required, it is possible to use Crystal Reports to produce the finished report. In this case it is recommended that the report is named after the type of statistical information it expects to receive.

Periodic Tasks

So far we have examined the new Statistics module, learned how to search for statistical information and considered the reporting options available. Next we explore how statistical information is generated.

In order to create useful reports, it is necessary to populate the Statistics module with meaningful records. In the simplest case it is possible to create statistic records manually by collating system information and inserting new statistic records with the required keys, dates, times and value. However it would not take long before someone forgets to add the required records thus rendering the analysis incomplete.

The Periodic Tasks facility provides a framework in which tasks can be executed on a regular basis. Each task is a perl script generating one or more records for insertion into the estastistics table. At the heart of the framework is the **vwperiodic** program.

vwperiodic

The **vwperiodic** script is run at regular intervals to generate statistical information. Its primary purpose is to invoke all the task scripts for a given time period (hourly, daily, weekly, monthly). The usage message for **vwperiodic** is:

```
Usage: vwperiodic [-q] [-d yyyy:mm:dd[:HH:MM:SS]] period
```

where:

<code>-d yyyy:mm:dd[:HH:MM:SS]</code>	is the date to use for periodic tasks.
<code>-q</code>	specifies quiet mode, i.e. do not output progress.
<code>period</code>	specifies the time period for which statistical data is generated. Allowable values: <ul style="list-style-type: none"> • daily • hourly • monthly • weekly

Extra periods may be added, e.g. fortnightly, as required.

The Unix task scheduler cron is used to execute **vwperiodic** at the required intervals. The crontab entries used to invoke **vwperiodic** are:

```
#
# Run periodic tasks
#
30 * * * * /home/vw/client/bin/vwrun vwperiodic hourly 2>&1 |
/home/vw/client/bin/vwrun vwlogger -t "KE Vitalware Periodic Tasks
Report" periodic
0 6 * * * /home/vw/client/bin/vwrun vwperiodic daily 2>&1 |
/home/vw/client/bin/vwrun vwlogger -t "KE Vitalware Periodic Tasks
Report" periodic
30 6 * * 0 /home/vw/client/bin/vwrun vwperiodic weekly 2>&1 |
/home/vw/client/bin/vwrun vwlogger -t "KE Vitalware Periodic Tasks
Report" periodic
0 7 1 * * /home/vw/client/bin/vwrun vwperiodic monthly 2>&1 |
/home/vw/client/bin/vwrun vwlogger -t "KE Vitalware Periodic Tasks
Report" periodic
```

The table below shows when each instance of **vwperiodic** is executed:

Command	Executed
<code>vwperiodic hourly</code>	30 minutes past the hour being analysed.
<code>vwperiodic daily</code>	6 hours past the day being analysed.
<code>vwperiodic weekly</code>	6 hours and 30 minutes past the week being analysed, on the Sunday morning.
<code>vwperiodic monthly</code>	7 hours past the month being analysed.

All output from running periodic tasks is sent to **vwlogger** which places the output into a file based on the current date (`yyyy-mm-dd`) in the `logs/periodic` directory. The log files provide a useful starting point if you suspect a problem with the execution of periodic tasks. As you can see, each task period is invoked **after** the time period for which it is generating statistics. The execution is delayed in order to allow any activities started in the task period to complete before the periodic tasks are run. It is also important to ensure that any system maintenance routines are not running while periodic tasks are executing, otherwise access to required tables may be denied.

When **vwperiodic** is invoked it looks for periodic tasks stored in either:

- `etc/periodic/period`

-OR-

- `local/etc/periodic/period`

where *period* is the argument supplied to **vwperiodic** (e.g. `hourly`). Each task is a perl script with a `.pl` (perl library) extension. If more than one task is found in the above directories, each task is executed sequentially in alphabetical order. Tasks in `local/etc/periodic` override scripts with the same name in `etc/periodic`.

Tasks

Each task is a perl function called by **vwperiodic** to generate statistical information. The *bare-bones* perl required for a task is:

```
#!/usr/bin/env perl

#
# Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
use KE::Statistics;
no warnings 'redefine';

#
# Calculate the number of records per table.
#
sub
Periodic
{
    my $session = shift;
    my $date = shift;
    my $period = shift;

    #
    # Insert task code here
    #
}

```

vwperiodic calls the function `Periodic($session, $date, $period)` within the task script. The script then generates the statistical data and creates the required statistics record(s). The arguments to `Periodic()` are:

- `$session` A `KE::Statistics::Session` object provides a connection to the back-end database environment. The object may be used to gather information to generate statistical values and to create statistics records.
- `$date` A `KE::Statistics::Date` object contains the date and time at which **vwperiodic** was invoked. The `$date` object is used to determine the date/time range of the statistical information for the task invoked.
- `$period` A string that contains the name of the time period for the task being run. Typical periods include hourly, daily, weekly and monthly. Administrators may create new periods (e.g. fortnightly) as required, in which case `$period` will contain the name of the new period.

A perl module is provided to help with the creation of statistics records and the generation of statistical values. The module is `KE::Statistics` and must be included in a task to gain access to its objects (via `use KE::Statistics;`). The

module provides a suite of classes to manipulate statistical information. The classes are:

`KE::Statistics::Session`
(page 24)

A `KE::Statistics::Session` object is used to gather information from the back-end server. The object may query any table or set of tables to allow statistical information to be generated. A set of methods allow information about the server environment to be gathered (e.g. list of registered users, list of tables, etc.).

`KE::Statistics::ResultSet`
(page 27)

A `KE::Statistics::ResultSet` object is returned by the `KE::Statistics::Session->search($texql)` method. The object provides access to the records returned as a result of the specified query.

`KE::Statistics::Date` (page 29)

The `KE::Statistics::Date` object makes the manipulation of dates easier. The object contains a breakdown of a date (`{year}`, `{month}`, `{day}`, `{hour}`, `{minute}` and `{second}`). A number of methods are provided that allow the date/time to be manipulated.

`KE::Statistics::Statistics`
(page 33)

The `KE::Statistics::Statistics` object is designed to provide easy insertions into the `estatistics` table. A `Statistics` object allows the columns within a record to be set and the record written. A check is made to see if the record already exists in the table and if so an update is performed rather than an insertion. This allows periodic tasks to be re-run to refresh data without duplicate records being created.

The task script below is used to generate the number of records on a per table basis each day:

```
#!/usr/bin/env perl

#
# Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
use KE::Statistics;
no warnings 'redefine';

#
# Calculate the number of records per table.
#
sub Periodic
{
    my $session = shift;
    my $date = shift;
    my $period = shift;

    #
    # Run texlist -l and parse the results
    #
    my %data;
    for my $line (split(/\n/, `texlist -l`))
    {
        my @bits = split(/\s+/, $line);
        $data{$bits[0]} = $bits[2];
    }

    #
    # Create a statistics object we can use to insert
    # statistics records.
    #
    my $stats = $session->statistics();
    my $yesterday = $date->yesterday();
    $stats->setDate($yesterday);
    $stats->setDateFrom($yesterday);
    $stats->setDateTo($yesterday);
    $stats->setKey1("Records by Table ($period)");

    #
    # Now add the data for each type of operation
    #
    for my $table(keys %data)
    {
        $stats->setKey2($table);
        $stats->setValue($data{$table});
        $stats->write();
    }
}

1;
```

The example shows how it is possible to obtain a `KE::Statistics::Statistics` object (`$session->statistics()`) and use it to create statistics records. A point of interest is that the three `Date` values are set to yesterday's date. As the task is invoked 6 hours after the day has ended, it is necessary to use the date of the day before.

The task below generates statistical data about the number of audit operations performed on a per user and per table basis:

```
#!/usr/bin/env perl

#
# Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
use KE::Statistics;
no warnings 'redefine';

#
# Calculate user statistics for operations.
#
sub
Periodic
{
    my $session = shift;
    my $date = shift;
    my $period = shift;

    #
    # Zero the operations count for all users of all tables.
    #
    my $data = {};
    foreach my $user (@{$session->users()})
    {
        foreach my $table (@{$session->tables()})
        {
            foreach my $operation (@{$session->operations($table)})
            {
                $data->{$user}->{$table}->{$operation} = 0;
            }
        }
    }

    #
    # Get back all the records for the supplied date.
    #
    my $yesterday = $date->yesterday();
    my $query = "select AudUser, AudOperation, AudTable from
eaudit
                "where AudDate = DATE" . $session->quote() .
                $yesterday->dateText() . $session->quote();
    my $results = $session->search($query);
    die ("Invalid query $query") if (! defined($results));

    #

```

```

value      # Move through the results incrementing the appropriate
           #           in           the           results           table.
           #
           # while           ($results->next())
           {
           my           $user           =$results->text("AudUser");
           my           $table          =$results->text("AudTable");
           my           $operation      = $results->text("AudOperation");

           $data->{$user}->{$table}->{$operation}++;
           }
           $results->close();

           #
           # Create a statistics object we can use to insert
           #           statistics           records.
           #
           my           $stats          =           $session->statistics();
           $stats->setDate($yesterday);
           $stats->setDateFrom($yesterday);
           $stats->setDateTo($yesterday);
           $stats->setKey1("Audit Statistics by Operation by Module
by           User           ($period)");

           #
           # Now move through the results table adding the
appropriate           records
           #           to           the           statistics           table.
           #
           foreach           my           $user           (keys(%{$data}))
           {
           $stats->setKey4($user);
           foreach           my           $table          (keys(%{$data->{$user}}))
           {
           $stats->setKey3($table);
           foreach           my           $operation      (keys(%{$data-
>{$user}->{$table}}))
           {
           $stats->setKey2($operation);
           $stats->setValue($data->{$user}-
>{$table}->{$operation});
           $stats->write();
           }
           }
           }
           }
1;

```

The above task shows how the `KE::Statistics::Session` object can be used to obtain information about the Vitalware environment (list of registered users, etc.) and also query tables (eaudit table). For a complete description of all the methods available in the `KE::Statistics` perl module please see Appendix A (page 21).

Creating a new period

The Periodic Tasks facility is designed to be extensible: new periods can be added as required. In this section we will add a new period that generates statistical information on a quarterly basis. The steps required are:

1. Determine a name for the period, e.g. quarterly.
2. Create the directory in which the quarterly tasks will be stored, e.g. `local/etc/periodic/quarterly`.
3. Add an entry to cron so that **vwperiodic** is invoked at a suitable time. The entry for quarterly will look like:

```
0 7 1 1,4,7,11 * /home/vw/client/bin/vwrun vwperiodic
quarterly 2>&1 | /home/vw/client/bin/vwrun vwlogger -t "KE
Vitalware Periodic Tasks Report" periodic
```

4. Add the quarterly tasks to `local/etc/periodic/quarterly`.

Statistics generate on a quarterly basis. Note that the quarterly tasks are run at 7:00 am the day after the quarter ends.

Regenerate missing data

In some cases it may be necessary to generate statistic records for time periods that have passed, for instance periods before the Periodic Tasks facility was installed. It is possible to run **vwperiodic** using the `-d` option to specify the date passed through to the period tasks. In effect, the `-d` option makes it possible to alter the value of `$date` passed through to the `Periodic()` function. It is up to the task itself to examine the date and generate the correct information, where possible.

The date specified with the `-d` option should correspond to the date and time at which the original tasks would have been executed. For example, to run the daily tasks for 15 February 2009, the following command should be used:

```
vwperiodic -d 2009:02:16 daily
```

Notice how the date given was for the next day as this corresponds to the date on which cron would have invoked the daily tasks for 15 February 2009. By varying the date supplied it is possible to generate statistical information for periods before Periodic Tasks was installed. If a record already exists for the statistic generated, the value is simply updated.

The generation of data for previous time periods is successful only if the data for the period specified exists: it is not possible to generate auditing information if the audit records do not exist for the period specified.

SECTION 2

Appendix A - KE::Statistics perl module

The `KE::Statistics` module provides a set of objects to make the creation of tasks easier. The module is located in the `utils/KE` directory on the Vitalware server. The code is documented using POD (plain old documentation). The information in this Appendix was generated from the POD in the module.

Name

`KE::Statistics` - A set of objects usable by periodic scripts.

Synopsis

```

use                                                    KE::Statistics;

sub
Periodic
{
    my          $session      =          shift;
    my          $date         =          shift;
    my          $period       =          shift;

    my          $users        =          $session->users();
    my          $tables       =          $session->tables();
    my          $operations   =          $session->operations("registry");

    my $query = "Select all from registry where Key1 = " .
                $session->quote() . "User" . $session->quote();
    my          $results      =          $session->search($query);

    while                                             ($results->next())
    {
        my          $key      =          $results->text("Key1");
        ...
    }
    $results->close();

    my          $stats        =          $session->statistics();
    my          $yesterday   =          $date->yesterday();
    $stats->setDate($yesterday);
    $stats->setDateFrom($yesterday);
    $stats->setDateTo($yesterday);
    $stats->setKey1("Records By Table");

    $stats->setValue("3");
    $stats->write();
}

```

Description

The `KE::Statistics` module provides a set of objects to facilitate the generation of records for the estastistics table. The Periodic Tasks subsystem provides a plug-in mechanism that allows new tasks to be added to the existing framework. Each task is contained within a perl library (.pl file) and must contain at least one function, the `Periodic($session, $date, $period)` method.

The arguments are:

- | | |
|------------------------|---|
| <code>\$session</code> | A <code>KE::Statistics::Session</code> object provides a connection to the back-end database environment. The object may be used to gather information to generate statistical values and to create estastistics records. |
| <code>\$date</code> | A <code>KE::Statistics::Date</code> object contains the date and time at which the Periodic Tasks subsystem was invoked. The <code>\$date</code> object is used to determine the date/time range of the statistical information for the task invoked. |
| <code>\$period</code> | A string that contains the name of the time period for the task being run. Typical periods include <code>hourly</code> , <code>daily</code> , <code>weekly</code> and <code>monthly</code> . Administrators may create new periods (e.g. <code>fortnightly</code>) as required, in which case <code>\$period</code> will contain the name of the new period. |

The `Periodic()` function is called by the Periodic Tasks subsystem at a scheduled time (e.g. hourly, daily, weekly, monthly, etc.) to create records in the estastistics table.

The following objects are provided within the module:

KE::Statistics::Session

A `KE::Statistics::Session` object is used to gather information from the back-end server. The object may query any table or set of tables to allow statistical information to be generated. A set of methods allows information about the server environment to be gathered (e.g. list of registered users, list of tables, etc.).

As a `Session` object is provided as an argument to the `Periodic()` function, it is not necessary to create the object yourself, rather the supplied object should be used (which is efficient as only one `Session` object is used by all tasks invoked in the current execution). As the `Session` object is shared, you must not `close()` it in your task.

Methods

`new()`

```
$session = KE::Statistics::Session->new();
```

Creates a connection to the server environment. As the Periodic Tasks subsystem provides a `Session` object to your task, it is not necessary to use this method. The return value is an instance of a `Session` object.

`search($texql)`

```
$results = $session->search("count(select all from eparties)");
```

Executes a `TeXQL` query statement on the server. The `$texql` argument may be any valid `TeXQL` query statement. The return value is a `KE::Statistics::ResultSet` object. If the query statement is invalid, an `undef` value is returned.

`statistics()`

```
$stats = $session->statistics();
```

After your tasks have generated statistical information, it is necessary to write the data into `estatistics` records. The `KE::Statistics::Statistics` object provides a convenient object for creating `estatistics` records. The `statistics()` method returns a `Statistics` object that may be used to create the records.

`quote()`

```
$texql = "select all from eparties where NamLast contains " . $session->quote() . "Badenoff" . $session->quote();
```

When building `TeXQL` statements, non-numeric values must be enclosed within *quotes*. The quote character is configurable and is set to avoid escaping characters within values. The default quote character is `\001` (`Ctrl+A`). The `quote()` method returns the current quote character.

`close()`

```
$session->close();
```

Once all communication with the server environment is complete, the connection needs to be closed so that system resources can be returned to other users. The `close()` method terminates a `Session` connection. As the Periodic Tasks subsystem handles the creation and closing of the session, you should not call this method.

`users()`

```
foreach my $user (@{$session->users()})
```

The `users()` method returns a reference to a list of registered users in the server environment. The list is built from records in the server registry (`registry` table).

`tables()`

```
foreach my $table (@{$session->tables()})
```

The `tables()` method returns a reference to a list of tables in the server environment. The **Table Access Registry** entry is used to build the list of tables.

```
operations($table)
```

```
foreach my $operation (@{$session->operations("eparties")})
```

The `operations()` method returns a reference to a list of audit operations enabled for the table supplied in the `$table` argument. The list returned is populated by operations defined by `texaudit`. Use `texaudit -h` to get a complete list of the available operations.

KE::Statistics::ResultSet

A `KE::Statistics::ResultSet` object is returned by the `KE::Statistics::Session->search($texql)` (page 25) method. The object provides access to the records returned as a result of the specified query. Once you have finished dealing with the `ResultSet` object, it is necessary to `close()` it so that system resources can be returned to other users.

Methods

`new()`

```
$results = KE::Statistics::ResultSet->new($cursor)
```

A `ResultSet` object provides a convenient mechanism for dealing with a query cursor (`$cursor`). The cursor is returned by the server environment when a search has completed. As a `ResultSet` object is returned by `KE::Statistics::Session->search($texql)` (page 25), it is not necessary to create instances of `ResultSet` objects.

`next()`

```
while ($results->next())
```

When a `ResultSet` object is returned by `KE::Statistics::Session->search($texql)` (page 25), the current record is positioned before the first record. The `next()` method moves the current record position to the next matching record. A value of 0 is returned if you are past the last matching record, otherwise 1 is returned.

`text($column)`

```
$value = $results->text("NamLast");
```

The `text()` method returns the value for the column specified by `$column` argument. The value is returned as a string. If the column does not exist, an `undef` value is returned.

`close()`

```
$results->close();
```

Once you have finished with the records in the `ResultSet` object, you should `close()` (page 25) the object so that server resources are returned to users. If you do not close a `ResultSet` object, it will be closed by the Periodic Tasks subsystem once it has completed processing all tasks.

KE::Statistics::Date

In order to make the manipulation of dates easier, the `KE::Statistics::Date` object is provided. The object contains a breakdown of a date (`{year}`, `{month}`, `{day}`, `{hour}`, `{minute}` and `{second}`). A number of methods are provided that allow the date/time to be manipulated.

To help with arithmetic manipulation of dates the `julianNumber()` (page 30) method is provided to return the Julian date (see http://en.wikipedia.org/wiki/Julian_day). The integer part of the floating point number returned represents the day number, while the fractional part encodes the time within the day. Normal arithmetic may be applied to the number. The `julianDate()` (page 30) method is used to convert a Julian date to a `Date` (page 33) object. For example, the following code could be used to find the date three days back from today:

```
$now = KE::Statistics::Date->new();
$then = KE::Statistics::Date->julianDate($now->julianNumber() - 3);
```

Subtracting two Julian dates will result in the number of days, hours, minutes and seconds between them:

```
$diff = $now - $then;
```

When using dates with `TeXQL` query statements, always specify the date in ODBC format (`yyyy-mm-dd`). The `dateText()` (page 30) method provides the value in the correct format. Similarly, time values should be specified using a 24 hour clock (`HH:MM:SS`). The `timeText()` (page 30) method provides the value formatted correctly.

Methods

```
new($year, $month, $day, $hour, $minutes, $seconds)
```

```
$date = KE::Statistics::Date->new();  
$date = KE::Statistics::Date->new(2009, 02, 11);  
$date = KE::Statistics::Date->new(2009, 02, 11, 16, 55, 02);
```

The `new()` (page 25) method creates a new instance of a `Date` (page 33) object. Up to six arguments may be provided to initialise the `Date` object with a given date and/or time. If any arguments are missing, the component for the current date/time is used. Thus, calling `new()` without any arguments provides a `Date` object with the current date and time.

```
clone()
```

```
$newdate = $date->clone();
```

The `clone()` method creates a copy of a `Date` object initialised with the same date/time as the calling `Date` object.

```
yesterday()
```

```
$yesterday = $date->yesterday();
```

Returns a new `Date` object initialised with yesterday's date. The value is 24 hours before the calling `Date` object; that is, the time component is not changed.

```
lastHour()
```

```
$newdate = $date->lastHour();
```

Returns a new `Date` object initialised with the date/time one hour before the date/time of the calling `Date` object.

```
lastSecond()
```

```
$newdate = $date->lastSecond();
```

Returns a new `Date` object initialised with the date/time one second before the date/time of the calling `Date` object.

```
lastWeek()
```

```
$newdate = $date->lastWeek();
```

Returns a new `Date` object initialised with the date/time one week before the date/time of the calling `Date` object.

```
lastMonth()
```

```
$newdate = $date->lastMonth();
```

Returns a new `Date` object initialised with the date/time one month before the date/time of the calling `Date` object. If the resulting date is past the end of the month, the last day of the month is used.

```
set($year, $month, $day, $hour, $minute, $second)
```

```
$date->set(2010, 12, 14); $date->set(undef, undef, undef, 0, 0, 0);
```

The `set()` method allows any component of a `Date` object to be assigned a value. If `undef` is provided for a component, the component's current value is maintained. If a component is missing, a value of `undef` is assumed.

```
compare($date)
```

```
if ($date1->compare($date2) == 0)
```

The `compare()` method compares two `Date` objects for equality. The return value can be used to determine the equality of the objects:

```
-1 - date argument is lower than date object
0  - date argument is same as Date object
+1 - date argument is greater than Date object
```

```
compareDate($date)
```

```
if ($date1->compareDate($date2) == 0)
```

The `compareDate()` method compares two `Date` objects for equality at the date level. The time component is ignored. The return value can be used to determine the equality of the object's dates:

```
-1 - date argument is lower than date object
0  - date argument is same as date object
+1 - date argument is greater than Date object
```

```
compareTime($date)
```

```
if ($date1->compareTime($date2) == 0)
```

The `compareTime()` method compares two `Date` objects for equality at the time level. The date component is ignored. The return value can be used to determine the equality of the object's times:

```
-1 - date argument is lower than date object
0  - date argument is same as date object
+1 - date argument is greater than date object
```

```
dateText()
```

```
$texql = "select all from eaudit where AudDate = DATE" .
    $session->quote() . $date->dateText() . $session->quote();
```

The `dateText()` method returns a text representation of the object's date in ODBC format (`yyyy-mm-dd`). The value is suitable for `DATE` values in `TexQL` queries regardless of the date format used on the server.

```
timeText()
```

```
$texql = "select all from eaudit where AudTime = TIME" .
    $session->quote() . $date->timeText() . $session->quote();
```

The `timeText()` method returns a text representation of the object's time in ODBC format (`HH:MM:SS`). The value is suitable for `TIME` values in `TexQL` queries regardless of the time format used on the server.

```
julianNumber($date)
```

```
$julian = $date->julianNumber();
```

The return value of `julianNumber()` is a floating point number representing the day number in the integer part and the time (in 1/86400th of a second) in the fractional part. Note that the Julian number for a day represents midday for the given day. Any time before midday will have an integer value one less than any time after midday. To

get the Julian number for any time within a day it is necessary to add 0.5 before calling `int()`. Thus:

```
$daynumber = int($date->julianNumber() + 0.5);
```

returns the Julian day number. See http://en.wikipedia.org/wiki/Julian_day for details.

```
julianDate($number)
```

```
$date = KE::Statistics::Date->julianDate($number);
```

A `Date` object is returned containing the date and time expressed by the Julian date number passed as an argument. The `julianDate()` method provides a mechanism for getting a `Date` object after some date numeric arithmetic has been performed.

```
weekDay()
```

```
$day = ("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")[$date->weekDay()];
```

Returns the numeric day of the week for the given `Date` object, where 0 = Sunday, 6 = Saturday.

KE::Statistics::Statistics

The `KE::Statistics::Statistics` object is designed to provide easy insertions into the `eststatistics` table. A `Statistics` object allows the columns within a record to be set and the record written. A check is made to see if the record already exists in the table and if so an update is performed rather than an insertion. This allows periodic tasks to be re-run to refresh data without duplicate records being created.

An `eststatistics` record consists of four main components:

Keys A hierarchy of up to ten Keys may be specified. The Keys are used to define the variables used to arrive at the statistical value. The first Key should contain a title defining what the statistical value is. For example:

```
Key1: Audit Statistics by Operation by Module by User
      (daily)
Key2:                                     delete
Key3:                                     ebirths
Key4: bill
```

allows you to determine from `Key1` what information is being stored. The next three Keys are the variables (*Operation*, *Module*, *User*) available. The above convention should be used so users can easily locate records within the `eststatistics` table.

Date Three date fields exist in `eststatistics`: *DateExact*, *DateFrom* and *DateTo*. The *DateExact* field is filled if the statistical value represents a period of a day or less (that is daily or hourly), otherwise it is left empty. The *DateFrom* and *DateTo* fields should always be filled with the commencement and completion dates respectively.

Time Three time fields exist in `eststatistics`: *TimeExact*, *TimeFrom* and *TimeTo*. The *TimeExact* field is filled if the statistical value represents a single point of time in a day, otherwise it is left empty. If the period is a range of time in a day, the *TimeFrom* and *TimeTo* fields should be filled with the commencement and completion times respectively.

Value The *Value* is the statistical datum associated with the set of defined Keys for the given date and/or time. For example, a *Value* of 10 with the above Keys would indicate user `bill` has deleted 10 location records for the specified date/time period.

When completing an `eststatistics` record, the appropriate fields should be filled based on the period the value covers.

Methods

```
new($session)
```

```
    $stats = KE::Statistics::Statistics->new($session);
```

A new instance of a `Statistics` object tied to the supplied `Session` (`$session` (page 23)) is created. You should not create instances of a `Statistics` object directly, rather `$session->statistics()` (page 25) should be used as this ties the created object to the session.

```
setDate($date)
```

```
    $stats->setDate($date);
```

Sets the *DateExact* column in *estaticistics* to the value of the `Date` object supplied. The *DateExact* column should be filled if the statistic record is for a particular day (that is, daily) or a time range within a day (that is, hourly).

```
setDateFrom($date)
```

```
    $stats->setDateFrom($date);
```

Sets the *DateFrom* column in *estaticistics* to the value of the `Date` object supplied. The *DateFrom* column should always be filled. It contains the starting date for the statistics period.

```
setDateTo($date)
```

```
    $stats->setDateTo($date);
```

Sets the *DateTo* column in *estaticistics* to the value of the `Date` object supplied. The *DateTo* column should always be filled. It contains the finishing date for the statistics period.

```
setTime($date)
```

```
    $stats->setTime($date);
```

Sets the *TimeExact* column in *estaticistics* to the value of the `Date` object supplied. The *TimeExact* column should only be filled if the statistic record is for a single point in time, otherwise the column should be left empty.

```
setTimeFrom($date)
```

```
    $stats->setTimeFrom($date);
```

Sets the *TimeFrom* column in *estaticistics* to the value of the `Date` object supplied. The *TimeFrom* column contains the starting time for the statistics period. It should only be filled for statistic records for a single point in time, or a time range (that is hourly).

```
setTimeTo($date)
```

```
    $stats->setTimeTo($date);
```

Sets the *TimeTo* column in *estaticistics* to the value of the `Date` object supplied. The *TimeTo* column contains the completion time for the statistics period. It should only be filled for statistic records for a single point in time, or a time range (that is hourly).

```
setKey1($value)
```

```
    $stats->setKey1($value);
```

Sets the *Key1* column in *estaticistics* to the value supplied.

```
setKey2($value)
```

```
    $stats->setKey2($value);
```

Sets the *Key2* column in estatistics to the value supplied.

```
setKey3($value)
```

```
    $stats->setKey3($value);
```

Sets the *Key3* column in estatistics to the value supplied.

```
setKey4($value)
```

```
    $stats->setKey4($value);
```

Sets the *Key4* column in estatistics to the value supplied.

```
setKey5($value)
```

```
    $stats->setKey5($value);
```

Sets the *Key5* column in estatistics to the value supplied.

```
setKey6($value)
```

```
    $stats->setKey6($value);
```

Sets the *Key6* column in estatistics to the value supplied.

```
setKey7($value)
```

```
    $stats->setKey7($value);
```

Sets the *Key7* column in estatistics to the value supplied.

```
setKey8($value)
```

```
    $stats->setKey8($value);
```

Sets the *Key8* column in estatistics to the value supplied.

```
setKey9($value)
```

```
    $stats->setKey9($value);
```

Sets the *Key9* column in estatistics to the value supplied.

```
setKey10($value)
```

```
    $stats->setKey10($value);
```

Sets the *Key10* column in estatistics to the value supplied.

```
setValue($value)
```

```
    $stats->setValue($value);
```

Sets the *Value* column in estatistics to the value supplied. The statistical value is a floating point number.

```
write()
```

The `write()` method saves the data in the `Statistics` object to the estatistics table. If a record already exists with the same Keys, dates and times, the value is updated, otherwise a new record is created.

Bugs

Encoding dates as a Julian number with the time as the fractional component can lead to issues when subtracting dates, as the result represents the number of days, hours, minutes and seconds between the two dates. If you need to find the number of days between two dates, it is necessary to clear the time component before applying the subtraction:

```
$date1->set(undef, undef, undef, 0, 0, 0);  
$date2->set(undef, undef, undef, 0, 0, 0);  
$days = abs($date2->julianNumber() - $date1->julianNumber());
```

While this not a bug, it is something to keep in mind when manipulating Julian date numbers.

See Also

For a complete description of how Julian date numbers are generated and used see:

http://en.wikipedia.org/wiki/Julian_day

Vitalware Documentation

Record Recall

Document Version 1.0

Vitalware Version 2.1



Contents

SECTION 1	Record Recall	3
	Overview	3
	Recall Single Record	4
	Recall batch mode	7
	Registry Entry	10

Record Recall

- [Overview](#)
- [Recall single record](#)
- [Recall batch mode](#)
- [Registry entry](#)

Overview

Have you ever changed a record and wished later that you could change it back to how it was before saving it? Have you ever performed a global replace only to find the changes were not exactly what you had in mind? Wouldn't it be nice if Vitalware provided a facility that allowed you to recall the data for a single record or group of records to an earlier version? Well, the new **Record Recall** facility introduced in KE Vitalware 2.1.01 provides this functionality.

KE Vitalware 2.0.03 saw the introduction of a new auditing facility. The facility maintains a complete list of changes made to records within a Vitalware installation. The new **Audit Trails** module allows users to view these changes along with the associated metadata. This metadata includes:

- Date modified
- Time modified
- Who made the changes

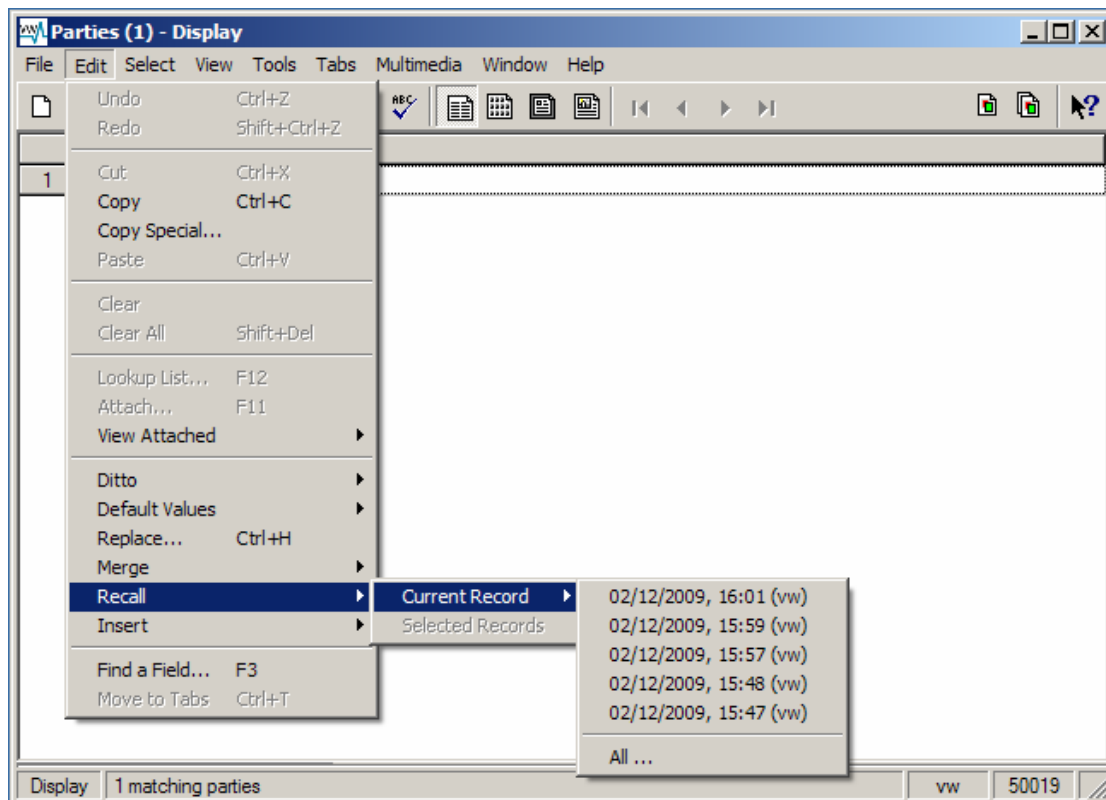
One use of the audit information is to provide statistical analysis about changes to data within an Vitalware environment. It is possible to use this data to produce reports about the number of records inserted, modified and deleted on a system wide or per user basis.

A list of all modifications made each time a record is saved is produced as a record is changed over time. The **Record Recall** facility provides an automated way of applying changes to a record so that it looks like an earlier version. In essence it allows the modifications made to be "undone" so that the record appears as it did at an earlier time, although it is important to understand that the record is not reset to the earlier version: rather the record is modified so that it has the same data as the earlier version. The distinction here is that the "recalled" record is just another change to the current record, not a winding back to a previous version. As such, all existing audit trail records are maintained and a new one is created for the "recalled" record.

The **Record Recall** facility allows a single record to be recalled to the data it contained in a previous version and also provides a batch mode for a group of (one or more) records to be recalled to their state at a given date and time.

Recall Single Record

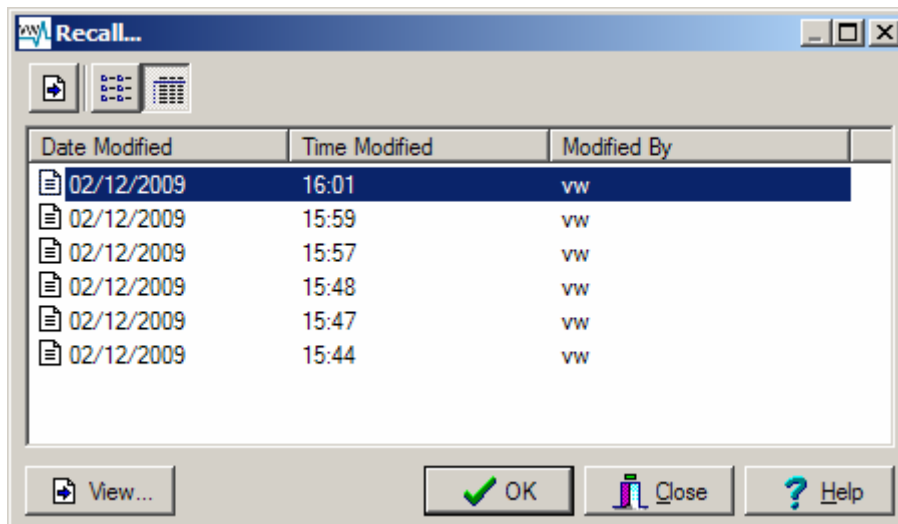
The **Record Recall** facility is invoked from the **Edit>Recall** menu. It may be selected while viewing or editing a record. If you want to recall the data for a single record, the **Current Record** sub-menu is used:



(All dates are shown in dd/mm/yyyy format. Vitalware will use the date format defined on your server when displaying dates)

The **Current Record** sub-menu lists the date and time of the last five modifications of the current record, as well as the user who made the changes. If you select one of these entries, the current view is changed to Detail mode and the data is updated to reflect the values in the record as at the date and time selected. The record is left in Edit mode and you may save it to affect the recall, or cancel the changes.

If there are more than five modifications, you can use the **All...** menu entry to view a list of all the dates and times:



You may select any entry from the list and click **OK** to recall that version of the data. Once again the record will be updated to contain the data as it was at the date and time selected. You will be left in Edit mode, allowing you to save or discard the changes. The **View** button invokes the **Audit Trail** module with the update audit records for the current record displayed. Note that only the audit records related to data changes will be shown; other audit records (e.g. insertion, searching) are not shown.

Once a date and time is selected all modifications to the current record from the most recent to the entry selected are "undone". For example, if you selected 02/12/2009 15:57 (vw) from the list above, the data changed by the following records would be reversed in the order listed:

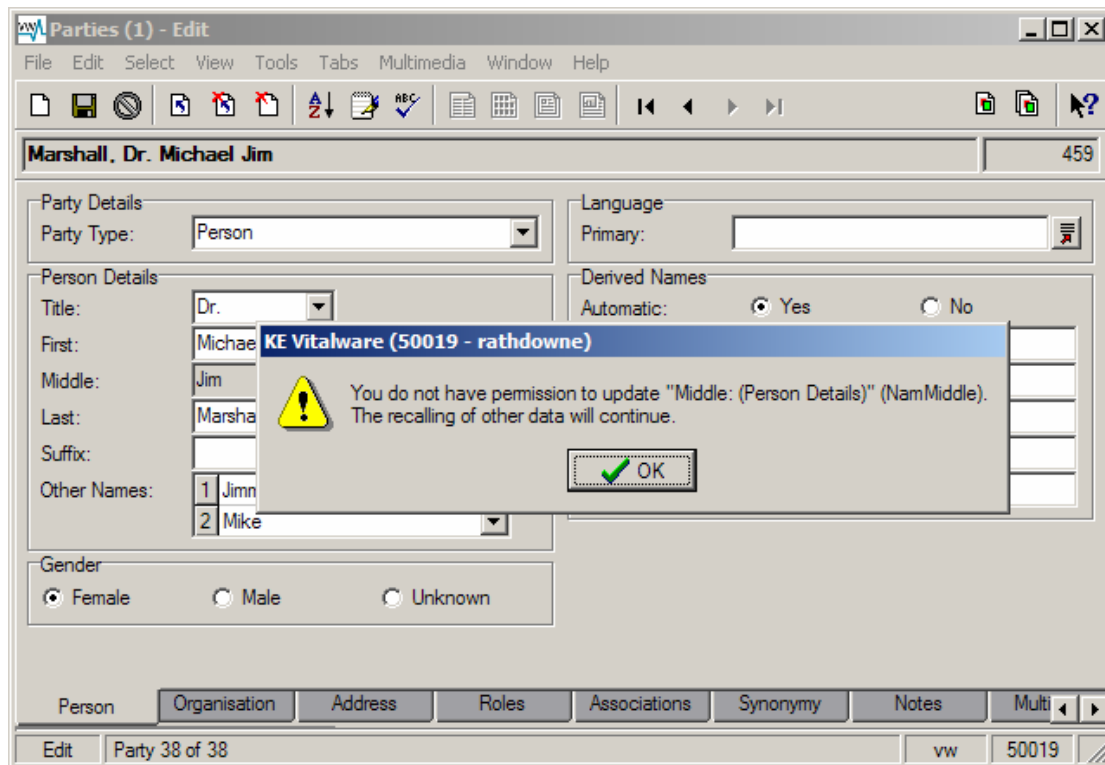
- 02/12/2009 16:01 (vw)
- 02/12/2009 15:59 (vw)
- 02/12/2009 15:57 (vw)

Once again you will be left in Edit mode so you may save or discard the changes.

While you are in Edit mode, recalling a previous version of the record will only update fields that have data to be recalled within the specified period; all other fields are not affected. For example, if you have just added data to fields for the first time, that data will not be removed by recalling an earlier version of the record.

It is possible to recall the data for a record even if you did not make the changes to the record. However, Vitalware will enforce both field and Record Level Security when applying a recall. If you do not have edit privileges for the current record, either through Record Level Security or a lack of the **daEdit** privilege, you cannot select the **Recall** menu option.

If the data recall would result in a field being changed that you do not have permission to update (missing **duEdit** privilege), a message similar to the following will display:

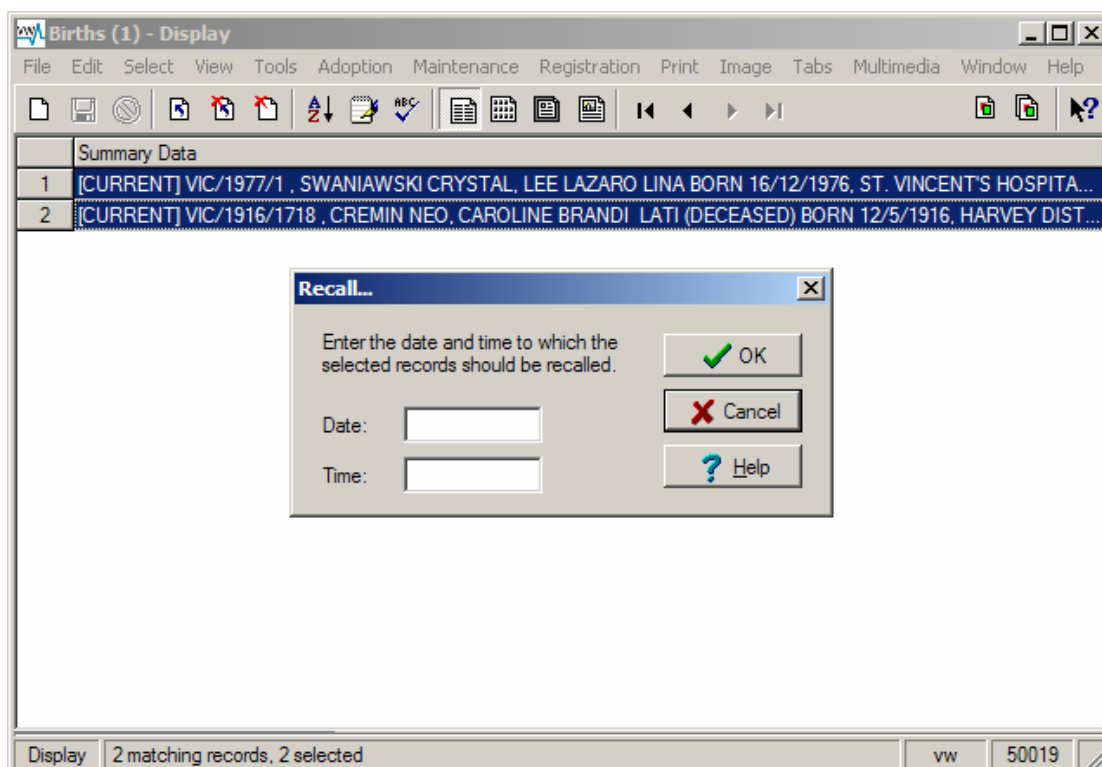


The recall will continue to update fields once **OK** is clicked. In this case the record will not contain an exact copy of the data as it was at the date and time selected. Vitalware will leave you in Edit mode and you may save or discard the recalled data.

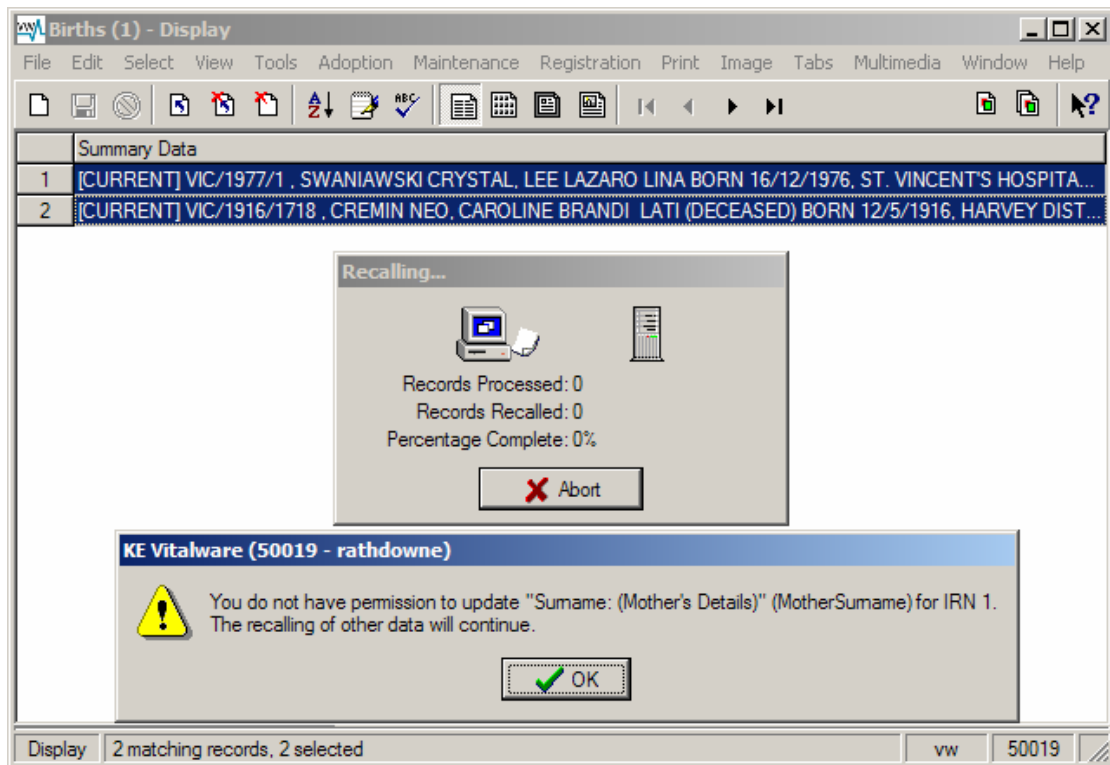
Recall batch mode

It is also possible to recall a group of (one or more) records as a single batch. In this case the records are updated and saved without the need to recall each record individually. The **Edit>Recall>Selected Records** menu option is available when one or more records are selected and you have the **daReplace** privilege (that is, you are allowed to use the **Replace** command).

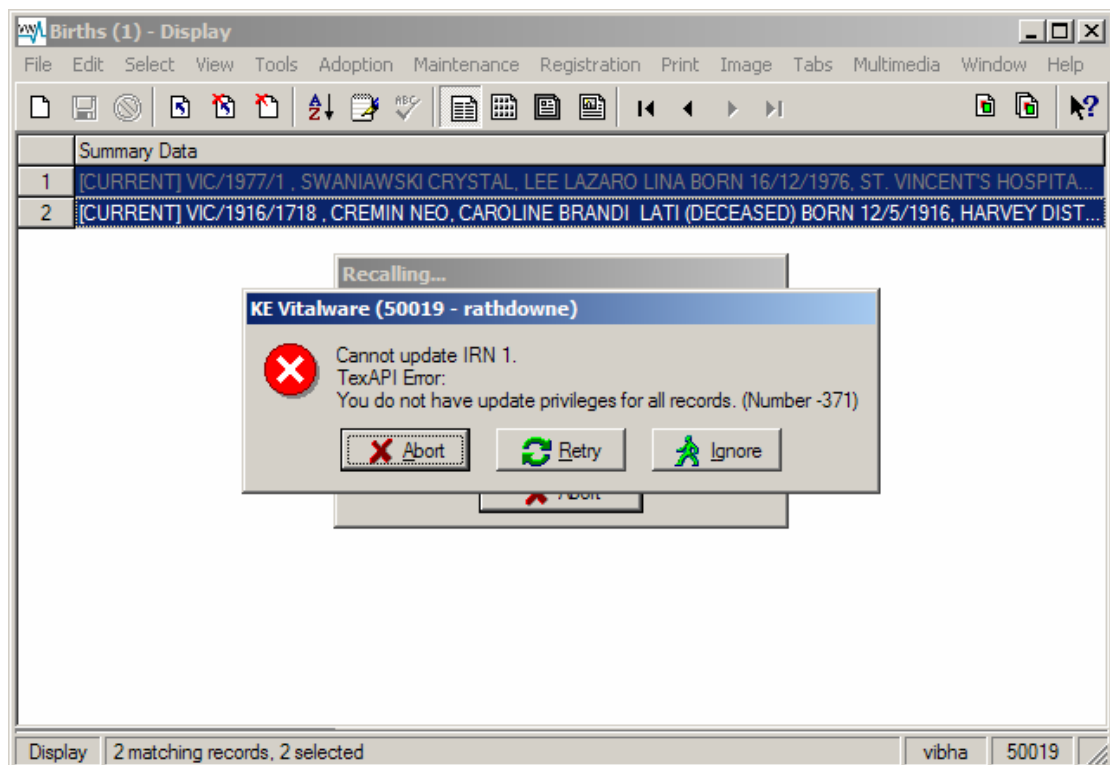
When the menu option is invoked you will be prompted for the date and time to which the selected records will be recalled:



Once you have supplied a date and time and clicked **OK**, Vitalware will begin recalling each record selected and applying an update to reflect the values in the record at the date and time specified. As with recalling a single record, you must have permission to modify the fields that are recalled. If not, the process will stop and a similar message to that for a single recall is displayed:



You must also have permission to update the record itself (via Record Level Security). If you do not have permission to modify a record, an error is displayed:



You can decide to cancel or continue the recall. Once the operation is complete a summary dialogue is displayed:



After the batch recall has finished, the selected records will be updated with the recalled values.

Registry Entry

The **Recall Record** facility uses a Registry entry to determine which fields should not be updated when applying a recall. The format of the entry is similar to that for the *Ditto* entry:

```
System|Setting|Table|table|Recall Skip  
Columns|column;column;...
```

where:

```
table is the name of the module in which the columns are  
to be skipped; and  
column;column;... is a semi-colon separated list of  
column names.
```

Vitalware does not recall the *Date Modified*, *Time Modified* and *Modified By* fields when updating a record (as these fields contain the data concerning the person performing the recall operation).

Record Templates

KE Vitalware Documentation

Document Version 1.0

KE Vitalware Version 2.1



Contents

Record Templates	1
How to create records using Record Templates	3
How to create a Record Template	12
How to define a Record Template	15
Atomic Fields	17
Nested Table Fields	17
Double Nested Table	18
An example record	19
template tag	20
source tag	20
input tag	21
prompt tag	22
help tag	22
value tag	23
records tag	23
number tag	23
report tag	23
column tag	24
Some example Record Templates	25
Example 1	25
Example 2	27

SECTION 1

Record Templates

Vitalware has provided a Ditto utility for a long time. The Ditto utility allows users to extract information from an existing record and copy it into one being added. Using Ditto it is possible to copy data from:

- One field in a record into the same field in another record.
- A tab in a record into the same tab in another record.
- All fields in a record into another record.

While this facility is useful for adding similar records, it does have some shortcomings:

- Only a single record can be added at a time.
- Only single fields, tabs or the entire record can be extracted.
- Incrementing numbers, such as Registration numbers, must be entered for each record.
- A series of records with consecutive numbers is difficult to create.

Vitalware also provides a Default Values facility that allows one or more fields to be initialised with values when adding a new record. Users can define a number of Default Values templates and select one to be used when the next insertion is initiated. While Default Values can be defined for any field, it is not possible to extract data from existing records.

KE Vitalware 2.1.01 sees the introduction of the Record Template utility, which combines the functionality of the Ditto and Default Values facilities:

- A number of records can be created in a batch and added to a set of one or more currently listed records.
- An optional starting IRN can be specified, allowing consecutive IRNs to be allocated.
- Data can be extracted from the current (source) record and added to new records. Data may be mapped from one field in the source record to another field in the created record.
- A starting number and incrementation can be specified, allowing a range of consecutive values to be allocated.
- A wizard is provided to walk through the process.
- A report is generated listing the IRN and incrementing numbers allocated for each record created.
- An XML based template description is used to specify what data is placed in created records.

Some useful applications of the Record Template utility include:

- Creation of Part records for an existing object record.
- Insertion of preparation records for a specimen record (e.g. tissue samples).
- Reserving a block of IRNs or Registration numbers.

In the next section we will look at how to use the new facility, followed by how to set up your own templates.

How to create records using Record Templates

The Record Template utility can be used to create a batch of records based on an existing record.

1. Search for or otherwise list a group of records.

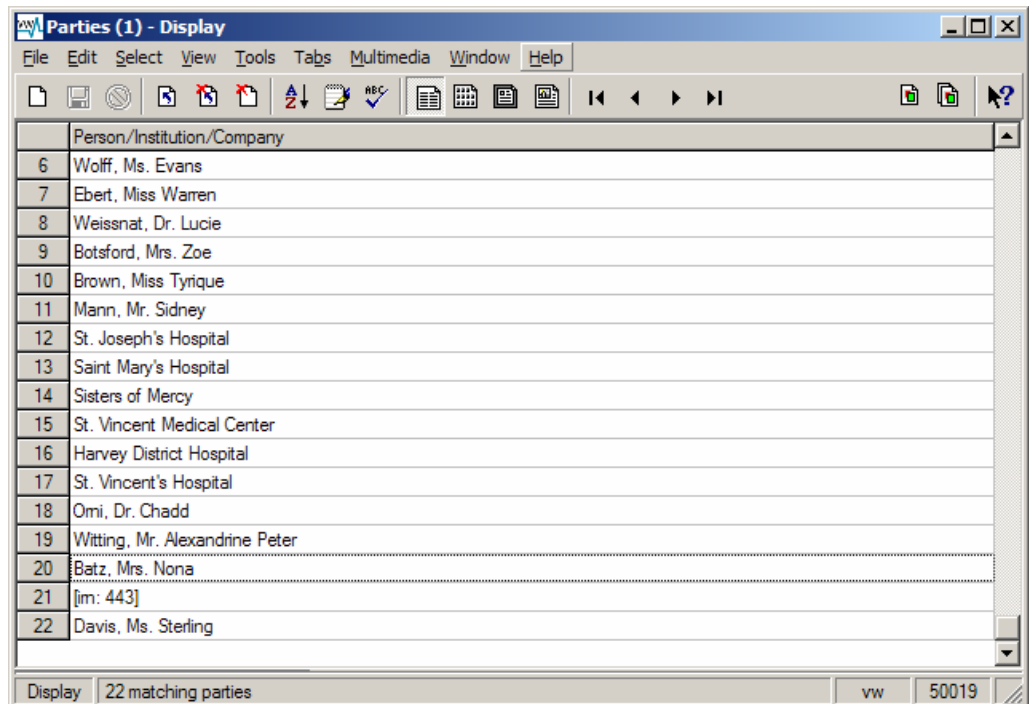
As the purpose of the Record Template utility is to create a series of records based on an existing record, the first step is to retrieve one or more records. Any means may be used to retrieve the record to be used as the *source* record (the record from which values are to be extracted).

2. Make the *source* record the current record.

The current record is identified differently depending on the display view:

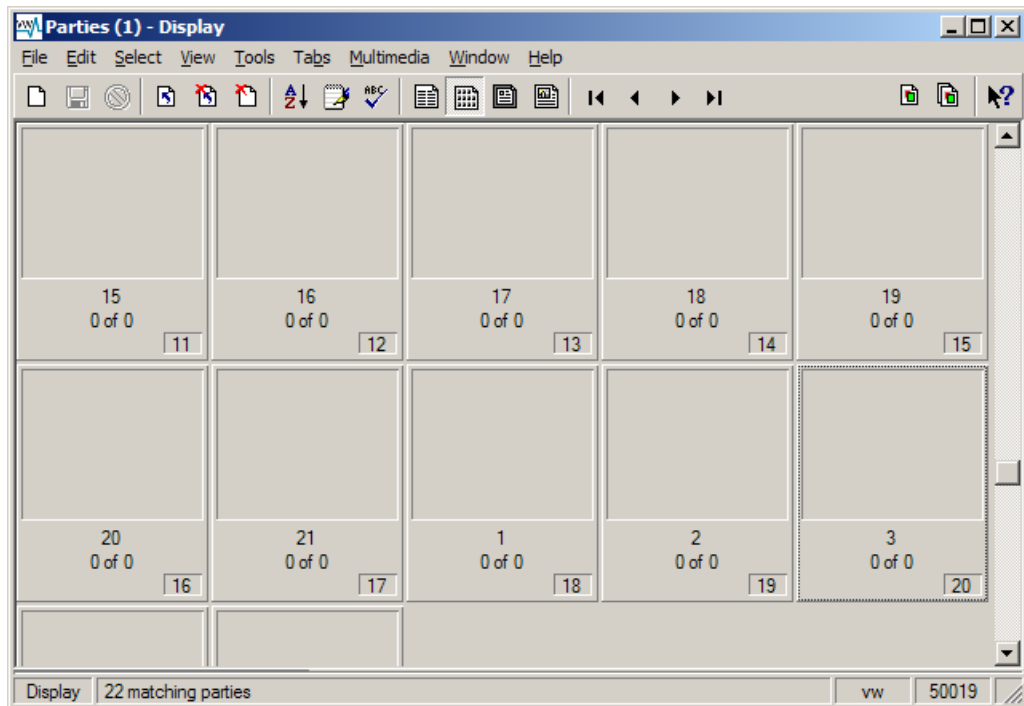
List View

The current record is enclosed in a dotted rectangle. In this example, record number 203 is the current record:



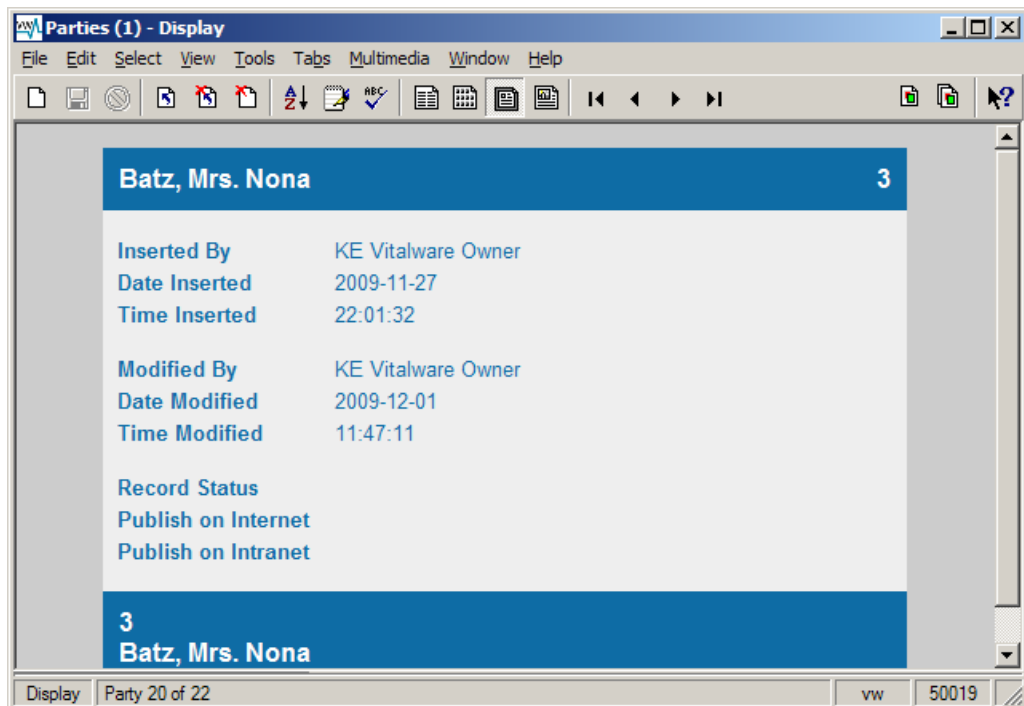
Contact Sheet View

A dotted rectangle appears around the image and label of the current record. In this example, image number 203 is the current record:



Page View

The current record is the displayed record:



Details View

The current record is the displayed record:

The screenshot shows a software window titled 'Parties (1) - Display'. The main content area is divided into several sections:

- Party Details:** Party Type: Marriage Celebrant
- Person Details:** Title: Mrs., First: Nona, Middle: (empty), Last: Batz, Suffix: (empty), Other Names: *
- Language:** Primary: (empty)
- Derived Names:** Automatic: Yes (selected), No; Salutation: Nona; Full: Mrs. Nona Batz; Brief: N. Batz; Cited: Batz, Nona; Taxonomic: Batz
- Gender:** Female (selected), Male, Unknown

At the bottom, there are tabs for 'Person', 'Organisation', 'Address', 'Roles', 'Associations', 'Synonymy', 'Celebrant', and 'Nona'. The status bar shows 'Display Party 20 of 22', 'vw', and '50019'.

3. Select **Tools>Templates** in the Menu bar

-OR-

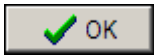
Use the keyboard shortcut, ALT+T+M.

The Record Templates box displays with a list of pre-defined Record Templates (if any):

The screenshot shows a dialog box titled 'Record Templates...'. It contains a table with the following data:

Title	Owner
Create Celebrant records	vw

At the bottom of the dialog, there are four buttons: 'New...', 'OK', 'Close', and 'Help'.

4. Select a Record Template from the list and select **OK** .

The Record Template Wizard displays:

Record Templates
Choose how many records to create.

Select the number of records to create and a starting IRN and click Next, or accept the defaults and click Next.

Number of records to create:

Starting IRN: (empty = next available IRN)

◀ Back Next ▶ Close Help

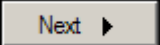
Specify the following:

- **Number of records to create**

Enter the number of records to be created when the Wizard is completed. There may be a maximum number of records that may be created: the limit is set by the creator of the Record Template. If a limit has been set, a hint will appear next to the box into which the record count is entered (as above).

- **Starting IRN**

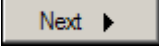
Enter the IRN (Internal Record Number) of the first record to be created. Subsequent numbers will be given to each new record. If a starting IRN is not specified, the next available number will be used. A Template creator may choose to hide this setting, in which case the next available number is always used.

5. Select **Next**  to continue.

The Input Values screen displays:

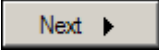
Input Values screens are where the data to be inserted into the new records is specified. The exact layout of the screen will vary depending on what information the Template creator wants to use. The screen has four elements:

- **Input field**
This is the Vitalware field into which the entered value will be added. The text displayed is the prompt of the field followed in brackets by the field group in which it appears.
- **Input prompt**
To the left of the input text box a prompt can be found indicating the type of value to be entered (*Enter a series title* in the example above). The creator of the Template specifies what prompt is displayed.
- **Input text box**
Enter the value to be inserted in the record created. The input text box may be single or multi-lined as defined by the Template creator. The creator may also indicate what type of data should be entered (text, integer or float) and whether a value is mandatory.
- **Input help**
This is a help message specified by the Template creator and designed to provide more information about, and examples of, the input value (*Series title* in the picture above). The input help is optional and will not appear on the Input Values screen if not defined by the Template creator.

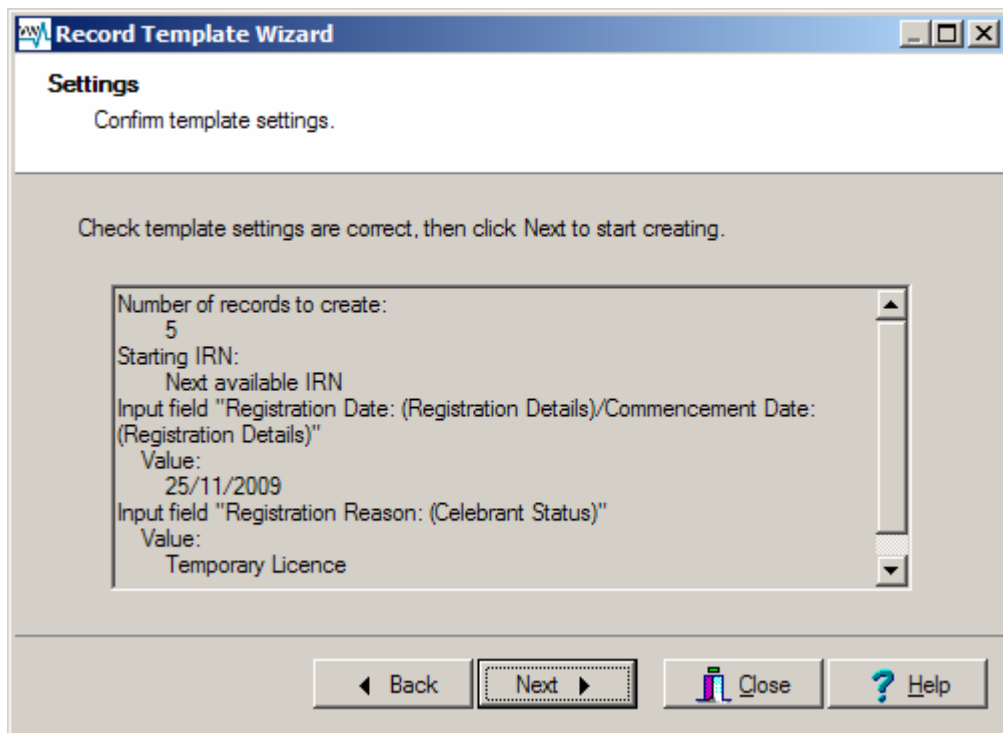
6. Select **Next**  to continue.

A number of Input Value screens may be displayed depending on how many input values are required for the created records. Input values may also be used to request the starting value for fields that contain incrementing data.

For example, a *Registration Number* may consist of the current year followed by a number within the year (e.g. 2008.23, 2006.154, etc.). An input value may ask for the year on one screen and the starting number within the year on the next screen. When the records are created, numbers will be allocated sequentially from the starting number for the supplied year. Using this mechanism it is easy to pre-allocate a batch of *Registration Numbers* to a set of records.

7. Select **Next**  to move through all the Input Value screens.

The Settings screen displays:



Record Template Wizard

Settings
Confirm template settings.



Check template settings are correct, then click Next to start creating.

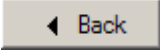
Number of records to create:
5


Starting IRN:
Next available IRN

Input field "Registration Date: (Registration Details)/Commencement Date:
(Registration Details)"
Value:
25/11/2009

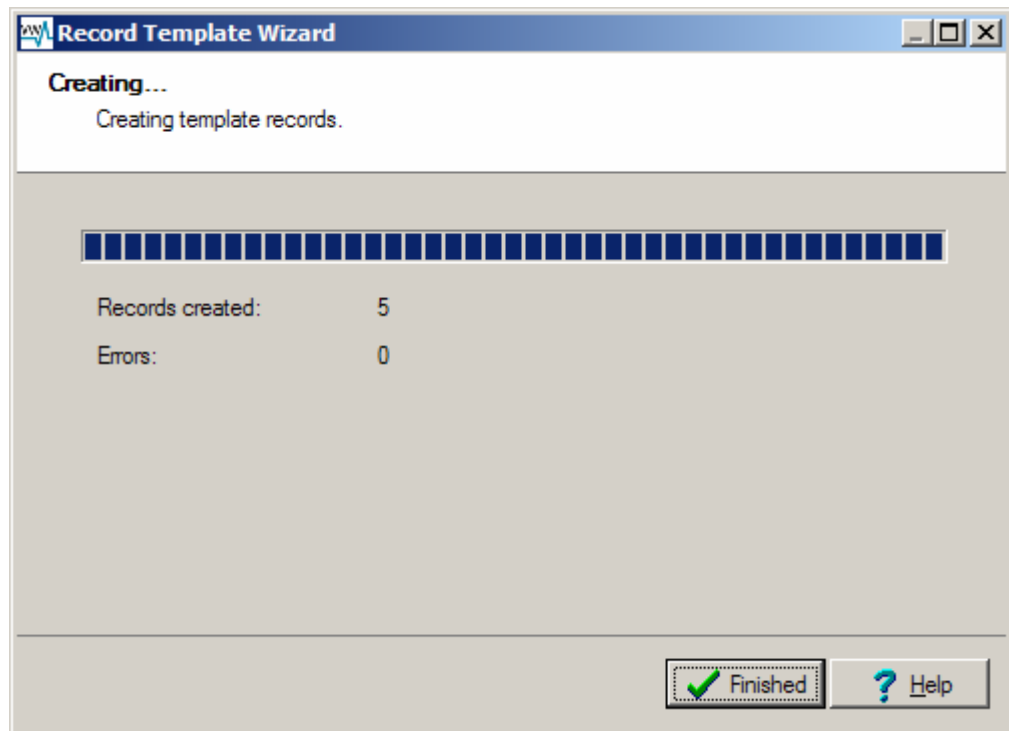
Input field "Registration Reason: (Celebrant Status)"
Value:
Temporary Licence

◀ Back Next ▶  Close  Help

This is a summary of the number of records to be created, the starting IRN and any input values. Select **Back**  if necessary to amend any details.

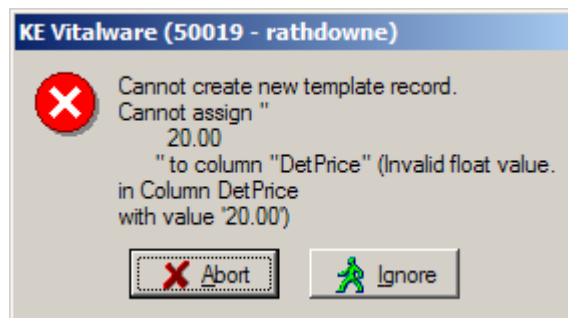
8. Select **Next**  to continue.


The Creating screen displays:



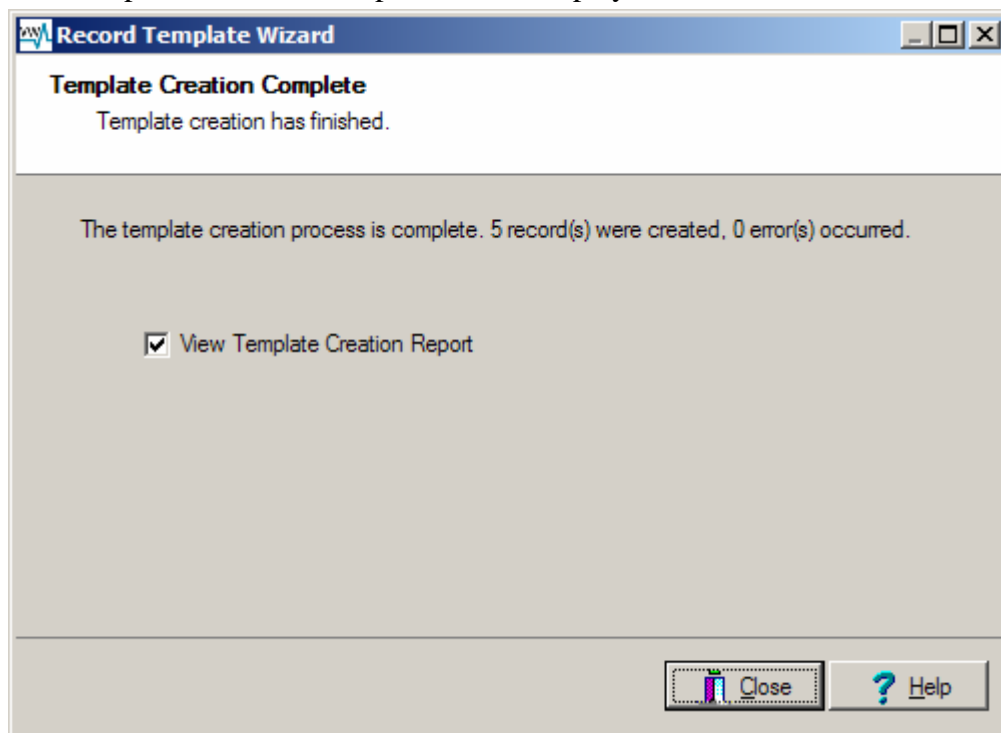
The records are now created. A gauge provides a visual indicator of creation progress. The numbers of records created and errors encountered are displayed.

If an error occurs, the creation process is halted and a message displays:

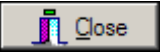


- i. Select **Abort** to end the creation process without further records being created
 - OR-
 - ii. Select **Ignore** to move on to the creation of the next record.
9. Select **Finish**  once the records are created.

The Template Creation Complete screen displays:



Specify the following:

- **View Template Creation Report**
Select whether a report listing the records created will be shown.
- Select **Close**  to finish the creation process. If **View Template Creation Report** was selected, a report displays:

```
Template record creation started 01 Dec 2009 14:46:00
Number of records to create: 5
Starting IRN: Next available IRN
Input field "Registration Date: (Registration
Details)/Commencement Date: (Registration Details)"
  Value: 25/11/2009
Input field "Registration Reason: (Celebrant Status)"
  Value: Temporary Licence

Record 1, created (irn: 453)
Record 2, created (irn: 454)
Record 3, created (irn: 455)
Record 4, created (irn: 456)
Record 5, created (irn: 457)

Number of errors: 0
Number of records created: 5
Template record creation finished 01 Dec 2009 14:46:13
```

The entry for each record created may vary as the Template creator may include data from the created records (e.g. *Registration Number*).

The records created are added to the records currently displayed. They are placed immediately after the current record, so that moving forward one record will display the first of the new records.

How to create a Record Template

The creation of a Record Template requires producing an XML description that details:

- Which fields are to be copied from the source record.
- Which input values are to be specified when the Template is used.
- The maximum number of records to be created.
- Whether a starting IRN may be specified.
- What data to show for each record created in the Template report.

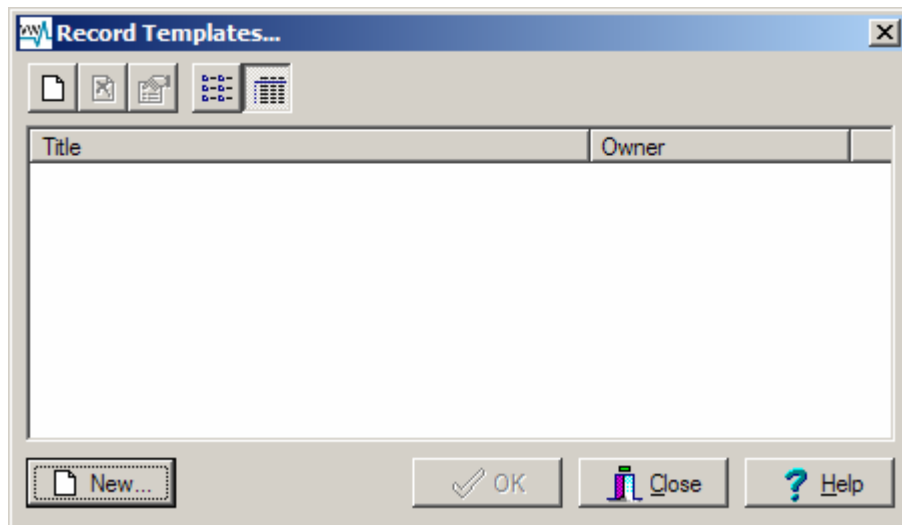
To be able to create a Record Template in a module a user must have (or be a member of a group that has) the *daTemplates* permission set for that module.

1. Search for or otherwise list a group of records.
2. Select **Tools>Templates** in the Menu bar

-OR-

Use the keyboard shortcut, ALT+T+M.

The Record Templates box displays with a list of pre-defined Record Templates (if any):



3. Select **New** 

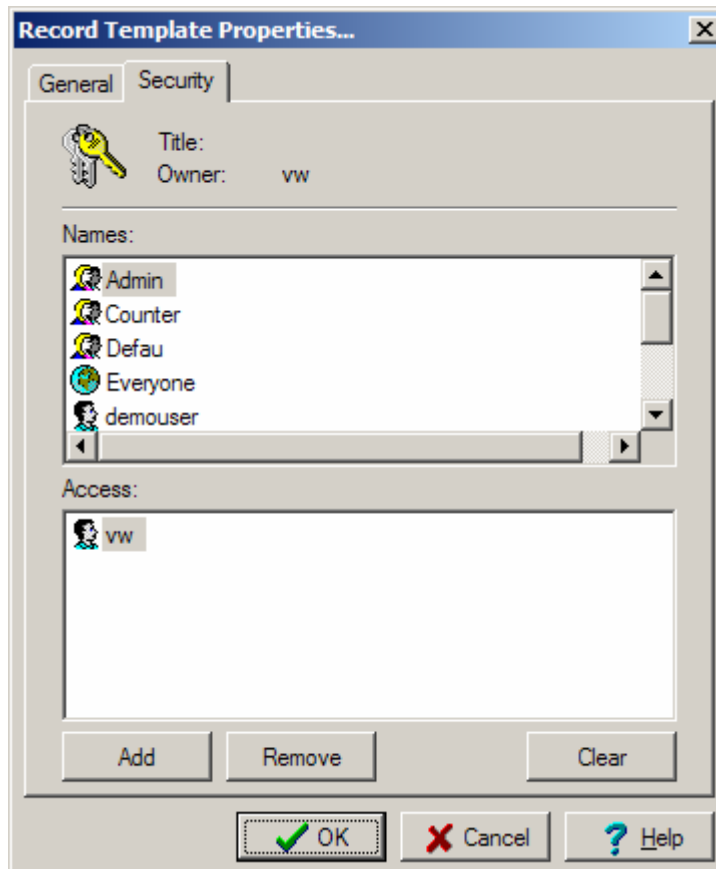
The Record Template Properties box displays:


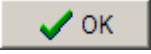
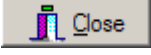
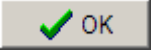


4. Enter a descriptive name for the Record Template in the top text field.
5. In the Record Template XML area enter the XML description for the Record Template.

The XML format is covered in detail in [How to define a Record Template](#) (page 15).

6. If required, select the **Security** tab to give other users permission to use this Record Template:



- i. Select users/groups from the Names list who are to have access to this Record Template.
 - ii. Select **Add** . Continue to select all users/groups who are to have access to this Record Template.
7. Select **OK** .
- Your new Record Template is added to the Record Template list.
8. Select **Close**  to return to your list of matching records
-OR-
Select **OK**  to use the new Record Template.

How to define a Record Template

The description of a Record Template is an XML document. The complete structure is:



Please be aware that there are comments and instructions in the code below. These are included as: `<!-- Comment / Instruction -->` and may require that you add / repeat code.

```
<template maxrecord="number" setIRN="yes|no">
  <tuple>
    <atom name="colname"> <!-- Atomic value: add atomic values as required -->
      text
      <source name="colname" rows="rowlist" nestedrows="rowlist"/>
      <input type="text|integer|float" cols="number" rows="number" increment="number" mandatory="yes|no">
        <prompt>
          text
          <source name="colname" rows="rowlist" nestedrows="rowlist"/>
        </prompt>
        <help>
          text
          <source name="colname" rows="rowlist" nestedrows="rowlist"/>
        </help>
        <value>
          text
          <source name="colname" rows="rowlist" nestedrows="rowlist"/>
        </value>
      </input>
    </atom>
    <table name="colname"> <!-- Nested or double nested table - add tables as required -->
      <tuple> <!-- nested table - repeatable -->
        <atom>
          <!-- Add code as for atom above -->
        </atom>
      </tuple>
    </table>
  </tuple>
</template>
```

```
<tuple> <!-- Double nested table - add as required-->
  <table>
    <source name="colname" rows="rowlist" nestedrows="rowlist"/>
    <tuple> <!-- Nested table - add as required-->
      <atom>
        <!-- Add code as for atom above -->
      </atom>
    </tuple>
  </table>
</tuple>
<source name="colname" rows="rowlist" nestedrows="rowlist"/>
</table>
</tuple>
<report>
  text
  <column name="colname"/>
</report>
</template>
```

Although the XML may look complex, the main part is the specification of the fields that require values to be set. The format of the XML for this part is exactly the same as that generated by the Vitalware XML Export facility, which is the same as that used by the Vitalware XML Import tool. Using this same structure means a skeletal XML record can be generated by building a report with the required fields and producing an XML Export file. Once you have the skeletal XML it can be expanded to include any additional options required.

A quick summary of the XML structure used by the three kinds of fields in Vitalware may make things clearer. The three field kinds are:

- atomic
- nested table
- double nested table

Atomic Fields

An atomic field contains a single value. It is represented by a single data entry area in the Vitalware client. The XML snippet used to represent an atomic value is:

```
<atom name="colname">value</atom>
```

where:

colname is the name of the field
value is the contents of the field

Nested Table Fields

A nested table field contains a list of values. A grid is used to display the list in the Vitalware client. The XML format for a nested table is:

```
<table name="colname">  
  <tuple>  
    <atom>value 1</atom>  
  </tuple>  
  <tuple>  
    <atom>value 2</atom>  
  </tuple>  
  ...  
</table>
```

where:

colname is the name of the field.
value 1, value 2 are the values in the list.
etc.

There is no limit to the number of <tuple> entries in a nested table.

Double Nested Table

A double nested table field consists of a list where each entry is itself a list. The Vitalware client uses the nested form construct (where a grid at the bottom of the tab controls what data is shown) where a grid is displayed in the top part of the tab. The XML required for a double nested table is:

```
<table name="colname">
  <tuple>
    <table>
      <tuple>
        <atom>value 1-1</atom>
      </tuple>
      <tuple>
        <atom>value 1-2</atom>
      </tuple>
      ...
    </table>
  </tuple>
  <tuple>
    <table>
      <tuple>
        <atom>value 2-1</atom>
      </tuple>
      ...
    </table>
  </tuple>
  ...
</table>
```

where:

<i>colname</i>	is the name of the field.
<i>value 1-1, value 1-2</i> etc.	are the list of values in the first list
<i>value 2-1, value 2-2</i> etc.	are the list of values in the second list and so on.

There is no limit to the number of values in any of the lists.

An example record

When specifying a record the three field types are enclosed within `<tuple></tuple>` tags. Let's consider an example where we are to encode the data in the following record:

The XML below represents the data entered by a user and does not include computed values (found in the *Derived Names* group box when *Automatic* is set to Yes):

```
<tuple>
  <atom name="NamPartyType">Person</atom>
  <atom name="NamTitle">Dr</atom>
  <atom name="NamFirst">Isaac</atom>
  <atom name="NamMiddle">Jim</atom>
  <atom name="NamLast">Huels</atom>
  <table name="NamOtherNames_tab">
    <tuple>
      <atom>Zak</atom>
    </tuple>
    <tuple>
      <atom>Jimmy</atom>
    </tuple>
  </table>
  <atom name="NamSex">Male</atom>
  <atom name="NamAutomatic">Yes</atom>
</tuple>
```

Fields that do not contain a value are not specified. You may include empty values if you want to remove any data already in the field (e.g. a Default value added when a new record is added). An empty value consists of a tag of the form:

```
<atom name="colname" />
```

We will now examine each of the tags that may be used to specify a Record Template in detail:

template tag

The `<template>` tag encloses the Record Template XML description. It must be the first tag, and the corresponding closing tag `</template>` must be the last tag. Attributes associated with the tag represent options available when the template is used.

Attributes

`maxrecords`

Specifies the maximum number of records that may be generated using this Template. If a number is given, a hint is displayed next to the *Number of records to create* input text box (page 3). If the attribute is not specified, no limit exists on the number of records that can be created.

`setIRN`

Indicates whether a starting IRN may be entered when the Record Template is used. If a value of `no` is supplied, the *Starting IRN* prompt and input box are removed from the Record Template Records screen. The default value is `yes`.

Contains

`<tuple>`
`<report>`

Contained within

`none`

source tag

The `<source>` tag extracts information from the source record (the current record). The tag is replaced with the value(s) extracted. Data can be extracted from any field kind (atomic, nested table and double nested table) into any field kind. Where a mismatch between the field kinds occurs the data is either converted to a newline separated value (when going from a table to an atomic field) or wrapped in table XML (when going from an atomic value to a table). Using attributes it is possible to extract parts of tables or double nested tables.

Attributes

`name`

Specifies the name of the column from which the value is to be extracted. The `name` attribute is mandatory.

`rows`

Contains a list of numbers indicating which rows should be extracted from a nested table. The list is a comma separated set of numbers or ranges. An

example list setting is `rows="1,3-5,7-"` which indicates that rows one, three to five and seven onward are to be extracted. If this attribute is not specified, all rows are extracted.

nestedrows

Contains a list of rows indicating which of the outer rows in a double nested table are to be extracted. The format of the row list is the same as for the `rows` attribute. The `rows` attribute is used to specify the inner row numbers to be extracted. The default is to extract all nested rows.

Contains

None

Contained within

`<table>`
`<atom>`
`<prompt>`
`<help>`
`<value>`

input tag

The `<input>` tag indicates that the user should be asked to enter a value when the Record Template is used. The tag is replaced with the value entered. Each `<input>` tag found in the Record Template description produces an Input Values screen when the Template is used.

Attributes

type

Defines the type of data the user may enter. The available types are:

- `text`
- `integer`
- `float`

When the user moves out of the input text box a check is made to ensure a legal value has been input. The default type is `text`.

cols

Indicates the width in characters of the input box displayed on the Input Values screen when the Template is used. The number does not limit the length of the value that may be entered. The default is 8.

rows

Specifies the number of rows the input text box should display on the Input Values screen when the Template is used. Users may enter more lines than the number specified. The default is 1.

increment

Indicates that the value entered should be incremented by the increment

amount after each record is created. For example, setting `increment="1"` would increase the value entered by the user by one for every record created.

`mandatory`

Determines whether an input value must be specified. A value of `no` indicates an empty value is acceptable, while `yes` ensures that a value is entered. The default value is `yes`.

Contains

```
<prompt>  
<help>  
<value>
```

Contained within

```
<input>
```

prompt tag

The `<prompt>` tag appears within an `<input>` tag and defines the prompt displayed on the Input Values screen. The default prompt is *Input value*.

Attributes

None

Contains

```
text  
<source>
```

Contained within

```
<input>
```

help tag

The `<help>` tag appears within an `<input>` tag and specifies a help message displayed below the input text box on the Input Values screen. If a help tag is not defined, a help message is not displayed.

Attributes

None

Contains

```
text  
<source>
```

Contained within

```
<input>
```

value tag

The `<value>` tag appears within an `<input>` tag and contains the initial value shown in the input text box on the Input Values screen. If a value is not specified, the input text box will be empty.

Attributes

None

Contains

text
`<source>`

Contained within

`<input>`

records tag

The `<records>` tag is replaced with the number of records to be created.

Attributes

None

Contains

None

Contained within

`<atom>`

number tag

The `<number>` tag is replaced with the number of the record being created.

Attributes

None

Contains

None

Contained within

`<atom>`

report tag

The `<report>` tag defines the text to be displayed for each record created in the Record Template report file. The text defined should identify the record created

uniquely (e.g. *Registration Number*).

Attributes

None

Contains

text
<column>

Contained within

<template>

column tag

The <column> tag is replaced with the value in the specified column name in the created record.

Attributes

name

Specifies the name of the column from which the value is to be extracted. The name attribute is mandatory.

Contains

none

Contained within

<report>

Some example Record Templates

Example 1

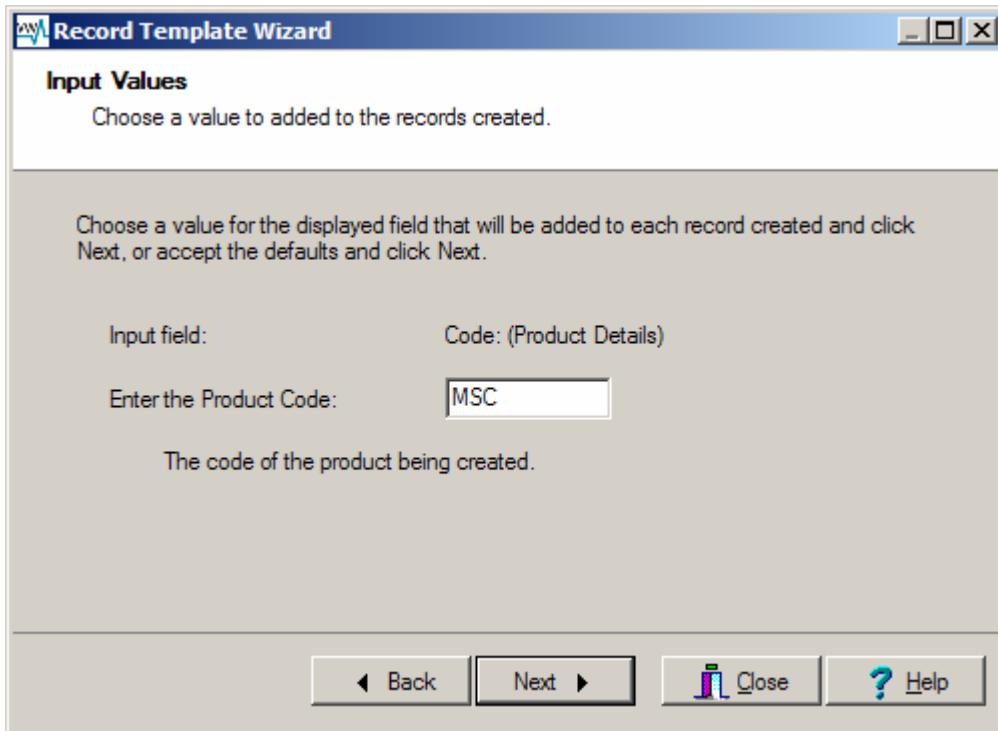
The Record Template XML for our first example is:

```
<template>
  <tuple>
    <atom name="DetProductType"><source
name="DetProductType"/></atom>
    <atom name="DetProductStatus">Available</atom>
    <atom name="DetProductCode">
      <input cols="15" rows="1" mandatory="yes">
<prompt>Enter the Product Code:</prompt>
        <help>The code of the product being created.</help>
      </input>
    </atom>
    <atom name="DetPrice">
      <input type="float" cols="10" rows="1" mandatory="yes">
<prompt>Enter the price:</prompt>
        <help>The price of the product.</help>
      </input>
    </atom>
  </tuple>
</template>
```

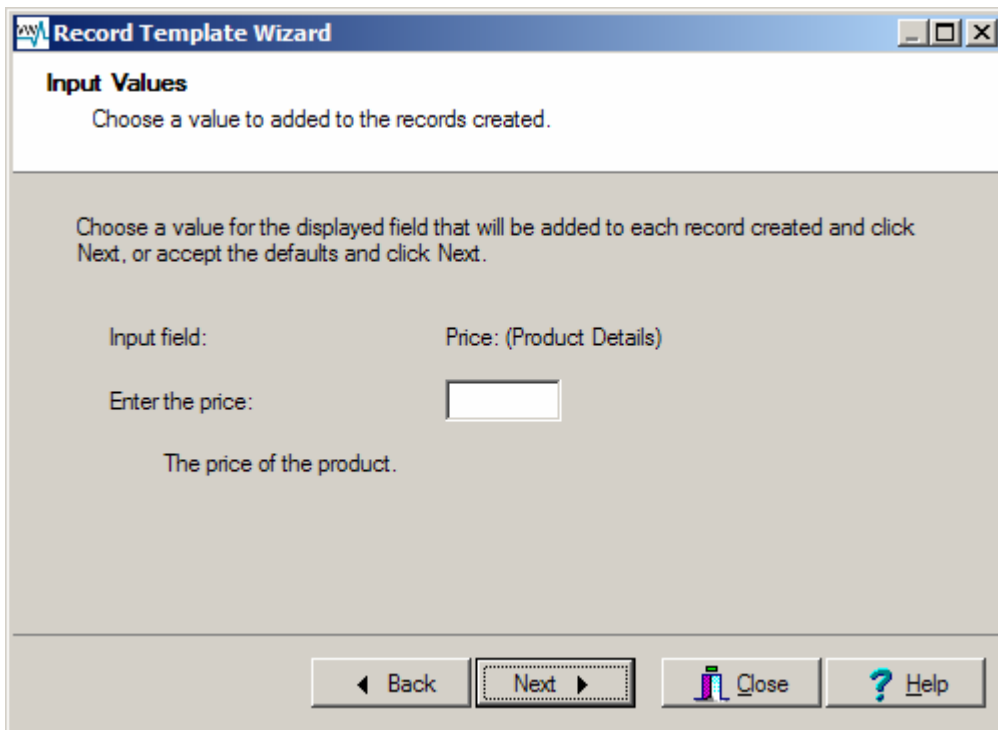
The Template is used to create Product records in the Products module. A suitable title would be *Create New Products*. The following values are set in the records created:

- Available in the *DetProductStatus* field.
- Copies the Product Type from the current record to the *DetProductType* field in the template records.
- Asks for the Product Code and sets the *DetProductCode* field to the value entered.
- Asks for the price and sets the *DetPrice* field to the value entered.

The Input Values screen used to request the 'Product Code' looks like:



The Input Values screen used to request the 'Price' looks like:



Notice how the input prompt and help use the text specified in the Template XML.

Example 2

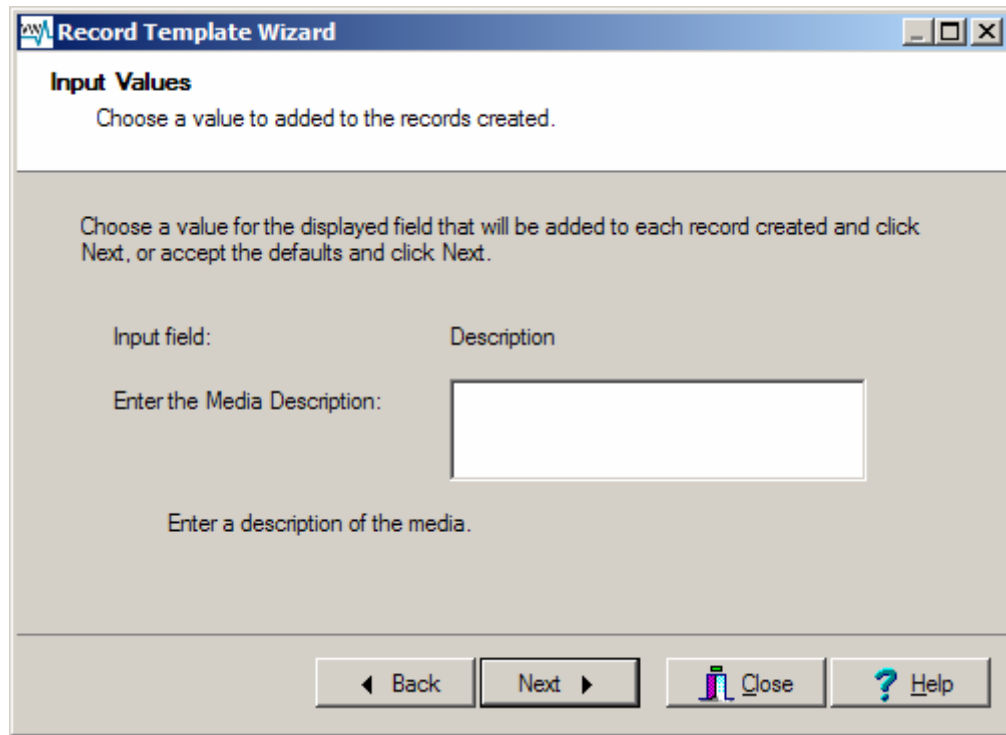
In this example we set up a Record Template for the Multimedia module that copies the Dublin Core fields and asks the user for the *Title*, *Creator* and *Description*. The input fields do not have to have a value, but are initialised with the value from the source record. The maximum number of records to be created will be set to 10 and a starting IRN cannot be specified. The following Record Template XML is suitable:

```

<template maxrecords="10" setIRN="no">
  <tuple>
    <table name="DetSubject_tab"><source name="DetSubject_tab"/></table>
    <table name="DetContributor_tab"><source name="DetContributor_tab"/></table>
    <table name="DetLanguage_tab"><source name="DetLanguage_tab"/></table>
    <table name="DetRelation_tab"><source name="DetRelation_tab"/></table>
    <table name="DetDate0"><source name="DetDate0"/></table>
    <atom name="DetResourceType"><source name="DetResourceType"/></atom>
    <atom name="DetPublisher"><source name="DetPublisher"/></atom>
    <atom name="DetCoverage"><source name="DetCoverage"/></atom>
    <atom name="DetSource"><source name="DetSource"/></atom>
    <atom name="DetRights"><source name="DetRights"/></atom>
    <atom name="MulTitle"><input cols="30" mandatory="no">
      <prompt>Enter the Media Title:</prompt>
      <help>Enter the title of the media.</help>
      <value><source name="MulTitle"/></value>
    </input></atom>
    <table name="MulCreator_tab">
      <tuple>
        <atom><input cols="30" mandatory="no">
          <prompt>Enter the Media Creator:</prompt>
          <help>Enter a description of the media.</help>
          <value><source name="MulCreator_tab" rows="1"/></value>
        </input></atom>
      </tuple>
    </table>
    <atom name="MulDescription"><input rows="3" cols="40" mandatory="no">
      <prompt>Enter the Media Description:</prompt>
      <help>Enter a description of the media.</help>
      <value><source name="MulDescription"/></value>
    </input></atom>
  </tuple>
</template>

```


Notice how the first value of the *MulCreator_tab* table is extracted as the default value for the media creator. The picture below shows the Input Values screen for media description:



The screenshot shows a window titled "Record Template Wizard" with a standard Windows title bar (minimize, maximize, close buttons). The main content area is titled "Input Values" and contains the following text:

Choose a value to added to the records created.

Choose a value for the displayed field that will be added to each record created and click Next, or accept the defaults and click Next.

Input field: Description

Enter the Media Description:

Enter a description of the media.

At the bottom of the window, there are four buttons: "Back" (with a left arrow), "Next" (with a right arrow), "Close" (with a window icon), and "Help" (with a question mark icon).

Vitalware Documentation

XSLT Processing Of XML Import Files

Document Version 1.0

Vitalware Version 2.1



Contents

Overview	1
XSLT processing	3
Pre-configured XSLT files	8

Overview

The advent of XML (eXtensible Markup Language) has provided a standards based mechanism for exchanging data between computer systems. XML, as the name implies, is extensible; that is the format in which the data is stored can be adapted to suit the data source. While this is one of the strengths of XML it also causes problems when importing data from one system into another in which the data formats do not match exactly. For example, consider this XML snippet describing a work of art in an imaginary Births:

```
<table name="ebirths">
  <tuple>
    <atom column="ChildNames">Agnes</atom>
    <atom column="ChildDOB">2007-07-02</atom>
    <atom column="MotherGivenNames">Samsani</atom>
    <atom column="MotherSurname">Annalise</atom>
  </tuple>
</table>
```

You receive this data from another institution using Vitalware and want to import it into your system, but there is a mismatch between some of the column names in your system and those in the originating institution. For example, in your Births module, the *ChildNames* column may be called *ChildGivenNames* and the *ChildDOB* column may be called *ChildDateOfBirth*. Before you can load the XML into your system it is necessary to *transform* it so that it appears like:

```
<table name="ebirths">
  <tuple>
    <atom column="ChildGivenNames">Agnes</atom>
    <atom column="ChildDateOfBirth">2007-07-02</atom>
    <atom column="MotherGivenNames">Samsani</atom>
    <atom column="MotherSurname">Annalise</atom>
  </tuple>
</table>
```

One way to make the change is to use a text editor and replace all instances of *ChildNames* with *ChildGivenNames* and *ChildDOB* with *ChildDateOfBirth*. If the amount of data is small or if the import is to occur only once then this solution is feasible. If, however, a number of imports will occur in which the data will be supplied in the same format, it makes sense to use XSLT (eXtensible Stylesheet Language Transforms) to apply the changes before the data is loaded. XSLT is an XML-based scripting language used to manipulate XML.

For example, the following script can be used to perform the required column renaming outlined above:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:map="urn:map" version="1.0">
  <!-- Output in XML format -->
  <xsl:output method="xml" encoding="utf-8"/>

  <!-- Mapping table of old names to new names -->
  <map:entries>
    <map:entry oldname="ChildNames" newname="ChildGivenNames"/>
    <map:entry oldname="ChildDOB" newname="ChildDateOfBirth"/>
  </map:entries>
  <xsl:variable name="map" select="document('')/*/*map:entries/*"/>
  <!-- For every node we copy it over. Note that attributes
  are handled by the next template. -->
  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

  <!-- Special handling of attributes. -->
  <xsl:template match="@*">
    <xsl:variable name="entry" select="$map[@oldname =
current()]" />
    <xsl:choose>
      <xsl:when test="name() = 'column' and $entry">
        <xsl:attribute name="column">
          <xsl:value-of select="$entry/@newname" />
        </xsl:attribute>
      </xsl:when>
      <xsl:otherwise>
        <xsl:copy/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>

```

To execute the XSLT script an XSL engine is required. A number of products provide XSL engines that can be used to transform the XML for loading into Vitalware. One such product is Cooktop (<http://www.xmlcooktop.com>). When a file is received from an institution, it is only necessary to perform the transformation before importing the XML into Vitalware.

Vitalware 2.1.01 has streamlined the above process by adding XSLT processing as part of the Import tool for XML files: it is now possible to import an XML file and have it transformed as part of the Import process. The XSLT file used to transform the XML can be stored on your local machine (*local file*) or on the Vitalware server (*pre-configured file*). Files stored on the Vitalware server are available to all users. In general, the pre-configured files are "standard" transformations used to manipulate data from known sources. A known source can be:

- a standard format (e.g. Darwin Core or Dublin Core)
- a repeatable format.

Using *repeatable formats* it is possible to define XSLT files that allow for easy import of data from other Vitalware clients for customised modules such as the Births, Deaths and Marriages Events.

XSLT processing

The Vitalware Import Wizard has been extended to provide XSLT processing for XML-based import files. The extensions are only available for files with a .xml file suffix. If you have XML files with a .txt suffix, it will be necessary to rename them in order to use the XSLT processor.

To access the XSLT processor, in the module in which records are to be imported:

1. Select **Tools>Import** from the Menu bar.

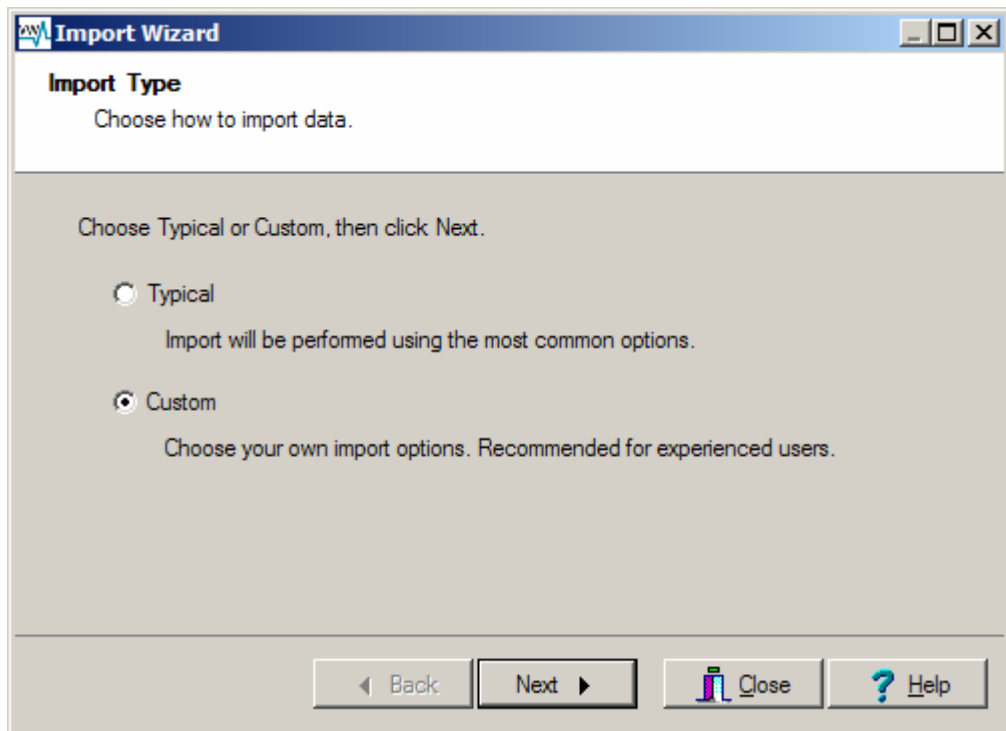


If the Import option is greyed out, you do not have the necessary permissions to use it.

The Select File To Import box displays.

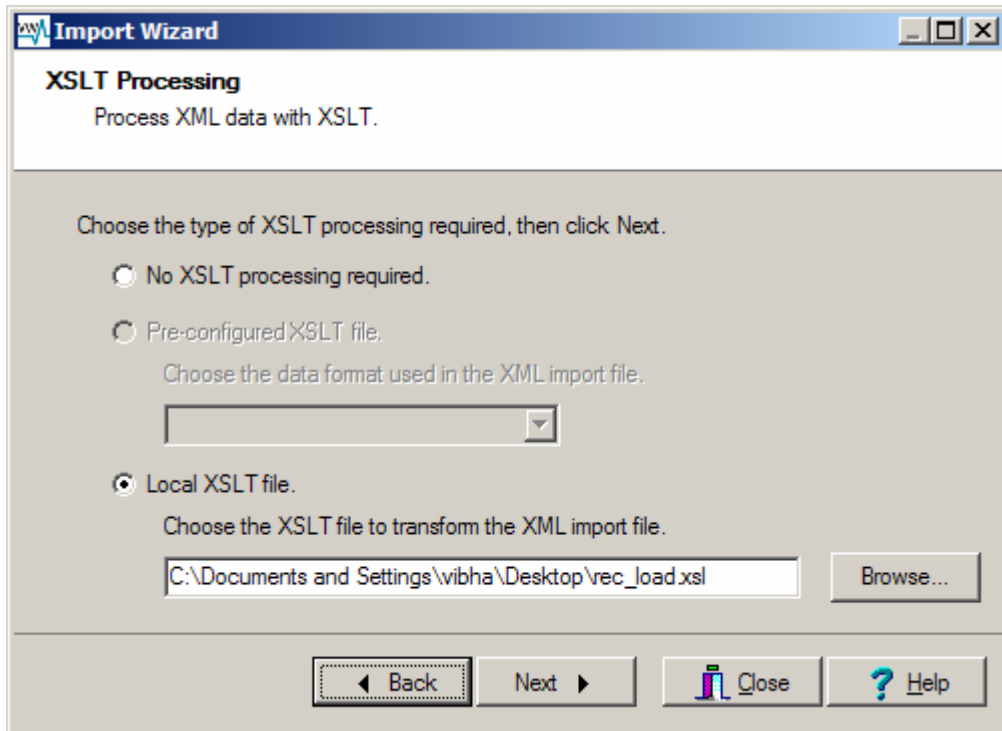
2. Navigate to the file that contains the data to be imported, select it and select **Open**.

The Import Wizard displays:



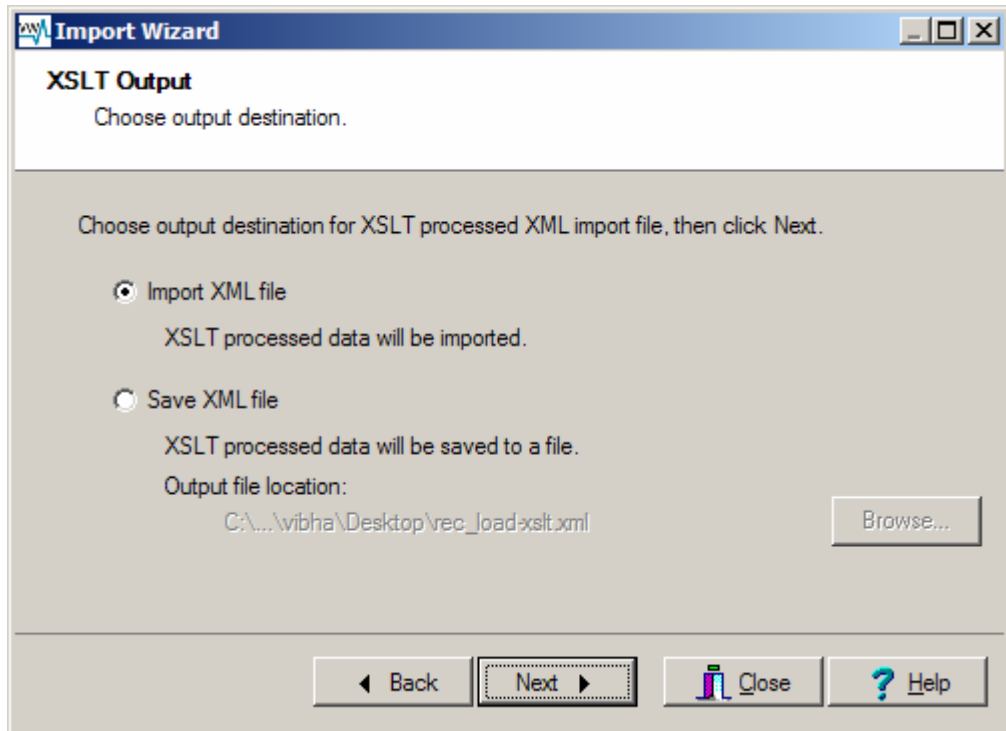
3. Select **Custom** from the Import Type screen and click **Next**.

The XSLT Processing screen displays:



Three options are available:

- **No XSLT processing required**
The XSLT processor is not invoked and the import data file is passed to the Import tool for loading.
 - **Pre-configured XSLT file**
A drop-list is populated with all the server side XSLT files. These files contain "standard" XSLT scripts used to transform known XML formats. Selecting this option and one of the pre-configured entries will result in the XSLT file being copied from the server to your local machine and executed by the XSLT processor.
 - **Local XSLT file**
If you want to use an XSLT file that resides on your local machine, choose this option and browse to the file.
4. To use the XSLT processor choose the second or third option and select **Next**.
The XSLT Output screen will display:



Two options are available:

- **Import XML file**

The output of the XSLT processor (the transformed XML) is fed into the Import facility for loading. The transformed XML is saved in a temporary file used by the Import tool. All error messages relating to the import refer to this temporary file. The name of the temporary file can be determined by using the **Verbose** option on the Logging screen. The temporary file is not removed until the **Finished** button is clicked on the Importing screen.

- **Save XML file**

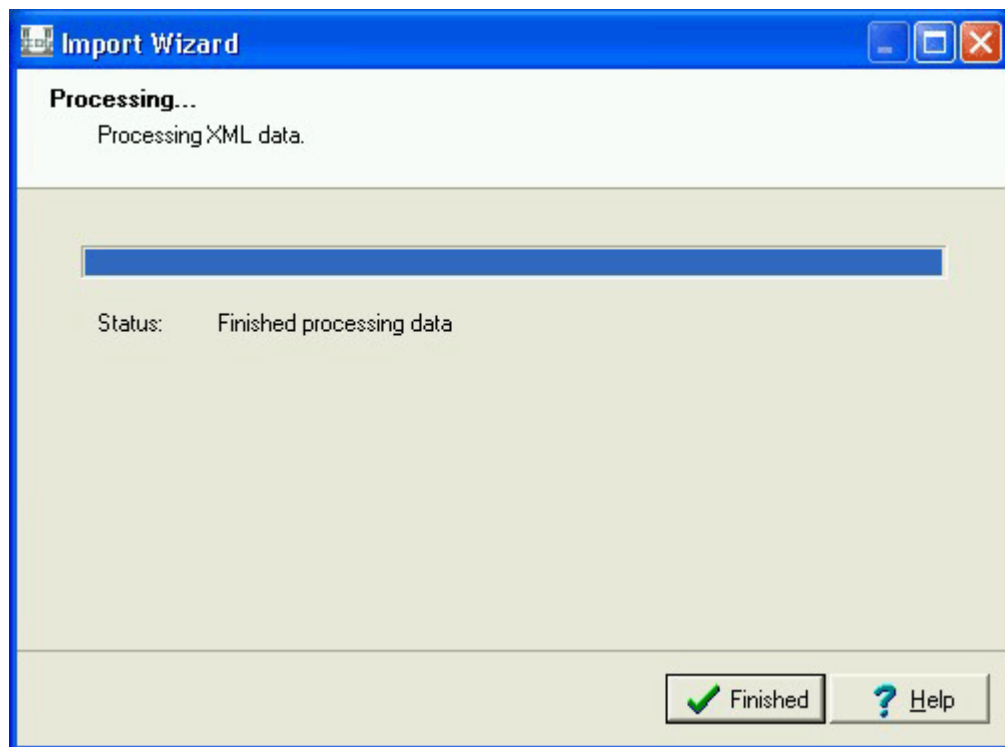
If you only want to run the XSLT processor and view the output of the transformation, use this option to select the name of the file into which the generated XML will be saved. The data import phase will not be run.

If **Save XML file** is selected, the level of logging can be set and the XSLT processing invoked; if the **Import XML file** option is selected, the normal Import sequence is followed (see the Vitalware Help for details).

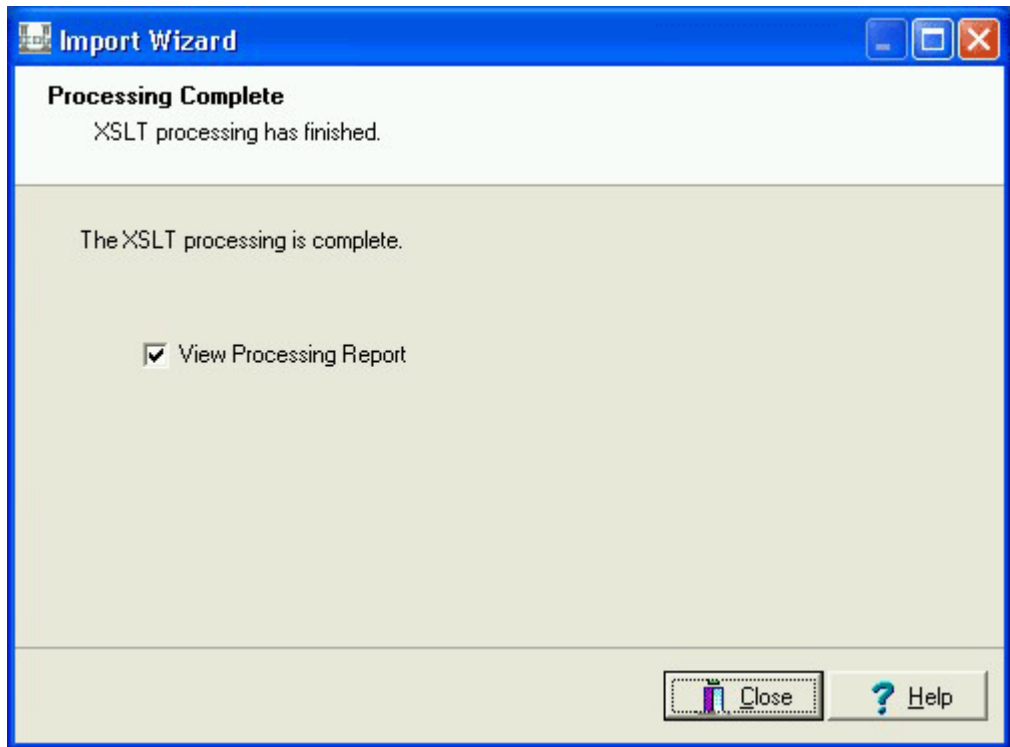
The table below indicates when the XSLT processor is invoked and whether the Import phase is executed:

Option	XSLT	Import
No XSLT processing required	✗	✓
Pre-configured XSLT File/Import XML file	✓	✓
Pre-configured XSLT File/Save XML file	✓	✗
Local XSLT File/Import XML file	✓	✓
Local XSLT File/Save XML file	✓	✗

When the XSLT processor is run a screen showing the status of the processing is displayed. Once the transformations are complete the Import phase will begin automatically for options that require the data to be imported (see the Vitalware Help for details). If the data is not imported (e.g. saving XML to a file), the processing screen will indicate that the transformations are complete:



5. When the **Finished** button is clicked the final screen displays allowing the generated report to be viewed:



The Vitalware XSLT processor uses the Microsoft XML libraries (MSXML). In order to use the XSLT processor it is necessary to have MSXML 3.0 or later installed (Windows 2000 SP4 or Internet Explorer 6 or later, Windows XP, Windows Vista, Windows Server 2003).

Pre-configured XSLT files

As described above (page 2) it is possible to have pre-configured XSLT files stored on the Vitalware server. These files are accessible to all users and are listed in the drop-list below the **Pre-configured XSLT file** option. The files are stored in a per table directory in one of two locations:

- `etc/import/table`
Location of client independent XSLT scripts. These script typically load into the core Vitalware modules that do not vary from client to client (e.g. Parties, Multimedia, etc.). Clients should not add scripts to this location as these scripts are added by KE Software.
- `local/etc/import/table`
Location of client specific XSLT scripts. Any scripts that transform data for institution specific modules (e.g. Births, Deaths) should be kept in this location. All client scripts should be added to this location.

When installing a script on the Vitalware server the `local/etc/import/table` directory may not exist, in which case it will be necessary to create it. For example, if you have a script called "TRANSFORMBTH.xslt" that transforms the XML for loading into your Vitalware Births module, you would store it under:

```
local/etc/import/ebirths/TRANSFORMBTH.xslt
```

The entry that appears in the drop-list in the Import wizard is the name of the file without its file suffix (e.g. TRANSFORMBTH for TRANSFORMBTH.xslt). The file name may contain spaces. XSLT scripts do not need to have an .xslt suffix, however this is the extension usually used.

Vitalware Documentation

FIFO Server

Document Version 1.0

Vitalware Version 2.1



Contents

SECTION 1	FIFO Server	1
	Overview	1
SECTION 2	Invoking the FIFO server	3
	Scripts and command line	4
	KE Texpress Validation	5
	C++ Client Code	6
SECTION 3	FIFO Server plugins	7
SECTION 4	Standard plugins	13
	Work Hours	13
	System Lookup	14
SECTION 5	Index	15

SECTION 1

FIFO Server

Overview

One issue with using a generic database server is that values often need to be computed or processes invoked when saving a record. In many cases these computations occur within the confines of the database server itself. In the case of KE Texpress (the database engine used by Vitalware) a powerful scripting language allows values to be computed and data adjusted when a record is saved. When a command needs to be run, the `system()` call may be used. In the case where complex data computations are required, particularly where the computed values are the result of other table lookups, the `system()` call provides a useful solution. The format of the call is:

```
result = system("command");
```

The "command" provided as the argument is run and the output returned. Where values are computed, the output is assigned to one or more columns. If a process is invoked (for example printing a specimen label), the output may be empty. The `system()` call provides a useful mechanism for allowing external programs to be invoked.

Similar functionality is available through the C++ TexVCL objects used by the client. The **KESession** object provides the `Execute()` method:

```
status = Session->Execute("command", output, error);
```

where:

<code>command</code>	is the program to be executed.
<code>status</code>	is the exit status of the command (zero indicates the command completed successfully).
<code>output</code>	is an AnsiString that receives any output sent to <i>stdout</i> .
<code>error</code>	is an AnsiString that receives any output sent to <i>stderr</i> .

Most script languages provide a mechanism for invoking commands from within the script itself and capturing the output (e.g. perl has a `system()` call and also provides back ticks).

The problem with invoking a command to start a process or to generate values is the expense in terms of computing resources. Each command invoked needs to learn about its environment (e.g. if a database table is consulted, the table schema needs to be loaded each time the command is called). Also, due to the way commands are started in a UNIX environment (via the `fork()` call), commands invoked by large programs (e.g. `texserver`) start up slowly. It is this combination of slow start up and the constant reloading of the environment that may result in a high load being placed on the server machine.

The FIFO service was introduced in KE Vitalware 2.1.01 to address these two issues:

- The first is addressed by removing the need to start a new command. Instead the FIFO server provides mechanisms where KE Texpress, C++ client code and scripts can ask the FIFO server to perform some function. Similar to command execution, the return value is sent back to the caller. The big difference however is that the FIFO server is running all the time, rather than starting each time a request is made. This removes the need for a command to be started.
- The second issue is addressed by the FIFO server providing access to resident database servers, rather than starting a new database server each time data is required. As the database servers are resident, the schema is only read when the database server is first loaded.

Using the FIFO server dramatically increases the rate at which commands can be executed, while lowering the overall load placed on the server. The FIFO server has been designed to be extensible by adding plugins that provide new functionality.

SECTION 2

Invoking the FIFO server

The FIFO server requires two pieces of information and returns one. The two pieces of information required as input are:

plugin The name of the function inside the FIFO server to be invoked. The name can consist of any characters except a newline character. It is normally descriptive (e.g. *Work Hours* to invoke the Work Hours calculator).

data Information forwarded to the plugin used to compute values or start processes. For example the data:

```
27/11/2009,10:00:00|03/12/2009,12:30:00
```

supplied to the *Work Hours* plugin provides two dates and times (the first set is the Start Date & Time, the second set is the End Date & Time) for which the Work Hours are to be calculated. The format of the input *data* is plugin dependent.

The information returned is the computed value. The format of the data returned depends on the plugin invoked. For example, the value returned for the above *Work Hours* input is (based on the default values discussed later on page 13):

```
4|2|30|0
```

Where the first value (4) represents the no. of Work Days, the second value (2) is the no. of Work Hours, the third value (30) is the no. of Work Minutes and the fourth value (0) is the no. of Work Seconds.

Only one instance of the FIFO server runs for a given client environment. All users access this instance when requests are made of the FIFO server. In order to ensure that all requests are handled serially, a simple file locking mechanism is used. This guarantees that the correct output is received for the input provided.

The FIFO server is installed as a background load. The `vwload` command is used on the server to control access:

To start the FIFO server use:

```
vwload start fifo
```

To check the status of the server use:

```
vwload status fifo
```

To stop the server use:

```
vwload stop fifo
```

Scripts and command line

The command `vwfifo` is used to invoke the FIFO server from the command line or from within a script. Its usage message is:

```
Usage: vwfifo plugin [data]
```

where

plugin	name of fifo plugin to invoke
data	data passed to fifo plugin [default: stdin]

For the *Work Hours* plugin example above, the following command could have been used:

```
vwfifo "Work Hours" << EOF
27/11/2009,10:00:00|03/12/2009,12:30:00
EOF
4|2|30|0
```

The data may also be supplied as an argument:

```
vwfifo "Work Hours" "27/11/2009,10:00:00|03/12/2009,12:30:00"
4|2|30|0
```

giving the same response. `vwfifo` is often used to debug new plugins.

KE Texpress Validation

The FIFO server may also be invoked from within KE Texpress. When a record is saved, a validation handler is run. The handler checks for consistent data but may also be used to compute values. The following code segment shows how to invoke the FIFO server from within the validation handler:

```
/* FIFO settings.
*/
fifoin = getenv("VWPATH") . "/loads/fifo/input";
fifoot = getenv("VWPATH") . "/loads/fifo/output";
fifolock = getenv("VWPATH") . "/loads/fifo/lock";

/* "System Yes" value
*/
if ((YES = getenv("SYSYES")) == "")
{
    YES = fifo(fifoin, fifoot, fifolock, "System
Lookup\nSystem Yes");
    setenv("SYSYES", YES);
}
```

The `fifo()` call is used to communicate with the FIFO Server. Its arguments are:

<code>fifoin</code>	The path to the input side of the FIFO server. The name of the plugin and data are written to this file (the file is actually a named pipe created when the server is invoked).
<code>fifoot</code>	The path to the output side of the FIFO server. The results are read from this file (the file is also a named pipe created when the server is invoked).
<code>fifolock</code>	The path to an empty file used as a lock to ensure that only one process can access the FIFO server at a time. The locking ensures that correct results are returned for a given request.
<code>fifovalue</code>	The information to be forwarded to the FIFO server. The first line must contain the name of the plugin to be invoked. All remaining lines are passed to the plugin as data.

The code above calls the *System Lookup* plugin (which returns the value associated with the name of the lookup list supplied as data), asking for the value of the *System Yes* table. The returned value is remembered so it only needs to be looked up once. The values for `fifoin`, `fifoot` and `fifolock` defined above should always be used. Care should be taken with values returned by the FIFO server. In many cases the return value may have a trailing newline character that may need to be removed.

C++ Client Code

A new method has been added to the **KESession** object that communicates with the FIFO server. The method is:

```
AnsiString  
__fastcall  
KESession::Fifo(AnsiString fifoin, AnsiString fifoout, AnsiString  
fifolock, AnsiString fifovalue)
```

The arguments are the same as for the Texpress validation call. The return value is the information sent back from the FIFO server. In order to provide easier access to the server, a new method has been added to the base window class **TBaseFrm** that invokes `Fifo()` with the correct paths. The method is:

```
AnsiString  
__fastcall  
TBaseFrm::FifoServer(AnsiString plugin, AnsiString data)
```

The simplified version only requires the name of the plugin to invoke and any data to be passed to it. For example to get the value of the *System Yes* lookup list, the following call could be used:

```
AnsiString results = FifoServer("System Lookup", "System Yes");
```

SECTION 3

FIFO Server plugins

The FIFO server is designed to be extensible. In fact the server itself just provides a framework without any services built in. All services are provided by **plugins** that are loaded when the FIFO server is started. A plugin is really just a perl library that contains a registration function used to define what plugin types are handled. The standard plugins are located in **etc/fifo**, while client specific plugins can be found in **local/etc/fifo**. When the FIFO server starts it looks in both the standard and local directories for all files with a **.pl** extension (a perl library). The file is loaded and the `Register()` function invoked to determine what plugins are handled by the script.

The shell of a plugin looks like:

```
#!/usr/bin/perl
#
# Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
no warnings 'redefine';

#
# Registration function.
#
sub
Register
{
    my $plugins = shift;

    #
    # We handle the "Plugin Name" method.
    #
    $plugins->{"Plugin Name"} = \&Plugin;
}

#
# The handler for the "Plugin Name" plugin
#
sub
Plugin
{
    my $plugin = shift;
    my $data = shift;
```



```

        #
        # Plugin code
        #
    }

1;

```

The `Register` subroutine is called by the FIFO server passing in a reference to a hash. It is necessary to add the following to the hash:

- The name of the plugin to be handled.
- A reference to the function to invoke when the plugin is called.

As many different handlers as required may be registered within the one plugin. When a call is made to the FIFO server and the plugin name matches the one registered, the corresponding plugin subroutine is called. Two arguments are supplied to the plugin subroutine:

- The plugin name that matched.
- A reference to a list of input lines (where the newline has been removed from each line).

Any value returned by the handler is sent back to the caller.

An example may make things clearer. Let's create a local plugin that provides two functions:

- Addition, which will add up all the numbers supplied as data.
- Multiplication, which will multiply the numbers supplied.

The plugin will be placed in **local/etc/fifo/math.pl**. The code is:

```

#!/usr/bin/perl

#
# Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
no warnings 'redefine';

#
# Registration function.
#
sub
Register
{
    my $plugins = shift;

    #
    # We handle the "Addition" and "Multiplication" method.

```

```
#
$plugins->{"Addition"} = \&Addition;
$plugins->{"Multiplication"} = \&Multiplication;
}

#
# The handler for the "Addition" plugin
#
sub
Addition
{
    my $plugin = shift;
    my $data = shift;
    my $total = 0;

    #
    # Add up the values supplied
    #
    foreach my $value (@{$data})
    {
        $total += $value;
    }
    return($total);
}

#
# The handler for the "Multiplication" plugin
#
sub
Multiplication
{
    my $plugin = shift;
    my $data = shift;
    my $total = 1;

    #
    # Add up the values supplied
    #
    foreach my $value (@{$data})
    {
        $total *= $value;
    }
    return($total);
}

1;
```

Notice that the `Register` subroutine adds two handlers (one for `Addition` and one for `Multiplication`). Associated with each handler is the subroutine to call when

the handler is matched (Addition and Multiplication respectively). Next we restart the FIFO server (using `vwload stop fifo` and `vwload start fifo`). Now we can use `vwfifo` to test our plugin:

```
vwfifo Addition << EOF
1
2
3
4
EOF
10
vwfifo Multiplication << EOF
1
2
3
4
EOF
24
```

Although the example above is trivial it does present the basics involved in setting up a new plugin. In many cases the plugin needs to access data stored in a KE Texpress table. In this case the plugin can use either `OldServer()` to get a reference to a **texql** object (as defined in `texql.pm`) or `NewServer()` for a reference to a **texapi** object (as defined in `texapi.pm`). Using either of these objects you can retrieve data from existing tables and use it to build the result. As an example the plugin below determines whether a Birth IRN has any Death attached:

```
#!/usr/bin/perl

#
# Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
no warnings 'redefine';

#
# Registration function.
#
sub
Register
{
    my $plugins = shift;

    #
    # We handle the "Has Death" method.
    #
    $plugins->{"Has Death"} = \&Death;
```

```
}

#
# The handler for the "Has Death" plugin
#
sub
Death
{
    my $plugin = shift;
    my $data = shift;
    my $texql;
    my $row;

    #
    # Check if we have any records.
    #
    $texql = OldServer();
    $texql->Command(
        "select all from ebirths where DeathRegistration is
not NULL";

    #
    # Get the result
    #
    $row = $texql->Row();
    $texql->Finish();

    #
    # Return the result
    #
    return(defined($row) ? "Yes" : "No");
}
1;
```


SECTION 4

Standard plugins

The FIFO server provides two standard plugins as part of the Vitalware 2.1.01 distribution. These are:

- *Work Hours*
- *System Lookup*

Work Hours

The *Work Hours* plugin returns the total work time elapsed between the Start Date/Time and the End Date/Time. It uses the following registry entries to determine the work hours:

Group|Default|Table|eorders|Daily Business Hours|0|Value

Group|Default|Table|eorders|Daily Business Hours|1|Value

Group|Default|Table|eorders|Daily Business Hours|2|

Group|Default|Table|eorders|Daily Business Hours|3|

Group|Default|Table|eorders|Daily Business Hours|4|

Group|Default|Table|eorders|Daily Business Hours|5|

Group|Default|Table|eorders|Daily Business Hours|6|

Group|Default|Table|eorders|Business Day Length|

System|Setting|Public Holidays|

The following default values will be used in case the above registry entries are not found:

Group|Default|Table|eorders|Daily Business Hours|0|00:00-00:00

Group|Default|Table|eorders|Daily Business Hours|1|08:00-17:00

Group|Default|Table|eorders|Daily Business Hours|2|08:00-17:00

Group|Default|Table|eorders|Daily Business Hours|3|08:00-17:00

Group|Default|Table|eorders|Daily Business Hours|4|08:00-17:00

Group|Default|Table|eorders|Daily Business Hours|5|08:00-17:00

Group|Default|Table|eorders|Daily Business Hours|6|10:00-14:00

Group|Default|Table|eorders|Business Day Length|09:00

System|Setting|Public Holidays|10/03/2008;30/06/2008;25/12/2008

```
vwfifo "Work Hours" "27/11/2009,10:00:00|03/12/2009,12:30:00"
```

```
4|2|30|0
```

System Lookup

The *System Lookup* plugin returns the text value for a given system lookup list. The name of the lookup list is supplied as input and the language dependent value is returned.

```
vwfifo "System Lookup" "System Yes"
```

Yes

Index

C

C++ Client Code • 6

F

FIFO Server • 1

FIFO Server plugins • 7

I

Invoking the FIFO server • 3

K

KE Texpress Validation • 5

O

Overview • 1

S

Scripts and command line • 4

Standard plugins • 13

System Lookup • 14

W

Work Hours • 13

Vitalware Documentation

Configuration

Document Version 1.0

Vitalware Version 2.1



Contents

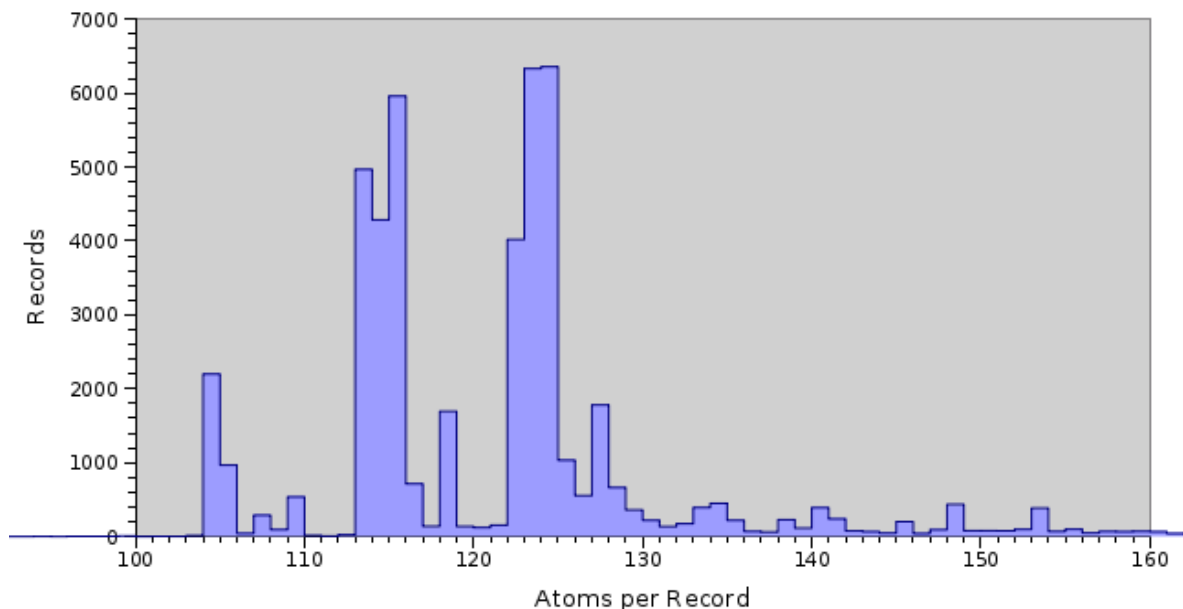
SECTION 1	KE Vitalware Configuration	5
	Overview	5
	The Basic Theory	6
	Coding Scheme	6
	Superimposed Scheme	7
	False Matches	8
	Calculating k and b	8
	Record and Segment Descriptors	12
	Calculating a value for N_r	12
	Bit Slicing	14
	False match probability	15
	What is an atom?	16
	Atoms per record	18
	Summary of variables	23
	Configuration tools	24
	texanalyse	24
	texdensity	26
	texconf	28
	Setting configuration parameters	30

KE Vitalware Configuration

- [Overview](#)
- [The Basic Theory](#)
- [Record and Segment Descriptors](#)
- [What is an atom?](#)
- [Atoms per record](#)
- [Summary of Variables](#)
- [Configuration tools](#)
- [Setting configuration parameters](#)

Overview

Texpress 8.2.01 has seen an overhaul of the configuration tools used to provide optimal indexing for Texpress tables. Trying to achieve optimal performance for a given table has been a bit of a *black art* due to the simplistic approach to automatic configuration taken by earlier versions of Texpress. As a result less than optimal performance has been noticed, particularly for very large data sets. In many instances manual configuration was the only way to get near optimal performance. Most configuration issues resulted from assumptions that were applicable for data sets with a normal distribution of record sizes, but which did not hold for the diverse data sets found in a "normal" Vitalware installation. In particular, where data has been loaded from a number of disparate legacy systems the distribution of record sizes does not follow a single normal distribution, but resembles a number of normally distributed data sets, one per legacy system, overlaying each other. The histogram below shows an example distribution of record sizes for the *Parties* module:



As you can see the distribution of record sizes does not follow a normal distribution. However if you look closely you will notice the histogram is made up of three normal distributions, with centres at 105, 115 and 125, superimposed on each other. Each of these distributions reflects data from a legacy system, so in fact we have three different data sources where each data source is distributed normally, but the combined data set is not!

Prior to Texpress 8.2.01 the automated configuration tools assumed that the data set followed a normal distribution and produced indexes based around this assumption. The end result was that for non normal distributed data sets poor indexing parameters were used, leading to slow response times and excessive false matches. The new configuration facility attempts to cater for all distributions of data sets while still providing optimal querying speed with minimum disk usage. The rest of this document will take a close look at the input parameters to the configuration process as a solid understanding of these values allows targeted manual configuration if it should be required. First of all we need to start with the basics.

The Basic Theory

Texpress uses a *Two Level Superimposed Coding Scheme for Partial Match Retrieval* as its primary indexing mechanism. In this section I would like to explore what we mean by *Superimposed Coding Scheme* and look at the variables that affect optimal configuration. The scheme is made up of two parts: the first is a *coding* scheme, and the second is the *superimposing* mechanism. In order to demonstrate how these strategies function, a working example will be used. For the example we will assume we have a simple Vitalware *Parties* record with the following data:

Field Name	Value
<i>First Name</i>	Boris
<i>Surname</i>	Badenov
<i>City</i>	FrostBite Falls
<i>State</i>	Minnesota

Coding Scheme

The first part of the indexing algorithm involves **encoding** each of the field values into a bit string, that is a sequence of zero and one bits. Two variables are used when encoding a value. The first is **k**, which is the number of 1s we require to be set for the value and the second is **b**, which is the length of the bit string. The variables **b** and **k** are the first two inputs into the configuration of the indexing mechanism.

To encode a value we use a *pseudo random number generator*. We need to call the generator **k** times where the resulting value must be between 0 and **b** - 1. For each number generated we convert the bit position to a 1. The important feature of a pseudo random number generator is that if we provide the same inputs (that is the same **k**, **b** and value), then the same **k** numbers will be generated, thus the same inputs will always produce the same outputs. Suppose we use **k=2** and **b=15** to encode our sample record. The table below shows example bit strings generated for the given **b** and **k** values:

Value	Bit Positions	Bit String
<i>Boris</i>	3, 10	00010 00000 10000
<i>Badenov</i>	1, 4	01001 00000 00000
<i>FrostBite</i>	3, 7	00010 00100 00000
<i>Falls</i>	8, 14	00000 00010 00001
<i>Minnesota</i>	4, 9	00001 00001 00000

Notice how the two words in the *City* field are encoded separately. It is this separate encoding that provides support for word based searching, that is, searching where only a single word is

specified. The pseudo random number generator used by Texpress also takes one other input, the column number of the value being indexed. The reason for this input is that the same word in different columns should result in different bit strings, otherwise a search for the word would find it in all columns (this is how the Texpress *Also Search* facility is implemented, where each *Also Search* column uses the same column number, that of the originating column).

Superimposed Scheme

Once all the bit strings are generated they are OR-ed together to produce the final bit string. The bit string is known as a *record descriptor* as it contains an encoded version of the data in the record, in other words it describes the contents of the record in the form of a bit string. Using our example the resulting record descriptor would be:

00010 00000 10000	OR
01001 00000 00000	
00010 00100 00000	
00000 00010 00001	
00001 00001 00000	
<hr/>	
01011 00111 10001	
<hr/>	

False Matches

The indexing scheme used by Texpress can indicate that a record matches a query when in fact it does not. These *false matches* are due to the encoding mechanism used. When a query is performed the query term is encoded into a bit string as described above. The resulting record descriptor, generally known as a *query descriptor*, is AND-ed with each record descriptor and where the resultant descriptor is the same as the query descriptor, the record matches (that is, the record descriptor has a 1 in every position the query descriptor has a 1). Using our example above, let's assume we are searching for the term **Boris**. We encode the term and compare it against the record descriptor:

00010 00000 10000	AND	(Boris query descriptor)
01011 00111 10001		(Record descriptor)
<div style="background-color: #ffff00; display: inline-block; padding: 2px 10px;">00010 00000 10000</div>		
		(Resultant descriptor)

Since the *resultant descriptor* is the same as the *query descriptor* the record is flagged as a match. Now let's consider searching for **Natasha**. In order to demonstrate a false match, let's assume **Natasha** is encoded as follows:

Value	Bit Positions	Bit String
<i>Natasha</i>	7, 9	00000 00101 00000

When we perform our search we get:

00000 00101 00000	AND	(Natasha query descriptor)
01011 00111 10001		(Record descriptor)
<div style="background-color: #ffff00; display: inline-block; padding: 2px 10px;">00000 00101 00000</div>		
		(Resultant descriptor)

As you can see the query descriptor is the same as the resultant descriptor so it looks like the record matches, except that the record for descriptor 01011 00111 10001 does not contain **Natasha**. This is known as a *false match*. In order to "hide" false matches from users, Texpress checks each record before it is displayed to confirm that the record does indeed contain the specified search term(s); if not, the record is removed from the set of matches.

The reason for false matches is that a combination of the bits set for a series of terms in a record results in 1s appearing in the same positions as for a single term. Using our example you can see that *Frostbite* sets bit seven and *Minnesota* contributes bit nine, which happen to correspond to the bits set by *Natasha*. In order to provide accurate searching we need to minimise the number of false matches.

Calculating k and b

Now that we have had a look at how the superimposed coding scheme works, that is by translating the contents of a record into a record descriptor, we need to investigate how to

calculate **k** (number of bits to set to 1s per term) and **b** (length of the bit string). The first variable we will look at is **k**. In order to calculate **k** we need to introduce two new variables:

- d** The *bit density* is the ratio between the number of 1 bits set and the length of the descriptor. The value is expressed as a percentage. For example a bit density of 25% indicates that one in four bits will be 1 with the other three being zero when averaged over the whole record descriptor. If we use our example record descriptor of 0101100111 10001, there are eight 1 bits set out of a total of 15 bits, which gives a bit density of 8/15 or 53%. Texpress uses a default bit density value of 25%.
- f** The *false match* probability is the number of record descriptors we need to examine to get a false match. For example, a value of 1024 indicates that we expect to have one false match for every 1024 record descriptors checked when searching. Using this variable we can configure the system to provide an "acceptable" level for false matches. Texpress uses a value of 1024 for the false match probability for record descriptors.

The number of bits we need to set for a term is related to both the false match probability and the bit density. If we use a bit density of 25%, one in four bits is set to 1 in our record descriptor. This implies the probability of any given bit being a 1 is 1/4. If we set **k** to be 1, that is we set one bit per term, the probability that the bit is already set is 1/4. If we set two bits (**k**=2), the probability that both bits are set is 1/4 * 1/4 = 1/16. The table below gives more details:

k	Probability all bits set to one	Value
1	1/4	1/4
2	1/4 * 1/4	1/16
3	1/4 * 1/4 * 1/4	1/64
4	1/4 * 1/4 * 1/4 * 1/4	1/256
5	1/4 * 1/4 * 1/4 * 1/4 * 1/4	1/1024
6	1/4 * 1/4 * 1/4 * 1/4 * 1/4 * 1/4	1/4096

If the false match probability is set to 1024 (as used by Texpress), you can see from the above table we need to set **k** to 5. The reason five is selected is that in order to get a false match we need to have all the search term bits set to one, but not contain the term. Hence if **k** is five we have a one in 1024 chance that a record descriptor will match incorrectly. If we decrease the bit density to say 12.5% or one bit in eight set, a **k** value of four is sufficient, as shown in the table below:

k	Probability all bits set to one	Value
1	1/8	1/8
2	1/8 * 1/8	1/64
3	1/8 * 1/8 * 1/8	1/512
4	1/8 * 1/8 * 1/8 * 1/8	1/4096

It is easy to see that if we decrease the bit density (**d**), we also decrease the value of **k**. We can express the relationship between **k**, **f** and **d** as:

$$f = (100 / d)^k$$

With a bit of mathematics we can isolate **k** from the above formula, giving:

$$\mathbf{k} = \log(\mathbf{f}) / \log(100 / \mathbf{d})$$

Now that **k** is calculated we can use it to work out the value of **b** (the length of the bit string). We need to introduce one new variable in order to calculate **b**:

- i** The number of *indexed atoms* per record defines how many values are to be encoded into the record descriptor. In our example the value of **i** is five as we have five terms encoded in the record descriptor (*Boris, Badenov, Frostbite, Falls, Minnesota*). The value of **i** depends on the data within a record, which can vary from record to record. We will revisit **i** later when we look at how to determine a suitable value.

Using the value of **i** we can calculate the value of **b**. Using a simplistic approach we could take the value of **k** (bits to set per term) and multiply it by **i** (number of terms) to get the number of bits set to a 1. If we assume **d** (bit density) of 25%, we need to multiply the number of 1s set by four to get the value of **b**. Expressed as a formula, this is $(i * k * (100 / d))$. Using our example we would have:

$$5 * 5 * (100 / 25) = 100 \text{ bits}$$

So for our sample record we would have the following configuration:

Variable	Description	Value
f	False match probability	1024
d	Bit density	25%
i	Indexed terms per record	5
k	Bits set per term	5
b	Length of bits string (in bits)	100

In fact the formula used to calculate **b** is a bit simplistic. It is useful as an approximation, however it is not completely accurate. When we build the record descriptor for a record we use the pseudo random number generator to compute **k** bits per term. If we have **i** terms, we call the pseudo random number generator **i** times, once for each term. Each term will set **k/b** bits. Thus for each term the probability that a bit is still zero is $(1 - k/b)$, so for **i** terms the probability that a bit is still zero is $(1 - k/b)^i$. From this the probability that a bit is therefore 1 must be $(1 - (1 - k/b)^i)$. We also know the probability of a bit being 1 must be $(d / 100)$, that is the number of one bits based on the bit density. So we end up with:

$$1 - (1 - k/b)^i = d / 100$$

Now with a bit of mathematics we can isolate **b** from the above formula, giving:

$$b = k / (1 - \exp(\log((100 - d) / 100) / i))$$

Using the above formula for **b** with our example record we end up with a value of 90 bits rather than the 100 calculated using the simplistic method. If you did not understand the way **b** was calculated, it is not important, except to say that Texpress uses the latter formula when determining **b**. Now that we have the key concepts in place and have examined the variables used to calculate the number of bits to set per term (**k**) and the length of the bit string (**b**) we need to consider what we mean by *Two Level* when we talk about a *Two Level Superimposed Coding Scheme for Partial Match Retrieval*.

Record and Segment Descriptors

As you can imagine, an indexing scheme that is two level must have two parts to it and, in fact, this is the case. Fortunately the two parts are very similar with both parts using the theory covered in the last section. The first level is the *segment descriptor* level and the second is the *record descriptor* level:

Segment Descriptor	A <i>segment descriptor</i> is a bit string that encodes information for a fixed number of records. It uses the theory discussed above, except that a single segment descriptor describes a group of records rather than a single record. The number of records in a segment is part of the system configuration and is known as N_r . A typical value for N_r is around 10. Segment descriptors are stored sequentially in the <i>seg</i> file under the database directory, that is, the first segment descriptor encodes the first N_r records, the second the next N_r records and so on.
Record Descriptor	The <i>record descriptor</i> level contains one record descriptor per record. Record descriptors are grouped together into lots of N_r with the lots stored one after another in the <i>rec</i> file under the database directory.

The table below shows the relationship between segment descriptors and record descriptors where N_r is 4:

Segment Descriptor 1	Record Descriptor 1
	Record Descriptor 2
	Record Descriptor 3
	Record Descriptor 4
Segment Descriptor 2	Record Descriptor 5
	Record Descriptor 6
	Record Descriptor 7
	Record Descriptor 8
...	...

The segment descriptors are consulted first when a search is performed. For every matching segment descriptor the corresponding N_r record descriptors are checked to find the matching record(s). For each segment descriptor that does not match, the associated N_r record descriptors can be ignored. In essence we end up with a scheme that can very quickly discard records that do not match, leaving those that do match.

Calculating a value for N_r

It may be tempting to set the number of records in a segment to a very large number. After all if the segment descriptor does not match, it means N_r records can be discarded quickly. To a point this is correct. However the larger the number of records per segment, the higher the probability that a given segment will contain a match. For example, let's say we have 100 records per segment. The data for 100 records is encoded in each segment descriptor and record descriptors are grouped in lots of 100. Now let's assume we have 500 records in our database and we are searching for a term that will result in one match. It is clear in this case that only one segment descriptor will match (assuming no false matches), so we need to

search through 100 record descriptors to find the matching record. If the number of records per segment was 10, we would still get one match at the segment level, however we would only need to look through 10 record descriptors to find the matching record.

Setting the number of records per segment trades off discarding a large number of records quickly (by setting the value high) against the time taken to search the record descriptors in a segment if the segment matches (setting the value low). Also, multi-term queries need to be considered. Since a segment descriptor encodes terms from a number of records, a multi-term query that contains all the query terms spread across the records in the segment will match at the segment level, forcing the record descriptors to be checked. For example, if the search terms were *red* and *house* and the first record in a segment contained the word "red", while the third record contained the word "house", then the segment descriptor would match (as it encodes both words). The record descriptors are then consulted to see if any records contain both terms. Since a matching record does not exist in the segment, time has been "wasted" looking for a non-existent match.

When determining the best value for the number of records in a segment it is important to understand how the database will be queried. In particular, if a lot of single term searches are expected, it makes sense to have a reasonably large number of records per segment (say around 20-50). If multi-term searches are used and the search terms are either related or distinctive (that is, they do not occur in many records), the value may also be high.

If, however, multi-term queries will contain common non-related terms, a smaller value for records per segment is required. For example, the Vitalware *Parties* module contains records that have many common terms. Consider searching for all marriage celebrants in London (that is *Role=marriage celebrant* and *City=Glenorchy*). There are probably a lot of records where the role field contains *marriage celebrant*, similarly many records may have a city value of *Glenorchy*. However, there may not be many records that contain both terms. If the number of records per segment is set high, a large number of segment matches will occur (because at least one record in the segment has a role of *marriage celebrant* and at least one other record in the segment has a city value of *Glenorchy*). In this case it is better to set the number of records per segment to be low, say around 10. The Texpress configuration facility will use a value close to 10 for the number of records per segment as this provides a general purpose index without knowledge of the data and expected queries.

There is one other variable used when determining the number of records in a segment and that is the system *blocksize*:

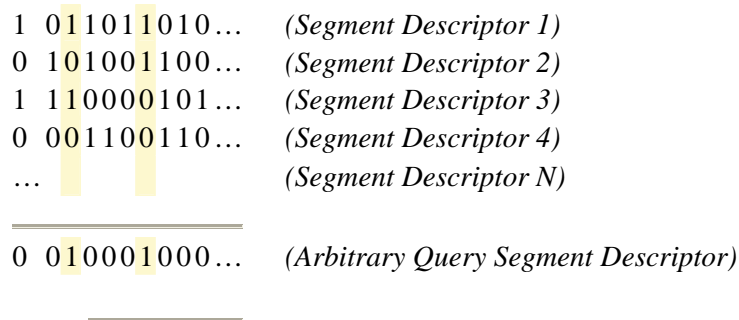
blocksize The *blocksize* is the number of bytes that are read or written at one time when a filesystem is accessed. All filesystem disk accesses occur using this fixed number of bytes. Even if you read one byte, the underlying filesystem will still read *blocksize* bytes and return the single byte to you. The *blocksize* of a filesystem is defined when the filesystem is created. Common blocksizes are 1024, 4096 or 8192 bytes. Texpress assumes a default blocksize of 4096 bytes.

In order to provide efficient searching it is important to ensure that all disk activity occurs in multiples of the filesystem *blocksize*. So, when selecting the number of records in a segment we need to make sure that the value selected fills an integral number of blocks. If we use our example, we saw that the value for **b** (length of the bit string) was 90 bits or 12 bytes (rounding up). If we have a blocksize of 4096 bytes, we can fit 341 (4096/12 rounded down)

record descriptors in one disk block. So for our sample data the value for N_r would be 341. As our example has only five terms in it this leads to a very high value. In practice, records contain many more terms so the number of records per segment is generally around 10 by default.

Bit Slicing

The final piece of the indexing puzzle is the use of *bit slicing* to provide a fast mechanism for searching the segment descriptor file. As discussed, a segment descriptor encodes information from N_r records into one descriptor. The descriptors are stored one after the other in the *seg* file in the database directory. When a query is performed the first step is to search each segment descriptor AND-ing it with the query segment descriptor to see if it matches. When we get a match we then check the record descriptors in the same way. The problem with this approach is that the *seg* file can be very large and searching through it sequentially can take some time. The diagram below shows a series of segment descriptors and a query descriptor used for searching:



It may be obvious from the table above that in order for a segment descriptor to match the query segment descriptor it requires a 1 bit in each position that the query descriptor has a 1 bit. The other bits in the descriptor are irrelevant. Using this piece of information the fastest way to determine what segment descriptors match is to read *slices* of the segment descriptors. Where the query segment descriptor has a 1 bit we read a slice (that is one bit from each segment descriptor). The slice is represented by the yellow area enclosed within the lines in the diagram. If we read a slice for each query descriptor 1 bit and AND them together, any resulting 1 bit must contain the position of a segment descriptor that has all 1 bits set as well. In the example above only segment descriptor 1 matches the query segment descriptor.

The problem with this approach is that reading individual bits from a filesystem is extremely inefficient. You may notice however that if we "flip" the segment descriptor file on its side, each *slice* can now be read with one disk access. The table below shows the segment descriptors "flipped":

```

1010...
0110...
1010...
1101...
0001...
1000...
1100...
0111...
1001...
0010...

```

If we take the slices we need to check and AND them, then where a 1 bit is set in the resulting slice that segment number matches the query segment descriptor:

```

1 0 1 0 ...   AND
1 1 0 0 ...
-----
1 0 0 0 ...
-----

```

From our original configuration we know that we set k bits per term, so if a single term query is performed we need to read k bit slices to determine what segments match the query. The *bit slicing* of the segment file is the reason why Texpress has exceptional query speed.

One issue with *bit slicing* the segment descriptor file is that in order to store the bit slices sequentially we need to know the length of a bit slice. We know the length of the segment descriptor, it is b , so b bit slices are stored in the file; but what is the length of each slice? In order to determine the length of a bit slice we need to know the capacity of the database, that is how many records will be stored. Using the number of records per segment (N_r) we can calculate the number of segments required; we use the symbol N_s to represent this number. So:

$$N_s = \text{capacity} / N_r$$

Thus the length of a bit slice in bits is N_s .

False match probability

When the *false match probability* was introduced it was defined as the number of descriptors to be examined to get one false match. So a value of 1024 is interpreted as the probability of one false match every 1024 descriptors examined. Using this measure is not very useful when configuring a Texpress database because the probability is tied to the capacity of the table rather than the descriptors examined when searching. In order to address this issue the false match probabilities used by Texpress are altered to reflect the probability of a false match when searching the segment file and the probability of a false match in a segment when searching the record descriptor file. In order to make these adjustments the following formulae are used:

$$\begin{aligned} \text{Probability of a segment level false match} &= 1 / (\mathbf{f}_s * \mathbf{N}_s) \\ \text{Probability of a segment false match} &= 1 / (\mathbf{f}_r * \mathbf{N}_r) \end{aligned}$$

If we look at the first formula we can see that the probability of a false match at the segment level has changed from one every \mathbf{f}_s descriptors to one every \mathbf{f}_s searches. The change removes the number of segment descriptors from the equation. By doing so, \mathbf{f}_s is now a constant value regardless of the number of segment descriptors. A similar change was made to the false match probability for record descriptors. The number of records per segment was introduced so that the probability is now the number of *segments* examined before a false match, rather than the number of records.

We have spent a good deal of time looking at the fundamentals of the Texpress indexing mechanism. There is one variable however that requires further investigation as it plays a large part in the automatic generation of configuration values. The variable is the number of *indexed atoms* per record (\mathbf{i}_s and \mathbf{i}_r).

What is an atom?

An *atom* is a basic indexable component. Each atom corresponds to one searchable component in the Texpress indexes. What an atom is depends on the type of the data field. The table below shows for each supported data type what constitutes an atom:

float integer	A single numeric value is an <i>atom</i> .
date	Every date value consists of three components (year, month, day). Each filled component is an <i>atom</i> . For example, if a date field contains 2008/1/1, the number of atoms is three, whereas a date value of 2008/1/ has only two atoms.
time	As for dates, time values consist of three components (hour, minute, second). Each component that has a value is an <i>atom</i> . For example, a time value of 10:23:15.0 contains three atoms, whereas 10:20 contains two atoms.
latitude longitude	Latitude and longitude values consist of four components (degree, minute, second, direction). Each component with a value is an <i>atom</i> . For example, a latitude of 28:12:12.123:N consists of four atoms, whereas 28:::N consists of two atoms.
string	A string value (rarely used in Vitalware) is a text based value that is indexed as one <i>atom</i> . The data value has all punctuation removed and the resulting string forms one component. For example, a string value of "12.temp.1" would produce an atom of "12 temp 1", which is indexed as a single value. In order to retrieve string values you must enter the complete string, rather than just a word.
text	For text based data each unique word in the text constitutes an <i>atom</i> . A word is a sequence of alphabetic or numeric characters delimited by punctuation (character case is ignored). For example, a text value of "This is a text value - it contains a series of words" contains ten atoms (there are eleven words, however "a" is repeated, so the number of unique words is ten).

If a column can contain a list of values, the number of atoms is the sum of the atoms for each individual value, with duplicate atoms removed. So, if you have an integer column that accepts multiple values and the data is 10, 12, 14 and 12, the number of atoms is three (as 12 is duplicated).

The table above reflects what an *atom* is for the standard data types used by Texpress. Texpress does, however, provide extra indexing schemes that provide different searching characteristics. The table below details what constitutes an atom for each of these additional indexing schemes:

Null Indexing Null indexing provides fast searching when determining whether a column contains a value or is empty (that is for *, !*, + and !+ based wildcard searches). It is available for all data types. There is one atom per column for all columns that have null indexing enabled, regardless of whether they contain multiple values, a single value or are empty.

Partial Indexing Partial indexing provides fast searching where the search term specifies leading letters followed by wildcard characters (e.g. a*, fre?, bil[ck]*). It is available for *text* and *string* data types. An atom for partial indexing is the number of unique partial components (words for text data, the whole value for string data) for each of the partial terms. For example, consider the text value "I like lollies.". It consists of three words, namely:

- I
- like
- lollies

If we are providing partial indexing for one and three characters, the one character atoms are:

- i
- l

and the three character atoms are:

- lik
- lol

So the number of atoms for partial indexing in the above example is four. For columns that contain multiple text values the number of atoms is the sum of the atoms per text value with duplicate atoms removed.

Stem Indexing Stem indexing provides searching for all words that have the same root word. This allows users to find a word regardless of the form of the word (e.g. searching for *electric* will find *electric*, *electricity*, *electrical*, *electrics*, etc.). It is available for *text* data types. Stem based indexing uses the same mechanism as standard indexing, except that the word is

transformed into its root word before being indexed. So the number of atoms generated is similar to that for normal text based indexing (except that the number of unique atoms is lower due to many words having the same root word). So, hypothetically, the number of atoms for stem indexing is roughly the same as the number for text based indexing. In order to reduce the indexing overhead (without losing search speed) Vitalware fully indexes each stem atom (that is it sets **k** bits), but reduces the indexing for the text word to 2 extra bits per word.

Phonetic Indexing

Phonetic indexing provides searching for words that sound similar, that is they contain the same sound groups (e.g. *Steph* and *Steff* are phonetically the same). Phonetic based indexing works exactly the same as for stem based indexing except that the sound groups of a word make up the atom rather than the root word. As with stem based indexing, two extra bits are set for the text word to provide text based searching.

Phrase Indexing

Phrase indexing provides fast searching for phrased based searches, that is, searches where the query terms are enclosed within double quotes (e.g. "red house"). It is available for the *text* data type. An atom for phrased based searching is the number of adjacent double word combinations in the textual data that are not separated by an end of sentence character. For example, consider the text "I like Lollies. Do you?" The adjacent double word combinations are:

- I - like
- like - lollies
- do - you

Each of these combinations is an atom, however Vitalware only sets one bit to provide phrase based searching, rather than the normal **k** bits.

Now that we have an understanding of what an atom is, we need to look at how Texpress computes the number of atoms in a record.

Atoms per record

When explaining the configuration variables used to configure a Texpress table, the number of atoms per record (**i**) was glossed over. The working example used a value of five based on the data found in a single record. In fact, arriving at a value for the number of index terms per record can be quite involved. It also turns out that the value chosen has a large impact on the overall configuration of the system (which is to be expected as it plays an important part in the calculation of **b** (length of the descriptor in bits)). Texpress 8.2.01 has introduced changes to the indexing system that track dynamically the number of atoms per record. Using these changes Texpress can provide very good configurations regardless of the distribution of the number of atoms per record.

How does Texpress decide on the number of atoms per record? First we need to examine what is the number of atoms in a record for any given record. Based on the previous section ([What is an atom?](#)) the number of atoms in a given record consists of three numbers:

- terms** The number of *terms* for which **k** bits are set when building the descriptor. Most atoms fall into this category.
- extra** The number of *extra* atoms where two bits are set for the complete word. Extra atoms result from *stem* and *phonetic* searching.
- adjacent** The number of *adjacent* atoms where a combination of two words are indexed together resulting in one bit being set.

Let's consider an example. If we have a record with one text field with *stem* based indexing enabled and it contains the text *I like lollies do you?*, then the breakdown of atoms is:

Index Type	Count	Bits Set	Atoms
<i>terms</i>	5	k	i, like, lollies, do, you
<i>extras</i>	5	2	i, lik, lol, do, you
<i>adjacent</i>	4	1	i-like, like-lollies, lollies-do, do-you

In order to arrive at the number of atoms for the record we need to compute a weighted number of atoms based on the number of bits set. The formula is:

$$i = (\text{terms} * k + \text{extra} * 2 + \text{adjacent} * 1) / k$$

So using our example and assuming that **k=5**, the number of atoms for the sample record is $(5 * 5 + 5 * 2 + 4 * 1) / 5$ or 8 (rounded up). Since we have two values for the number of bits to set per indexed term (**k_s** and **k_r** for segment descriptors and record descriptors respectively) we end up with two weighted values for the number of atoms: one for the segment level (**i_s**) and one for the record level (**i_r**).

As you can imagine it could be quite time consuming working out the number of atoms in a record for every record in a table. Texpress makes this easy by storing the three numbers required to determine **i** with each record descriptor. When a record is inserted or updated the counts are adjusted to reflect the data stored in the record. Using **texanalyse** the atoms for each record can be viewed. The **-r** option is used to dump the raw counts:

```

texanalyse -r eparties
Terms,Extra,Adjacent,RecWeighted,SegWeighted
118,9,14,123,122
126,10,21,132,131
102,3,2,103,103
139,12,39,148,146
130,12,28,137,136
138,15,36,147,145
136,15,36,145,143
...

```

The first three columns of numbers correspond to the number of *terms*, *extra* and *adjacent* atoms. The last two numbers are the weighted number of atoms for the record descriptor level (i_r) and the segment descriptor level (i_s) respectively. These numbers are the raw input used by Texpress to determine the overall number of atoms to use for configuration.

One approach for arriving at the number of atoms to use for configuration is to take the average of the weighted number of atoms in each record. In this instance a certain percentage of records would be below the average value and the rest above. For records with the average value the bit density of the generated descriptors will be d (25% by default). For records with less than the average number of atoms, the bit density will be less than d , and for those greater than the average, the bit density will be greater than d . From our initial calculations:

$$k = \log(f) / \log(100 / d)$$

We need to maintain the bit density (d) at about 25%, given that k is fixed, otherwise the false match probability drops and false matches are more likely. We need to ensure that the vast majority of records have a bit density of 25% or less. In order to achieve this we cannot use the average number of atoms per record, rather we need to select a higher value.

If we calculate the average number of atoms per record and know the standard deviation, we can use statistical analysis to determine a value for the number of atoms that ensures that most records are below this value (and so the bit density is below 25%). If the number of atoms in each record follows a normal distribution, which is generally the case when the data comes from one source, then analysis shows that 95.4% of records will have a number of atoms value less than the average plus two times the standard deviation, and 99.7% of records will have a number of atoms value less than the average plus three times the standard deviation. Calculating the standard deviation for the number of atoms in each record could take some time so fortunately **texanalyse** can be used to determine the value. If **texanalyse** is run without options the following output is displayed:

texanalyse eparties

Analysis of Indexed Atoms per Record

=====

Atoms	Records (Rec)	Records (Seg)
93	0	1
94	1	0
95	0	3
96	3	2
97	2	0
98	3	3
99	8	8
100	0	2
101	3	1
102	1	1
103	13	15
104	2198	2223
105	966	976
106	43	290
107	288	94
...		
397	0	0
398	1	0

Record Level Analysis

=====

Total number of records : 50046
Total number of indexed terms : 6169873
Average number of indexed terms : 123.3
Standard deviation : 18.5

Records <= average : 32726 (65.4<= 123.3)
Records <= average + 1 * standard deviation: 46156 (92.2<= 141.8)
Records <= average + 2 * standard deviation: 48310 (96.5<= 160.4)
Records <= average + 3 * standard deviation: 49114 (98.1<= 178.9)
Records <= average + 4 * standard deviation: 49456 (98.8<= 197.4)
Records <= average + 5 * standard deviation: 49638 (99.2<= 215.9)
Records <= average + 6 * standard deviation: 49732 (99.4<= 234.5)
Records <= average + 7 * standard deviation: 49838 (99.6<= 253.0)
Records <= average + 8 * standard deviation: 49938 (99.8<= 271.5)
Records <= average + 9 * standard deviation: 49990 (99.9<= 290.1)
Records <= average + 10 * standard deviation: 50014 (99.9<= 308.6)
Records <= average + 11 * standard deviation: 50027 (100.0<= 327.1)
Records <= average + 12 * standard deviation: 50036 (100.0<= 345.7)
Records <= average + 13 * standard deviation: 50038 (100.0<= 364.2)
Records <= average + 14 * standard deviation: 50044 (100.0<= 382.7)
Records <= average + 15 * standard deviation: 50046 (100.0<= 401.3)

Segment Level Analysis

```

=====
Total number of records                :    50046
Total number of indexed terms          :   6112301
Average number of indexed terms        :    122.1
Standard deviation                      :     17.8

Records <= average                     :    32744 (65.4<= 122.1)
Records <= average + 1 * standard deviation:  46163 (92.2<= 139.9)
Records <= average + 2 * standard deviation:  48306 (96.5<= 157.7)
Records <= average + 3 * standard deviation:  49126 (98.2<= 175.5)
Records <= average + 4 * standard deviation:  49459 (98.8<= 193.2)
Records <= average + 5 * standard deviation:  49645 (99.2<= 211.0)
Records <= average + 6 * standard deviation:  49728 (99.4<= 228.8)
Records <= average + 7 * standard deviation:  49832 (99.6<= 246.6)
Records <= average + 8 * standard deviation:  49938 (99.8<= 264.4)
Records <= average + 9 * standard deviation:  49989 (99.9<= 282.1)
Records <= average + 10 * standard deviation:   50014 (99.9<= 299.9)
Records <= average + 11 * standard deviation:   50027 (100.0<=
317.7)
Records <= average + 12 * standard deviation:   50036 (100.0<=
335.5)
Records <= average + 13 * standard deviation:   50038 (100.0<=
353.2)
Records <= average + 14 * standard deviation:   50044 (100.0<=
371.0)
Records <= average + 15 * standard deviation:   50046 (100.0<=
388.8)

```

The table under the *Analysis of Indexed Atoms per Record* heading shows for a given number of atoms (in the *Atoms* column) how many records have that number of atoms at the record level (*Records (Rec)*) and segment level (*Records (Seg)*). The *Record Level Analysis* and *Segment Level Analysis* summaries show the average number of atoms and standard deviation for each level. The table below the standard deviation shows the number of records less than the average plus a multiple of the standard deviation. The first number in the brackets is the percentage of records below the value and the number after the equal sign is the number of atoms. For example, if you take the line:

```
Records <= average + 3 * standard deviation:    49126 (98.2<= 175.5)
```

this indicates that 49126 records are below the average plus three times the standard deviation, which represents 98.2% of the records. The number of atoms represented by the average plus three times the standard deviation is 175.5. Using this table it is possible to determine what would be good values for i_s and i_r . Remember that we want most records to be below the selected value, in most cases over 98%. Using this as a guide, for the above output a suitable value for i_r would be 180 (178.9 rounded up) and for i_s 175 (175.5 rounded down). While these may seem to be good values at first glance, it is worth considering the uppermost extremes as well. For the record level analysis the maximum number of atoms in a record is 398. If we use a value of 180 for the number of atoms at the record level, this means that the record with the maximum number of atoms sets 2.2 times the number of bits than a record with 180 atoms. If we apply this to the bit density, this corresponds to a bit density of 55% (25% * 2.2). If we are using 5 as the value for k (five bits set per atom), then the false match probability for the record with 398 atoms is:

$$(55/100)^5 \approx 0.05 \text{ or } 1/20$$

Thus the record with 398 atoms has a false match probability of 1/20 rather than the required 1/1024. In general this is an acceptable value provided there are not many records with this high probability. If, however, the bit density is over 75% for the record with the maximum number of atoms, it may be worth increasing the value of *i* so that the density is lowered (by making *i* about one third of the value of the maximum number of atoms). In general, this is only required in extreme cases and tends to arise only where a large number of data sources are loaded into one table.

The idea of adding a number of standard deviations to the average number of atoms provides us with our last variable:

- v** The number of standard deviations to add to the average number of atoms per record to determine the value of *i*, commonly called the variance. Since we have a segment and a record level value for *i* we also have a segment and record level value for *v*. The default value for the segment level is 2.0, and for the record level 3.0; that is, we add in two times the standard deviation to calculate the value of *i_s*, and three times the standard deviation to calculate *i_r*.

Summary of variables

It may be opportune at this time to summarise the variables used as part of the configuration of a Texpress database. As Texpress uses a two level scheme, there are two sets of variables, one for the segment level and one for the record level. The variables with an *r* subscript apply to record descriptors, while those with an *s* subscript are used for segment descriptors:

Variable	Description	Default value
<i>N_s</i>	Number of segment descriptors	Calculated from capacity
<i>N_r</i>	Number of record descriptors per segment	Calculated with minimum of 10
<i>k_r</i>	Bits to set per term in record descriptor	Calculated
<i>k_s</i>	Bits to set per term in segment descriptor	Calculated
<i>b_r</i>	Bit length of a record descriptor	Calculated
<i>b_s</i>	Bit length of a segment descriptor	Calculated
<i>f_r</i>	False match probability for record descriptors	1024
<i>f_s</i>	False match probability for segment descriptors	4
<i>d_r</i>	Bit density of record descriptors	25%
<i>d_s</i>	Bit density of segment descriptors	25%
<i>i_r</i>	Indexed terms per record descriptor	Calculated from data
<i>i_s</i>	Indexed terms per segment descriptor	Calculated from data
<i>v_r</i>	Number of standard deviations to add to average at record level	3.0
<i>v_s</i>	Number of standard deviations to add to average at segment level	2.0
blocksize	Filesystem read/write size in bytes	4096

Configuration tools

Texpress 8.2.01 has a number of tools that provide detailed information about the indexing mechanism. Three programs are provided, where each focuses on one aspect of system configuration:

- **texanalyse** provides information about the number of atoms per record.
- **texdensity** shows the actual bit density for each segment and record descriptor.
- **texconf** generates values for calculated configuration variables (**k**, **b**, **N_r** and **N_s**).

texanalyse

The **texanalyse** tool allows information about the number of atoms per record to be obtained. It supports both the record level and segment level. The primary use of **texanalyse** is to check that the value used for the number of atoms per record (**i**) is suitable: in particular to check the records with the maximum number of atoms per record are within an acceptable range of **i**. We define acceptable as within three times the value of **i** when a bit density (**d**) of 25% is used. It is also instructive to use **texanalyse** to produce the raw data that can be fed to spreadsheet programs for analysis. The usage message is:

```
Usage: texanalyse [-R] [-V] [-c|-r] [-s] dbname
Options are:
  -c      print analysis in CSV format
  -r      print raw data in CSV format
  -s      suppress empty rows
```

If the raw atom data per record is required, the **-r** option is used:

```
texanalyse -r eparties
Terms,Extra,Adjacent,RecWeighted,SegWeighted
118,9,14,123,122
126,10,21,132,131
102,3,2,103,103
139,12,39,148,146
...
```

The output is in CSV (comma separated values) format suitable for loading into a spreadsheet or database. The data contains the number of *term* atoms, *extra* atoms and *adjacent* atoms for each record in the table. The weighted number of atoms for the record and segment level is also given.

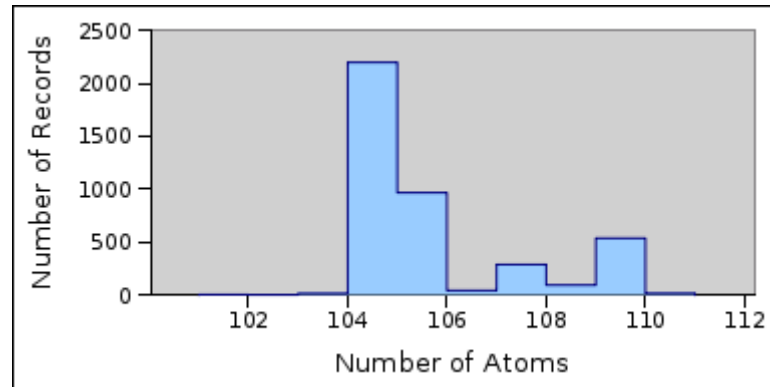
It is also possible to export in CSV format the number of records at both the record and segment levels that have a given number of atoms:


```

texanalyse -c eparties
Atoms,RecRecords,SegRecords
101,3,1
102,1,1
103,13,15
104,2198,2223
105,966,976
106,43,290
107,288,94
108,93,19
109,536,527
110,15,13
111,6,20
...

```

The output can be loaded into spreadsheet programs for analysis. The graph below shows the number of records for each atom count from the above output at the record level:



Producing a graph of the number of records with each atom count provides a useful mechanism for determining how close the distribution of the number of atoms in a record is to a normal distribution. In particular it can be used to see whether loading separate data sources has resulted in a number of normal distributions being overlaid (one per data source). The graph at the start of this article shows clearly that at least three separate data sources were loaded into the *Parties* table.

The final use for **texanalyse** is to produce a summary detailing the average number of atoms per record and the standard deviation. A list of the average plus an integral number of standard deviations is also shown:

texanalyse invoices

Analysis of Indexed Atoms per Record

=====

Atoms	Records (Rec)	Records (Seg)
49	17	17
50	0	0
...		
106	0	1

Record Level Analysis

=====

Total number of records : 1419
Total number of indexed terms : 90309
Average number of indexed terms : 63.6
Standard deviation : 9.4

Records <= average : 1170 (82.5<= 63.6)
Records <= average + 1 * standard deviation: 1244 (87.7<= 73.1)
Records <= average + 2 * standard deviation: 1313 (92.5<= 82.5)
*Records <= average + 3 * standard deviation: 1323 (93.2<= 91.9)*
Records <= average + 4 * standard deviation: 1418 (99.9<= 101.4)
Records <= average + 5 * standard deviation: 1419 (100.0<= 110.8)

Segment Level Analysis

=====

Total number of records : 1419
Total number of indexed terms : 91805
Average number of indexed terms : 64.7
Standard deviation : 9.8

Records <= average : 1167 (82.2<= 64.7)
Records <= average + 1 * standard deviation: 1243 (87.6<= 74.5)
*Records <= average + 2 * standard deviation: 1311 (92.4<= 84.3)*
Records <= average + 3 * standard deviation: 1323 (93.2<= 94.1)
Records <= average + 4 * standard deviation: 1417 (99.9<= 103.9)
Records <= average + 5 * standard deviation: 1419 (100.0<= 113.7)

The analysis tables at the end can be used to determine whether the computed number of atoms per record is suitable. The computed record level value is the average plus three times the standard deviation and the segment value is the average plus two times the standard deviation. The standard check is to ensure that the maximum number of atoms is less than three times the value chosen for **i** (assuming a bit density (**d**) of 25%).

texdensity

In order to test the effectiveness of a configuration it is useful to be able to determine the bit density for all segment and record descriptors. The **texdensity** utility provides this functionality. The usage message is:

Usage: `texdensity [-R] [-V] [[-cr|-cs] | [-dr|-ds]] [-s] [-nrn -nsn]`
 dbname

Options are:

```
-cr      print record descriptor density in CVS format
-cs      print segment descriptor density in CVS format
-dr      print record descriptor density
-ds      print segment descriptor density
-s       suppress empty values
-nrn     scan n record descriptors
-nsn     scan n segment descriptors
```

If some analysis of the bit density is required, the `-cr` or `-cs` option can be used to output in CSV format the number of bits set and the bit density per record or segment descriptor respectively:

```
texdensity -cr invoices
Index,Bits,Total,Density
0,424,2384,17.79
1,405,2384,16.99
2,404,2384,16.95
3,438,2384,18.37
...
```

The *Index* column is the record descriptor index (or segment descriptor index if `-cs` is used). The number of bits set is next, followed by the total number of bits that could be set and finally the bit density as a percentage. In general the bit density should be below the default value of 25%.

It is also possible to get the average bit density, the standard deviation and the maximum bit density:

```
texdensity -ds invoices
Segment descriptor analysis
=====

Descriptor  Bits set  Total bits  Density
+-----+-----+-----+-----+
|      0   |    1193   |    17280   |    6.90   |
|      1   |    1164   |    17280   |    6.74   |
|      2   |    1007   |    17280   |    5.83   |
|      3   |     971   |    17280   |    5.62   |
...
|    135   |         0   |    17280   |    0.00   |
+-----+-----+-----+-----+

Number of descriptors : 119
Average density       : 4.42
Standard deviation    : 1.10
Maximum density       : 10.39
```

The output above shows the summary for the segment descriptors of the **invoices** table. As the maximum density of 10.39 is well below the 25% density required, this indicates that the value for the number of atoms per record can be lowered. The computed number of atoms per record at the segment level was 83. If we lower the number of atoms per record

proportionally ($83 * 10.39 / 25$), the number of atoms to use is 34. After reconfiguring the number of atoms per record at the segment level to 34, the following density summary was found:

```
Number of descriptors : 119
Average density       : 10.45
Standard deviation    : 2.44
Maximum density       : 23.37
```

Based on this output the number could be lowered further as the average bit density is 10.45 and if we add three times the standard deviation we get 17.77 ($10.45 + 3 * 2.44$) which is still well below 25%. The reason segment descriptor bit densities are generally lower than record descriptor densities is due to the repeating of atoms in all the records that make up the segment descriptor. In particular, there are a number of fields in Vitalware that contain the same value for all records in a segment (e.g. Record Status, Publish on Internet, Publish on Intranet, Record Level Security, etc.). Since every record in the segment has the same value in these fields we should only count the atom once, however Texpress does not have any mechanism available for tracking how many atoms are repeated in a segment, so a worst case scenario is assumed where no atoms are repeated. In general this results in segment descriptor files that are larger than they need to be. Apart from using more disk space the searching mechanism is not affected overtly as *bit slices* are read rather than complete segment descriptors. It is possible to adjust the number of atoms per record at the segment level, resulting in some cases with substantial savings in disk space. The next section on setting configuration parameters details how this is set.

texconf

The **texconf** utility is a front end program to the Texpress configuration facility. Using **texconf** it is possible to alter any of the configuration variables and see the effect it has on the final configuration (that is the values of **N_r**, **N_s**, **b_r**, **b_s**, **k_r** and **k_s**). The usage message is:

```
Usage: texconf [-R] [-V] [-bn] [-cn] [-drn] [-dsn] [-frn] [-fsn] [-mn] [-
irn -isn|-nrn -nsn -vrn -vsn] dbname
Options are:
  -bn          filesystem blocksize of n bytes
  -cn          capacity of n records
  -drn         record descriptor bit density of n
  -dsn         segment descriptor bit density of n
  -frn         record descriptor false match probability of n
  -fsn         segment descriptor false match probability of n
  -mn          minimum number of records per segment of n
  -irn         record descriptor indexed terms per record of n
  -isn         segment descriptor indexed terms per record of n
  -nrn         scan n records to determine -ir value
  -nsn         scan n records to determine -is value
  -vrn         increase -ir value by n standard deviations
  -vsn         increase -is value by n standard deviations
```

A close look at the options will show that most correspond to the variables discussed in this article. Using the options you can test the effect of changing variables. Running **texconf** without any options will generate a configuration based on the default values using the current database capacity:

texconf invoices

Index Configuration

=====

Capacity of database (in records)	:	1632
Number of segments (Ns)	:	136
Number of records per segment (Nr)	:	12
Words per segment descriptor	:	540
Words per record descriptor	:	79
Bits set per indexed term (segment)	:	5
Bits set per indexed term (record)	:	7

Record Descriptor Configuration

=====

Records scanned to determine indexed terms	:	1419
Average number of indexed terms	:	63.6
Standard deviation of indexed terms	:	9.4
Standard deviations to increase average	:	3.0
Expected number of indexed terms	:	92
False match probability	:	0.000081 [1 / (1024 * Nr)]
Record descriptor tag length (bits)	:	144
Bits set per extra term	:	2
Bits set per adjacent term	:	1

Segment Descriptor Configuration

=====

Segments scanned to determine indexed terms	:	118
Average number of indexed terms	:	776.0
Standard deviation of indexed terms	:	108.8
Standard deviations to increase average	:	2.0
Expected number of indexed terms	:	994
Average number of indexed terms per record	:	64.7
Standard deviation of terms per record	:	9.1
Expected number of indexed terms per record	:	82
False match probability	:	0.001838 [1 / (4 * Ns)]
Bits set per extra term	:	2
Bits set per adjacent term	:	1

Index Sizes

=====

Segment size	:	4096
Segment descriptor file size	:	286.88K
Record descriptor file size	:	544.00K
Percent of record descriptor file wasted	:	1.56%
Total index overhead	:	830.88K

The output is broken up into four areas. The first area (*Index Configuration*) prints out the generated configuration values. These values can be entered into the Texpress configuration screen. The second area (*Record Descriptor Configuration*) details the settings used when calculating the record descriptor configuration. The third area (*Segment Descriptor Configuration*) shows the values used when calculating the segment descriptor configuration. The last area (*Index Sizes*) indicates the size of the index files required for the generated configuration. The table below shows where each of the configuration variables can be found:

Variable	Name in texconf
N_s	Number of segments (N_s)
N_r	Number of records per segment (N_r)
b_s	Words per segment descriptor (<i>multiply by 32</i>)
b_r	Words per record descriptor (<i>multiply by 32</i>)
k_s	Bits set per indexed term (segment)
k_r	Bits set per indexed term (record)
i_s	Expected number of indexed terms [Segment Descriptor Configuration]
i_r	Expected number of indexed terms [Record Descriptor Configuration]
f_s	False match probability [Segment Descriptor Configuration]
f_r	False match probability [Record Descriptor Configuration]
v_s	Standard deviations to increase average [Segment Descriptor Configuration]
v_r	Standard deviations to increase average [Record Descriptor Configuration]

Let's say that after some analysis we decide that the average number of atoms per record at the segment level should really be 34 instead of 82. We can run:

```

texconf -is34 einvoices
Index Configuration
=====
Capacity of database (in records)      :      1632
Number of segments ( $N_s$ )          :      136
Number of records per segment ( $N_r$ ) :      12
Words per segment descriptor          :      222
Words per record descriptor           :      79
Bits set per indexed term (segment)   :      5
Bits set per indexed term (record)    :      7
...
Index Sizes
=====
Segment size                          :      4096
Segment descriptor file size          :    117.94K
Record descriptor file size          :    544.00K
Percent of record descriptor file wasted :    1.56%
Total index overhead                  :    661.94K

```

Notice how the *Words per segment descriptor* value has decreased to reflect the lower number of atoms per record. Also the *Segment descriptor file size* has decreased. Using **texconf** you can determine the impact on the size of the index files when configuration variables are adjusted.

Setting configuration parameters

The final part of this document deals with setting configuration variables on a per database basis so that future configurations will use the values. The information above is all good in theory and using the configuration tools you can arrive at optimal indexes in both speed and size, but it is not much help if the next reconfiguration of the table loses all your settings. Fortunately Texpress 8.2.01 provides a database file in which you can store your

configuration settings. Settings found in this file override the default values used by Texpress.

The name of the file in which configuration parameters can be stored is called **params**. It is an optional file. The file contents are XML based and contain the configuration variables to be overridden. A complete listing of the file is (with the default values set):

```
<params>
  <configuration>
    <blocksize>4096</blocksize>
    <record>
      <atoms></atoms>
      <density>25</density>
      <falsematch>1024</falsematch>
      <variance>3.0</variance>
      <scan></scan>
    </record>
    <segment>
      <atoms></atoms>
      <density>25</density>
      <falsematch>4</falsematch>
      <variance>2.0</variance>
      <scan></scan>
      <minimum>10</minimum>
    </segment>
  </configuration>
</params>
```

The table below explains the use of each tag:

Tag	Description
blocksize	The underlying block size used by the filesystem on with the Texpress table is stored.
atoms	The number of atoms per record (i).
density	Bit density to use. Value between 1 and 99 (d).
falsematch	False match probability (f).
variance	Number of standard deviations to add to average (v).
scan	Number of records to use to calculate <i>atoms</i> .
minimum	Minimum number of records per segment.

So if you wanted to alter the number of atoms per record at the segment level to use 33, the following **params** file could be used:

```
<params>
  <configuration>
    <segment>
      <atoms>33</atoms>
    </segment>
  </configuration>
</params>
```

Values set in the **params** file are also used by **texconf**, so running **texconf** will use 33 for the number of atoms per record for the segment descriptor. You can use the **texconf** options to

override the value in the **params** file (`texconf -is40 invoices` will use a value of 40 for the number of atoms when calculating the length of the segment descriptor).

In most cases the values generated by the new configuration facility will provide near optimal indexes. In some rare cases it may be necessary to "tune" the configuration parameters to achieve savings in disk space (particularly at the segment level). If "tuning" is required, it is better to alter the **variance** value rather than specifying the number of atoms to use. The reason is as the database grows the average number of atoms per record will vary. If a **variance** is specified, the value for **atoms** can vary with it. So if we take the case of the invoices table which had an average number of atoms of 64.7 and a standard deviation of 9.1, we can set the **variance** to -3.5 (yes you can use negative variances) to get the number of indexed terms to be 32. The following **params** file could be used:

```
<params>
  <configuration>
    <segment>
      <variance>-3.5</variance>
    </segment>
  </configuration>
</params>
```

In conclusion, Texpress 8.2.01 introduces a new configuration subsystem that provides optimal indexes for the vast majority of tables. It also provides a suite of tools that can be used to check the efficiency of configurations and also generate new ones. Finally it provides a mechanism where the input variables for table configuration can be adjusted and set for future configurations.

Vitalware Documentation

Range Indexing

Document Version 1.0

Vitalware Version 2.1



Contents

SECTION 1	Range Indexing	5
	Overview	5
	How range indexing works	5
	Manual range bucket configuration	8
	vwrangupdate	10
	System Maintenance	17

Range Indexing

- [Overview](#)
- [How range indexing works](#)
- [Manual range bucket configuration](#)
- [vwrangeupdate](#)
 - [Number of range buckets](#)
 - [Bucket selection methods](#)
 - [Distribution method](#)
 - [Interval method](#)
 - [Partition method](#)
 - [Configuring vwrangeupdate](#)
- [System maintenance](#)

Overview

KE Vitalware 2.0.01 saw the addition of new indexing methods to the database engine. In particular support for NULL indexing (whether a field is empty or not) and PARTIAL indexing (fast searching for leading characters, e.g. a*) was added. Tools were provided that allowed System Administrators to configure, via the Vitalware Registry, which fields required the new indexing methods. The new indexing facilities did not provide a mechanism for adjusting or tuning range indexes.

KE Vitalware 2.1.01 provides new tools that permit System Administrators to tune the range indexing used by Vitalware. Support for automatic optimisation of range indexes has also been added. Using these tools, Vitalware can now provide optimal range indexes with significantly faster range based searches.

How range indexing works

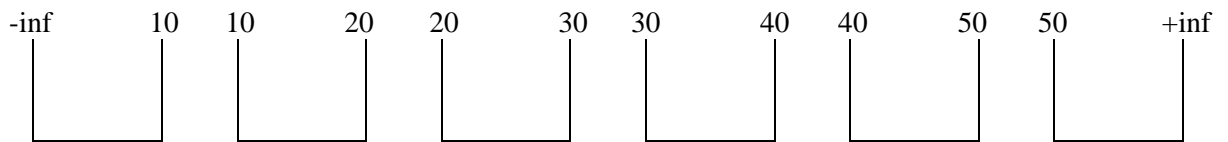
The range indexing employed by the Vitalware database server is designed to work with the *Two Level Superimposed Coding Scheme for Partial Match Retrieval* system used for all searching. As it is an extension of the standard indexing Vitalware still requires only one set of index files, thus avoiding the need for costly "join" based queries.

Range indexing is really a series of "mini" indexes on a per field basis. Unlike the *Two Level* scheme, where all search terms are placed in the one index, range terms are placed in per field indexes that are then concatenated to form one range index. Each field index consists of a number of *range buckets*. These buckets are used to indicate whether a given value falls within the bucket or not.

There are two related considerations when establishing range buckets:

1. Data distribution: as best as possible data should be distributed evenly between the buckets.
2. Logical query ranges. The aim is to minimise the necessity to check a bucket for a value as checking whether records in a bucket match the query takes time; therefore if users are likely to search on particular ranges (decades for instance: 1/1/1910 to 1/1/1920) it makes sense to configure range buckets appropriately.

An example may make things clearer. Consider *length of the child* in the Births or StillBirths module. Let's say that the length of child is generally from 10cm (for a pre-mature baby) to 50cm so we establish the following range buckets:

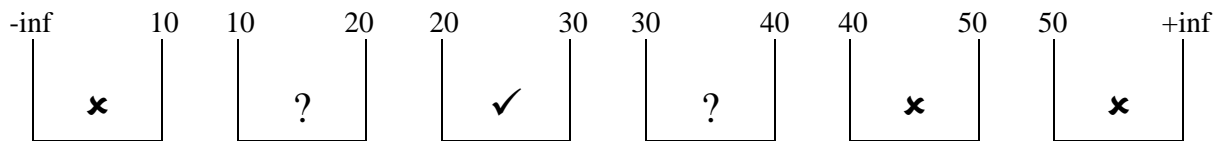


Note:

- infinity and +infinity will capture any values outside the specified ranges.

When a range search is performed these buckets are used to determine possible matches. Consider the following searches for babies with a length of between:

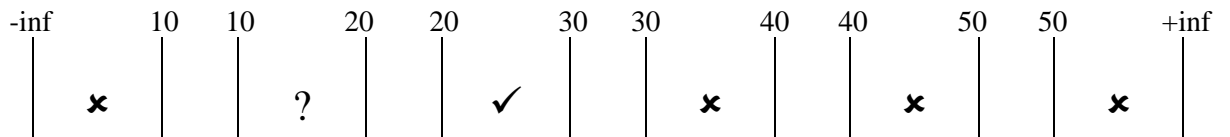
- 15cm to 35cm



In this case:

- Three buckets can be safely ignored.
- One bucket does not need to be checked and is definitely included in the search.
- Two buckets need to be checked for a match.

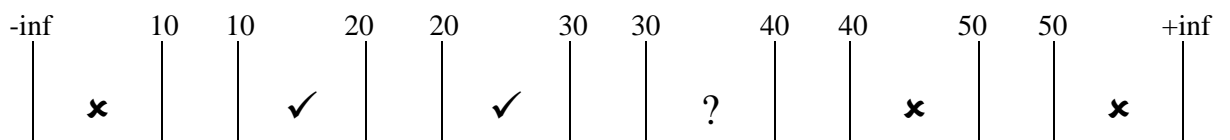
- 15cm to 30cm



In this case:

- Four buckets can be safely ignored.
- One bucket does not need to be checked and is definitely included in the search.
- One bucket needs to be checked.

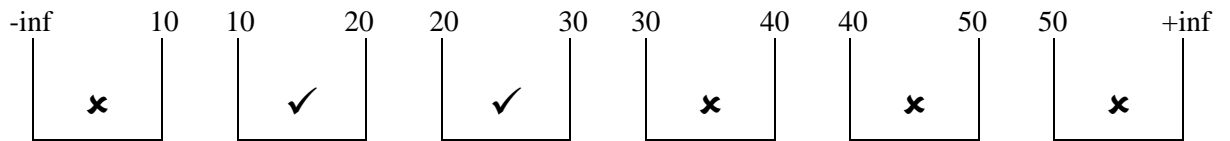
- 10cm to 35cm



In this case:

- Three buckets can be safely ignored.
- Two buckets do not need to be checked and are definitely included in the search.
- One bucket needs to be checked.

4. 10cm to 30cm



In this case:

1. Four buckets can be safely ignored.
2. Two buckets do not need to be checked and are included in the search.
3. No buckets need to be checked.

It should be clear from the last example that determining range buckets that match likely user searches has a direct influence on the speed of range based queries.

The problem with setting the range bucket values is that the bucket values depend on two variables. The first is the range of values entered into a field, in other words the *data distribution*. If the *Width* field contains a wide range of values, it makes sense to have a wide range of range buckets. If, however, the data is centred around one point, it may make sense to have a series of range buckets cover this period.

The second variable is the *query ranges* used to retrieve data. The problem with this variable is that for a given field it may be very hard to determine what sort of query ranges will be used without performing extensive analysis.

As these two variables cannot be known before data has been loaded Vitalware provides a default set of range buckets for each range searchable field. In general these buckets are satisfactory for a small numbers of records, however significant reductions in search times may be achieved by "tuning" the range buckets for large numbers of records.

Tuning the range indexes involves considering the two variables, data distribution and query ranges, and for each variable determining the set of range buckets that provides optimal performance. The objective in fine-tuning range indexing is to:

1. Minimise the number of buckets that need to be searched.
2. Minimise the number of records in buckets that need to be searched. Note that this objective will take precedence over the default distribution of records evenly between buckets.

Vitalware 2.1.01 provides a mechanism that allows System Administrators to have the range buckets tuned automatically based on the data distribution within a field. The facility does not take query ranges into account as this information requires subjective interpretation to determine which queries are important to have optimised and which queries can run a little slower.

The new facility does however provide data distribution information that may be used by a System Administrator to set the range buckets manually to achieve optimal performance where query ranges are known and can be weighted.

Hence the new facility provides automated range buckets, but allows System Administrators to override these settings and configure their own buckets manually.

Manual range bucket configuration

The setting of range bucket values is performed on a field basis. Each range indexed field may have a Registry entry that defines the range buckets for that field. If a Registry entry does not exist, the built-in values defined when the table was designed are used.

You can determine which fields have range indexing enabled by using the `vindexing` command. For example, running `vindexing eparties` will show all indexes available for the Parties module on a field by field basis:

```
Table "eparties"
  irn
      Type: Integer
      Indexing: Key
  SummaryData
      Type: Text
      Indexing: Word, Phonetic
  BioBirthEarliestDate
      Type: Date
      Indexing: Word, Range
  BioBirthLatestDate
      Type: Date
      Indexing: Word, Range
  ...
```

The word `Range` indicates fields that have range indexing enabled. Range indexing is available on fields of type:

- Integer
- Float
- Date
- Time
- Latitude
- Longitude

In general, all fields of the above types will have ranging already enabled. It is possible to enable and disable range indexing on fields of the above types via Vitalware Registry entries. The same Registry entry used to set range buckets can also "disable" ranging by setting the number of buckets to zero. Ranging is enabled by specifying one or more range bucket values on a field, provided the field is one of the above types. Range searching is not supported on *Text* or *String* based fields.

The Registry entry used to configure range buckets for a field is of the form:

```
System|Setting|Table|table|Range Buckets|colname|bucket;...
```

where *table* is the name of the Vitalware module that contains the field to be configured and *colname* is the name of the field (this can be determined by using the *What's this help?* facility in the Vitalware client, or via the `vindexing` command). The *bucket* setting is a semi-colon separated list of values to be used for range buckets. The format of the value depends on the field type.

When using the Range Buckets Registry entry it is important to make sure that the values specified are all fully qualified. In particular, full date values are required. The table below shows what constitutes a fully qualified value for each field type:

Field Type	Format	Examples
Integer	n	-10, 12, 25
Float	n.mmm	-23.0, -5, 2.125, 10
Date	yyyy-mm-dd	2003-10-23, 2008-03-17
Time	hh:mm:ss.sss	10:30:00, 18:00:00.000
Latitude	dd:mm:ss.sss:D	9:12:15.1:N,35:06:01:S
Longitude	ddd:mm:ss.sss:D	123:34:06.34:W

As an example, the Registry entry below could be used to set the range buckets on the CelRegistrationDate field in the Parties module:

```
System|Setting|Table|eparties|Range
Buckets|CelRegistrationDate|2000-01-01;2003-01-02;2006-01-
01;2009-01-01
```

Using the Range Buckets Registry entry System Administrators can set the range buckets on any range based field manually. In order to help with this configuration a tool is provided that can analyse the values in a field and provide a data distribution table, as well as suggesting suitable bucket values.

vwrangeupdate

The tool used to list, and optionally to set, suitable range buckets is called `vwrangeupdate`. It is found on the Vitalware server. To use this facility it is necessary to log in to the Vitalware server as user `vw`. The tool is used to perform a number of activities:

- print out suitable range buckets for examination
- install Vitalware Registry entries to be used when updating the indexes
- print a table of data distribution allowing manual configuration

The `vwrangeupdate` usage message is:

```
Usage: vwrangeupdate [-dip] [-qrv] [-mmin:max] [-nrecords] [[dbname][:column] ...]
```

where:

```
-d          use distribution based ranges [default]
-i          use interval based ranges
-mmin:max  minimum and maximum number of buckets to use [6:39]
-nrecords   records per bucket for distribution ranges [5000]
-p          use partition based ranges
-q          quiet mode, do not output progress
-r          update range Registry entries
-v          output data distribution table
```

It is possible to analyse anything from:

- a single column in one table
- to all columns in a table
- to a single column in all tables
- to all columns in all tables

The format used to specify a column for analysis is `dbname:column`, where `dbname` is the name of the table to be analysed and `column` is the name of the column. The following combinations are allowed:

- `dbname` - all columns in the table `dbname` will be analysed
- `:column` - column `column` in all tables will be analysed
- `dbname:column` - column `column` in table `dbname` will be analysed

Any number of the above entries may be supplied to `vwrangeupdate`. If an entry is not given, all columns in all tables are examined.

The default action is for `vwrangeupdate` to print out suitable range buckets after examining the data. The following is typical output after analysing the *Date Modified* field in the Parties table:

```

vwrangeupdate eparties:AdmDateModified
Processing eparties...
    Determining range columns...
    Checking registry entries...
    Exporting range data...
    Processing AdmDateModified...
        Range Buckets (distribution)
        =====
        2000-11-22
        2001-5-10
        2003-7-22
        2003-8-6
        2005-10-3
        2006-2-21

```

As you can see the output contains recommended range bucket values. These values could be used with the Range Buckets Registry entry to set the range buckets for the *Date Modified* field. The required Registry entry would be:

```

System|Setting|Table|eparties|Range
Buckets|AdmDateModified|2000-11-22;2001-05-10;2003-07-22;200
3-08-06;2005-10-03;2006-02-21

```

In fact it is possible to have `vwrangeupdate` add the Registry entry for you by specifying the `-r` option on the command line:

```

vwrangeupdate -r eparties:AdmDateModified
Processing eparties...
    Determining range columns...
    Checking registry entries...
    Exporting range data...
    Processing AdmDateModified...
        Range Buckets (distribution)
        =====
        2000-11-22
        2001-5-10
        2003-7-22
        2003-8-6
        2005-10-3
        2006-2-21
    Registry entry updated...

```

If you want to perform some analysis of the data, you can use the `-v` option to have a data distribution table printed:

vwrangeupdate -v eparties:AdmDateModified

Processing eparties...

Determining range columns...

Checking registry entries...

Exporting range data...

Processing AdmDateModified...

Value	Count
====	====
2000:11:22	1507
2001:5:10	1
2003:7:22	2
2003:8:6	1
2003:8:26	1
2003:9:4	2
...	
2007:12:27	1
2008:1:3	1
Distinct	74
Total	2328
Range Buckets (distribution)	
=====	
2000-11-22	
2001-5-10	
2003-7-22	
2003-8-6	
2005-10-3	
2006-2-21	

The Value column contains a sorted list of all values from the *Date Modified* field. The Count column indicates the number of occurrences of the value. At the end of the table the Distinct value provides the number of unique values and Total is the total number of values (including repeated values). With this information it is possible to perform some analysis (MS Excel may come in handy here!) and determine suitable range buckets.

Number of range buckets

When calculating the range bucket values another variable is the number of range buckets to use. The maximum number of buckets allowed is **39**. When determining the optimal number of buckets vwrangeupdate uses two pieces of information. The first is the minimum number of buckets that may be used. The default value is **6**. You can use the `-min:max` option to alter the minimum and maximum number of buckets allocated. For example:

```
vwrangeupdate -m2:15
```

will allocate a minimum of two and a maximum of fifteen buckets. To determine the number of buckets to use within this range vwrangeupdate computes the number of values in the column (the Total value from the data distribution table). It then divides this number by the number of records per bucket value (default value of 5,000) to give the number of buckets to allocate. For example, if a column has 40,000 values, 8 range buckets will be allocated. You can alter the number of records per bucket by specifying the `-nrecords` option. For example, if we use the following command:

```
vwrangeupdate -m2:15 -n2000
```

and the column contains 40,000 values, fifteen buckets are allocated ($40,000 / 2,000 = 20$; however the maximum number of buckets allowed is fifteen via the `-m2:15` option).

There are two special cases concerning the allocation of buckets. The first is for columns that do not contain any values. In this case only **one** bucket is allocated. The bucket is used to provide fast searching since any range search specified will not match the indexes; however the indexes can be used as part of the search.

The second case is where a column is part of the server table but is not used in the Vitalware client. This can occur when clients decide to sub-class standard modules and remove columns they do not require. In this case zero range buckets are allocated.

The following output shows range buckets for an existing column without data and for a column not used by the client:

```
vwrangepupdate eparties:AssStartDate0 eparties:InfoDate0
Processing eparties...
    Determining range columns...
    Checking registry entries...
    Exporting range data...
    Processing AssStartDate0...
        Range Buckets (distribution)
        =====
        1970-1-1
    Processing InfoDate0...
        Range Buckets (distribution)
        =====
```

Bucket selection methods

When determining what range buckets to use `vwrangepupdate` provides three algorithms that offer different focuses on bucket allocation. These algorithms are known as the *distribution*, *interval* and *partition* methods.

Distribution method

The *distribution* method tries to allocate the same number of values to each range bucket. The result is an even distribution of the values over the entire set of range bucket values. For example, if we have the following data distribution:

Value	Count
=====	=====
1959:1:20	1
1973:3:8	1
1974:3:8	1
1975:2:11	3
1977::	3
1977:2:8	1
1981::	1
1982:11:8	1
1983:4:12	3
1983:5:9	1
1984:9:11	1
1985::	1
1985:6:12	1
1986:2:11	1
1986:6:17	1
1986:9:	1
1988:6:28	2
1989:1:	2
1995:6:6	1
1995:9:	1
1996:2:	1
1996:2:28	4
1996:4:3	1
1996:9:17	1
1997:8:20	1
2003:10:17	1
Distinct	26
Total	37

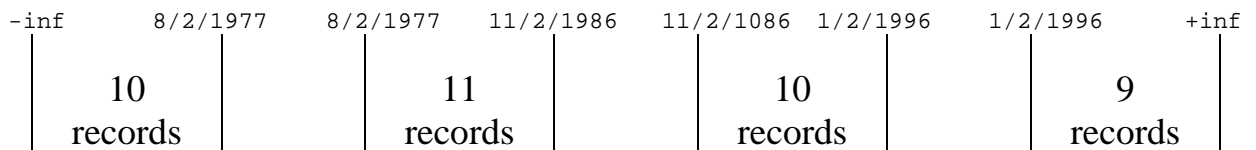
we get the following range buckets calculated:

```

Range Buckets (distribution)
=====
1977-2-8
1986-2-11
1996-2-1

```

The graphic below shows the values per bucket:



The *distribution* method provides optimal searching where the query ranges are not known in advance and where a reasonable spread of query ranges is expected. As equal numbers of records are placed in each range bucket the index will generally provide reasonable performance for any given query. The distribution method is the default method used by `emurangeupdate`. The `-d` option may be used to select this ranging method.

Interval method

The *interval* method takes a different approach to the *distribution* method. Rather than ensuring that an equal number of records is allocated to each range bucket, the *interval* method generates equal intervals between each bucket value. It does this by taking the difference between the minimum value and the maximum value and apportioning it between

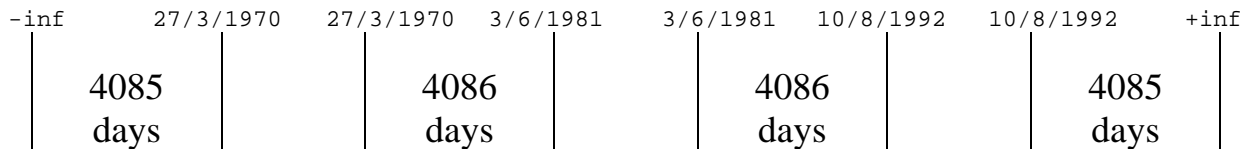
the specified number of bucket intervals. So for the data distribution provided for the *distribution* method the generated intervals are:

```

Range Buckets (interval)
=====
1970-3-27
1981-6-3
1992-8-10

```

The graphic below gives the intervals:



The advantage of the *interval* method is that it gives a decent set of range buckets when data is not available for a given field (generally because it has not been entered or imported). It also provides buckets that may be more in line with query ranges that correspond to fixed intervals (e.g. year ranges for date searches, hour intervals for time searches, etc.) and so gives better performance where query range intervals are commonly used. Interval ranges can be generated by specifying the `-i` option to `vwrangepupdate`.

Partition method

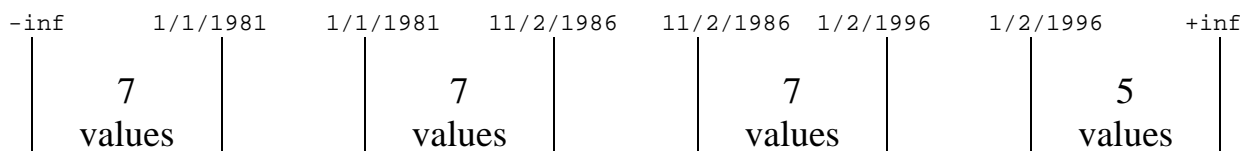
The *partition* method is similar to the *distribution* method, however rather than ensuring that equal numbers of records are in each range bucket it ensures that equal numbers of *unique* values are in each range bucket (if ten values are the same, they are distributed as a single value). For the data distribution provided for the *distribution* method the generated partitions are:

```

Range Buckets (partition)
=====
1981-1-1
1986-2-11
1996-2-1

```

The graphic below shows the number of distinct values for each range bucket:



The *partition* method is useful when a good distribution of buckets is required that takes into account the values that have already been entered without giving undue influence to the weighting of each value (as is the case with the *distribution* method). In general this method will provide good "all-round" performance as it gives sensible range intervals based on the data already entered. Partition ranges can be generated by specifying the `-p` option to `vwrangepupdate`.

Configuring `vwrangepupdate`

When `vwrangepupdate` is invoked it uses the Vitalware Registry to look for hints as to how many range buckets should be allocated and what type of distribution method should be used for any given column. Administrators may set Registry entries that can enable range searching on columns that do not have range support currently. It is also possible to disable range searching if a column does not require this type of indexing.

The format of the Registry entry is:

```
System|Setting|Table|table|Range Index|colname;...
```

where *table* is the name of the module containing the columns to be set. The *colname* setting is a list of semi-colon separated column names, listing the columns for which range searching is to be enabled (cf. `Null Index` and `Partial Index` Registry entries). It is possible to provide hints about the range index for the column by appending to the column name the number of range buckets required and the distribution method to be used. The format of the entry is:

```
colname=number:method
```

where *number* is the number of range buckets to allocate for the given *colname* and *method* is the distribution method to use. Allowable *method* values are:

- `distribution`
- `interval`
- `partition`

corresponding to the methods described above. A *number* setting of zero (0) will disable range indexing for the given field.

For example, if the *Date Modified* field is searched regularly using fairly small query ranges (e.g. monthly), you may want to increase the number of range buckets allocated and use the *partition* distribution method. You may also want to disable range indexing for the *Time Modified* field as users do not use it for range based searches (note that disabling range indexing **does not** mean you cannot perform range searches on a column, it just means that the search cannot use any indexing information to provide faster searching). The following Registry entry can be used to reflect these changes:

```
System|Setting|Table|eparties|Range  
Index|AdmDateModified=15:partition;AdmTimeModified=0
```

Notice how all range column settings are specified in the one Registry entry. While this may seem confusing it is consistent with the `Null Index`, `Partial Index`, `Stem Index` and `Phonetic Index` Registry entries.

If you only want to change the distribution method for a column, you need not specify the number of buckets. For example, the entry:

```
System|Setting|Table|eparties|Range  
Index|AdmDateModified=:partition
```

will ensure the *partition* method is used for the *Date Modified* field.

System Maintenance

The `vwrangepdate` utility has been designed not only to suggest suitable range buckets for range indexing, but also to update/create Registry entries that will result in the new range buckets being implemented the next time the database indexes are rebuilt (generally on the weekend).

Using the `-r` option with `vwrangepdate` will result in the creation of Range Buckets Registry entries reflecting the suggestions made by the utility. The Registry entries created differ slightly from the standard Registry entry in that the list of bucket values is prefixed with the string `auto:`. The string is used to indicate that the entry was made by `vwrangepdate` rather than by a System Administrator. Only Registry entries with the `auto:` prefix are updated by `vwrangepdate`; all other entries (that is, those created by other means) are not changed. This means that System Administrators may add their own entries manually and they will be preserved by `vwrangepdate`.

It is recommended that `vwrangepdate` is run on a regular basis to ensure that optimal range searching is available. The easiest way to provide automated configurations is to add an entry to user `vw`'s crontab (via `vwcron`), similar to:

```
#
# Calculate range buckets (first day of each month)
#
0 0 1 * * ${VWPATH}/bin/vwrun vwrangepdate -r 2>&1 | ${VWPATH}/bin/vw
run vwlogger -t "KE Vitalware Range Update Report" reindex
```

You may consider adding other options (e.g. `-m3:39` or `-p`) as required.