Rubicon

Reference Manual

For version 4.0.3.1

href
tools corp.

Disclaimer

HREF TOOLS CORP. ("HREF") DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON INFRINGEMENT. IN NO EVENT SHALL HREF BE LIABLE FOR ANY LOSS OR DAMAGE OF ANY KIND, INCLUDING BUT NOT LIMITED TO INCIDENTAL, INDIRECT, CONSEQUENTIAL OR SPECIAL DAMAGES, ARISING OUT OF THIS AGREEMENT OF THE DELIVERY, USE, SUPPORT OR OPERATION OF THE SOFTWARE. AMONG OTHER THINGS HREF WILL NOT BE LIABLE FOR ANY LOSS OR DAMAGE INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES. IN NO EVENT WILL HREF BE LIABLE FOR ANY DAMAGES IN EXCESS OF HREF's LIST PRICE FOR A LICENSE TO THE PROGRAM EVEN IF HREF SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. FURTHERMORE, SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS LIMITATION AND EXCLUSION MAY NOT APPLY TO YOU.

Trademarks

Rubicon is a trademark of HREF Tools Corp. Delphi and C++ Builder are trademarks of Embarcadero Software.

Copyright

Credits

Rubicon versions 1 and 2 were programmed by Deven Hickingbotham at Tamarack Associates, Palo Alta, California. Versions 3 and 4 were implemented by HREF Tools Corp. The text in this manual pertaining to Rubicon 1 and 2 was written by D. Hickingbotham. This manual was typeset by Ran Zhang.

Edition

28-May-2013 for Rubicon version 4.0.3.1

Buy Now!

Please pay for your Rubicon license. Start at http://www.href.com/rubicon

Fonts

This manual uses the Berkeley Book font from www.adobe.com for the main text, and Hadrianus from Scriptorium for Rubicon.

http://href.com

Contents

EVALUATION

In this chapter, we consider information that will influence your selection, acquisition and purchase of Rubicon; we advertise benefits and guide your choice.

Introduction

Rubicon has been designed to be easy to use by the end user, perform full text searches as quickly as possible, and to be compatible with a wide range storage and retrieval options.

End users have grown accustomed to the easy-to-use search interfaces offered by internet search sites. With Rubicon it is simple to offer this same style of interface whereby the user simply has to type in words. The end user also has the option of using a more sophisticated search using phrases, wildcards, applying And, Or, Not, Near, and Like logic, and/or iteratively narrowing or widening the search. The search results are returned faster than a conventional search regardless of field type or the location of the word or string in the field.

Search results are typically returned in under one tenth of a second, beating conventional search strategies by more than a thousand fold. This speed allows the host system to run more efficiently and also allows the end user more time to refine the search or explore other search avenues.

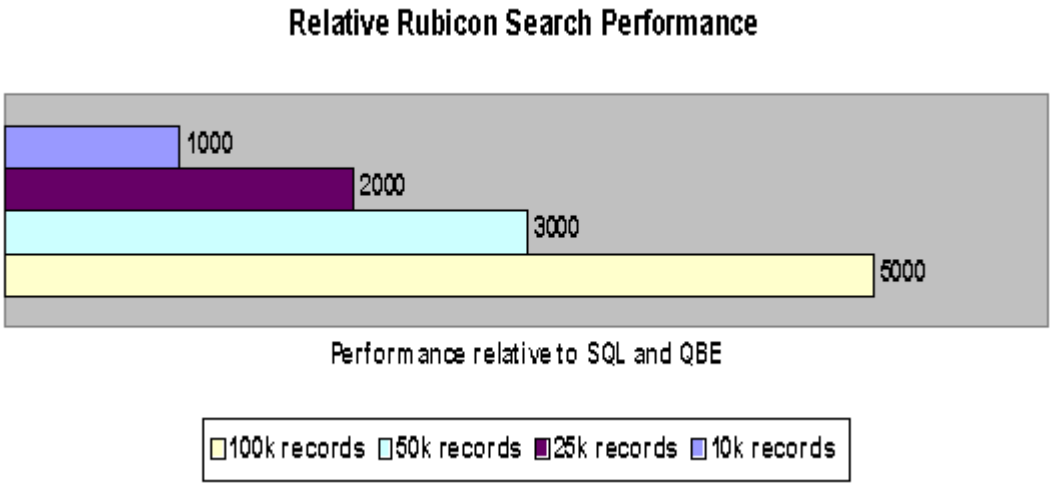
Rubicon's design allows it to work with more types of text. In many cases, the text being searched is stored in database tables, and Rubicon is compatible with many leading database engines.

From a developer's standpoint, Rubicon encapsulates this robust search technology in a set of native Delphi VCL components that build indexes, update indexes, and execute searches. All the components are entirely written in Delphi and are compatible with all versions of Delphi and C++ Builder.

Rubicon performs all searches by building a single Rubicon table that indexes all the words in the source text and the locations of the words. This means that most Rubicon searches never read the text – an important consideration in secure or resource constrained environments. Reads and writes against Rubicon index are further minimized by built-in caching and compression technology. For a stand alone user's perspective, this speeds up the entire search process. For network managers, this means that the search minimizes the use of network bandwidth and both client and server CPU cycles. And by building Rubicon indexes during off peak hours, the network manager can further free up precious peak period network resources.

How Fast Is It?

Rubicon is going to be significantly faster than conventional search strategies in all cases except when very small amounts of text are being searched. The largest performance gains are achieved against large amounts of text using complex searches. Appearing below are some relative database performance benchmarks using local tables and memo fields:



Thus a Rubicon search averages 1,000 times faster than a SQL or QBE search against a 10k record table, 2,000 times faster against a 25k record table, etc. As the number of records increase, conventional search times will rise linearly. However, Rubicon searches are generally not affected by the number of records in the table being searched, and therefore the relative advantage of a Rubicon search will continue to grow as the number of records increases.

The same kinds of performance gains can be expected when using remote tables. If the remote table is on a file server, then along with the performance improvement comes significantly lower network traffic. The same is often true for SQL tables since many queries against a memo field require that the data be brought back to the client for processing. And even if the query can be processed on the SQL server, Rubicon can process the search more efficiently on the client with minimal network traffic.

Keep in mind that SQL memo field searches are usually case sensitive (the Upper function does not convert memo fields in InterBase or BDE supported local tables). This means that SQL searches will have to test for various case combinations in order to return the same result set as Rubicon whose searches are not case sensitive.

What is new in Rubicon 4?

Rubicon 4 supports Unicode content within your databases. Rubicon 4 adds support for

64-bit compilation.

Rubicon 4.01 includes significantly refactored demo applications for the full range of database bridges.

Rubicon 4.01 includes a precompiled utility to handle the task of downloading and building sample databases.

Release notes are maintained at *www.href.com/rbrelnotes*.

What was new in Rubicon 3?

Rubicon 3 is the first edition to compile with a Unicode version of RAD Studio. The list of supported database bridges is slightly different, but basically all functionality within the Rubicon components is the same as before.

Customers upgrading from should find it extremely easy to recompile their Rubicon 2 projects in a current version of Delphi or C++Builder with Rubicon 3.

The Hunt4Packages utility is new.

The licensing scheme for Lite and Evaluation users is new.

What was new in Rubicon 2?

Rubicon has been almost completely overhauled in Version 2. The changes included:

•Abstracting all data access so that potentially any source of text could serve as input and providing wide flexibility for storage of index information.
•Implemented a standard TDataSet based set of classes from which Borland and third party datasets can be derived from. Rubicon includes drivers for several leading third party database engines, so now Rubicon can read and write more formats than ever!
•Fully supports TQuery based drivers for much more efficient access to SQL databases.
•Match tables support TClientDataSet and third party database engine formats.
•Improved international character support.
•Significantly improved compression reduces the memory and disk footprint or lets you process more text with the same amount of resources.
•Optimized several key routines by rewriting them in assembler or enhancing the algorithms.
•Ability to store index information in the database record by using an encoded string field. This significantly reduces the number of records containing blob data and results in faster read and write times.
•Shared caching.
•Units reorganized to enhance readability and extensibility.

Architecture

The design goal of the Rubicon search engine is to execute searches as fast as possible, minimize the use of system resources, and require little or no training of the end user. Rubicon is able to quickly and efficiently perform searches by pre-building or compiling indexes of all the words and their locations. Search results therefore become mostly a matter of manipulating these indexes rather than actually reading the source text.

Text, Words and Locations

Rubicon reads text and creates location indexes for each unique word. The text being indexed can be from any source. All that is needed is a driver to deliver the text to Rubicon and a way to track where the text is located within the source.

The location, which must be a unique integer value, is not the precise position of the word in the text, but rather the position of the word in a logical area of text. Typically, a location is a primary or unique-secondary-key in a database table, a paragraph number, or a file number.

Locations are presumed to be efficient. Efficient locations are ones that have few gaps between locations. For instance, Text that consists of locations 10001, 10002, 10003, etc., is efficient whereas locations 10, 20, 30, etc. is inefficient.

The process of actually reading the text and saving the indexes is described in the next section.

Terminology: throughout this documentation, **Text** refers to the text being indexed, while **Words** (uppercase W) refers to both the words and their index of locations. Creating Words refers to the process of creating storage for the Words.

TrbTextLink & TrbWordsLink

Rubicon 2 introduced the concept of links to the text being searched and the word indexes being created. These links are VCL components that act as the interface between Rubicon and the data storage.

The TrbTextLink and *TrbWordsLink* components are very similar to the virtual *TDataSet* introduced in Delphi 3; their purpose is to define a base class from which descendents are derived that actually implement the code that performs a storage task.

Terminology: TrbTextLink and TrbWordsLink will often be referred to as links or drivers.

TrbMake, TrbUpdate & TrbSearch

The core of Rubicon is comprised of three components: *TrbMake*, *TrbUpdate*, and *TrbSearch*. The first two components create and maintain the word indexes which are stored in Words. *TrbMake* reads

the entire Text and builds a new Words. Subsequent changes to the Text may be processed through *TrbUpdate* in order to keep Words current. The advantages and disadvantages of making or rebuilding versus updating indexes are discussed in more detail in the next section.

Searches are performed with the *TrbSearch* component. Properties are available that control what to *SearchFor*, the *SearchLogic* (*slAnd*, *slOr*, *slNot*, etc.), the *SearchMode* (to begin a new search or to iteratively widen or narrow the scope of a search), and the *RankMode* (*rmNone*, *rmCount*, *rmPercent*).

TrbSearch may be used by multiple simultaneous users to search the same Text and Words. *TrbMake* is inherently a single user tool since it has to read the entire Text and create Words. *TrbUpdate* is a single user or batch oriented component. Components designed for multiple simultaneous updates in are discussed later.

TrbMake versus TrbUpdate

Appearing below is a table summarizing the relative merits of *TrbMake* versus *TrbUpdate*.

Table 1: TrbMake vs. TrbUpdate

	TrbMake	TrbUpdate
Purpose	Processing large amount of text	Process one update at a time
Memory Usage	High	Low Caching increases usage
Speed Per Location	Fast – memory based	Slow – disk based Caching improves performance
Speed To Update One Location	Slow – must process all the text	Fast – only processes one update at a time
Speed To Update Many Locations	May be faster than update depending on the scope of changes	May be faster than make depending on the scope of changes
Speed To Update All Locations	Fastest	Much slower
Location Restrictions	None	See driver specific limitations

If the Text is updated in a batch mode (e.g. overnight updates), then *TrbMake* is probably the appropriate choice if the number of changes is substantial.

If the Text is continually updated and the end user needs to be able to locate even the most recent changes, then *TrbUpdate* must be used.

Keep in mind that careful Text design can minimize or eliminate the need for updating the dictionary. For instance, a parts database may consist of descriptions, inventories, and orders. While the inventories and orders portions of the database are going to be subject to frequent updates, the parts descriptions are probably relatively static. Thus, if the dictio-

nary is created just on part descriptions and this is in a separate table, then the need to dynamically update the dictionary is minimized.

Multiple User Updates

When multiple users are editing Text, *TrbUpdate* is not well suited to updating the Words. *TrbUpdate* uses caching to improve its performance, but in a multi-user environment the data held in cache memory could become out of sync. In order to safely use *TrbUpdate*, caching would have to be disabled, and this would drive up network usage as each word changed would require a read and a write of Words.

TrbClientUpdate and *TrbServerUpdate* provide a multi-user solution to updating the Words. Instead of writing updates directly to Words, all the clients use *TrbClientUpdate* to write to the *NetDataSet*. One application using *TrbServerUpdate* performs the updates to the Words by processing the pending changes in the *NetDataSet*. Since only *TrbServerUpdate* needs to cache records, it maintains a single coherent cache. This approach significantly reduces network traffic.

Note: Rubicon includes a ready to run server application, Server.dpr.

Segmentation

When *TrbMake* indexes text, it creates the indexes in memory and only writes them to Words when the indexing is complete. By using index compression and assuming a reasonable number of unique words, this task can be performed on most PCs.

However, when indexing very large amounts of text, or when the number of unique words if very high, the indexing task has to be broken down into smaller logical pieces or segments.

Segments are relative to locations. If the lowest location in Text is 10,000, and the highest location is 50,000, then there are 40,001 possible locations (offsets 0..40,000). The range of locations in this case is 40,000 (highest location minus lowest location). Without segmentation, all 40,001 locations would be processed at once. With segmentation, the task could be divided into pieces where each piece indexed N locations at a time.

Note: each location does not have to have text associated with it, although having a large number of 'empty' locations is inefficient.

Segmentation is enabled by setting the *SegmentSize* property in *TrbMake*. When enabled, the number of segments in Words will be the range of locations divided by the *SegmentSize*. In the example above, this would be (40,000 / 16,000), or two full segments and one partially full segment, for a total of three segments.

Segmentation increases the size of Words because a word can appear in more than one segment. It also results in slightly longer search times as more time has to be spent assembling the segments.

These drawbacks are usually negligible when compared to the reduced time spent updating. Updating segmented Words means only a segment of the index needs updating, not

the entire index. As a result, less memory is used and less data is read and written.

The update components can only work on and cache results from one segment at a time. If only new locations are being added to the Text, this works very well as all the updates will be processed in the last segment. However, when edits, deletes, or insertions are performed that cross segment boundaries, the components are forced to flush the cache and effectively restart in the new segment. If the pattern of usage results in many boundary changes, performance will suffer.

There is no clear rule as to when segmentation should be used. The number of locations being indexed, the efficiency of the locations, the number of unique words, the memory available, and the pattern and type of updates being performed are all factors in this decision.

Searching

Rubicon supports several types of searches. The default behavior is to find the locations which contain all the words entered in the search. For example, a search for 'personal computer' would find all the locations that contain the words 'personal' and 'computer'.

The *TrbSearch* property that determines the type of search is *SearchLogic*. The default value for *SearchLogic* is *slAnd*. To search for locations containing 'personal' or 'computer', set *SearchLogic* to *slOr*. To search for locations that contain neither 'personal' nor 'computer', set *SearchLogic* to *slNot*.

Searches are not case sensitive. Words are converted to uppercase using the *TrbEngine UpperCase* procedure.

Note: Rubicon uses an UpperCase procedure which differs from the SysUtils.UpperCase function.

Like Searches

Like searches find words that match or nearly match those words entered in the query. The default matching algorithm used is a slightly modified SoundEx routine. This algorithm may be replaced with a custom algorithm. To perform a like search, set the *TrbSearch SearchLogic* property to *slLike*.

Changing the Scope of a Search

After an initial search is performed, a user may wish to broaden or narrow the scope of a search. To control the scope of a search, set the *TrbSearch SearchMode* property to *smSearch* (default), *smWiden*, or *smNarrow*.

Wildcard Searches

Rubicon supports wildcard searches in all operations. Using a *SearchLogic* of *slAnd*, a search for 'person* comput*' would find locations that contain an instance of both 'person*' and 'comput*'. To determine which words were matched, use the *MatchingWords* method.

Proximity Searches

Searching for phrases or searching for one word near another word is a proximity search. In order to keep the size of the word indexes to a minimum, Rubicon does not store proximity information. As a result, when a proximity search is performed, Rubicon first narrows the list of possible locations using an *slAnd* search, and then reads the text of the eligible locations to determine whether they meet the search criteria. To use a proximity search, set the *TrbSearch SearchLogic* property to *slPhrase* or *slNear*. Whether two words are near one another is determined by the *TrbSearch NearWord* property.

Expression Evaluation

Rubicon supports expression evaluation in searches by setting the *SearchLogic* property to *slExpression*. Using *slExpression*, searches may take the form of:

```
windows
like windows
windows and driver and not video
windows near driver or "sound card"
(window* and driver) or (sound and card?)
```

slExpression allows the use of these familiar operators that are evaluated in the following precedence (highest appear first)

```
like, near
not
and, or
```

The syntax for these operators is

```
like <string>
<string> near <string>
not <expression>
<expression> or <expression>
<expression> and <expression>
```

where

`<string>` is a string with or without a wildcard (e.g. windows, window*)

`<expression>` is a `<string>`, another operator, or parentheses enclosing an expression

Appearing below are some common mistakes:

Common Search Mistakes

Mistake	Solution
(windows or driver) near video	Windows near video or driver near video
like (problem or corruption)	Like problem or like corruption
like 'delphi'	Like delphi
like "borland delphi"	None
windows or driver not video	Windows or driver and not video
windows driver	Windows and driver Use slSmart SearchLogic

Other common errors include not matching quotes (which may be paired single or double quotes) or parenthesis.When there is a syntax error, *ErrorPos* contains the approximate location of the error.

The following expressions are equivalent:

windows and driver near video	windows and (driver near video)
windows or driver and not video	(windows or driver) and not video
windows and driver and video	((windows and driver) and video)

The expression evaluator does not attempt to optimize the expression. This only becomes significant with searches using NEAR or phrases because these searches require reading the Text.

Ranking Search Results

Search results may be ranked by setting the *TrbSearch RankMode* property to *rmNone* (default), *rmCount*, or *rmPercent*. Ranking is similar to proximity searches in that Rubicon does not store the number of times a word appears in a location. To calculate the number of times a word(s) appears at a location, Rubicon must read the text for the location and count the matching words.

The *rmPercent RankMode* calculates the relative ranking (100 being highest) of a record relative to all the other matching records. This means that requesting a single location's rank requires all the locations to be counted and ranked.

For performance reasons, ranking is best used when the number of locations matching the search criteria is small. When there are a large number of matching locations, the user should be asked to forgo ranking or asked to be more specific in their search.

Databases and Tables

The core Rubicon components are not tied to databases or tables. As a result, the discussion up to this point has tried to treat Text and Words as abstract containers. As a practical matter, however, Text and Words are almost always tables residing in a database.

When working with tables, the concept of location translates to a record (a logical piece of text), and the location itself being an index field (e.g. CustNo).

Reading and writing of data is handled by the *TrbTextLink* and *TrbWordsLink* components. Much of the driver functionality is contained in the *TrbTextDataSetLink* and *TrbWordsDataSetLink* classes. As their names imply, they are entirely based on the *TDataSet* class. This makes for easy extensibility to Borland VCL controls like *TTable* and *TQuery*, as well as to third party virtual dataset descendents.

With Rubicon 2 and Rubicon 3, tables may be accessed via a TQuery which is often preferred when working with an SQL database. TQuery requires the BDE. Other, non-bde bridges are query based and operate with a query component specific to the vendor, for example, TIBOQuery or TADQuery.

Rubicon 4.031+ includes an extra multi-table query bridge which works with any query component that descends from TDataSet. This can be used when you want to index data in multiple tables using a database system which does not implement a "view" feature, or when your database bridge is table- not query-based. When your database system can implement a "view," you can use the name of the saved "view" as your tablename.

Rubicon components *TrbMake* and *TrbMatchMaker* create tables. This may require that the user be given table creation privileges.

Borland/DevCo/CodeGear/Embarcadero Drivers

Rubicon comes with native drivers for access to tables using IBExpress, ADO, BDE, dbExpress and FireDAC (AnyDAC). For each of these, there is a driver for the Text and a driver for the Words index. The naming convention for the drivers is *TCustomTextXxxxLink* and *TrbCustomWordsXxxxLink*, where Xxxx is Table, Query, etc.

The *TTable* based drivers are best suited for local tables, while the *TQuery* based drivers are geared for SQL compliant databases.

Third Party Drivers

Drivers have been written for many popular third party database engines. For more the most up to date information on these drivers, including installation instructions, please visit *www.href.com/rbnotes*.

When a vendor's database engine supports more than one table format, testing is conducted on the most popular format. If you experience a problem using a drive using a table format other than the primary one, you may wish to conduct some tests with the pri-

mary format before contacting HREF Tools.

Fields

Records typically have multiple fields, one of which is the index field with provides the location of the record, and one or more fields that contain the text to be indexed. When processing the record, Rubicon treats the text as coming from one location and is unaware of any field distinction.

This means that searches identify the record satisfying the search criteria, not the individual field. This is generally a plus as the user need not distinguish between fields when entering a search.

However, there are times when a search needs to be field specific, in which case there are two options: use the *SubFieldNames* property or build a Words table for a specific field(s). The *SubFieldNames* approach works well in most cases except when the search is likely to bring back a large result set. Creating a separate Words table improves search performance, but increases the maintenance task as more tables have to be updated.

Note: Proximity searches do respect field boundaries.

Transactions

Rubicon components are transaction aware. The components that write data publish a *Transactions* property that, when set, usually results in significantly faster write performance. Rubicon is not aware of how your application may be dealing with transactions, so if the application is explicitly controlling transactions, Rubicon transaction handling should be disabled in order to avoid collisions.

Utility Programs

Eight (8) utility programs are included with Rubicon, for data management: *Verify, AdoVerify*, *Compare*, *Optimize*, *Convert*, *EditProp*, *Accept* and *Server*.

Use the **Verify** utility to check the integrity of Words. If *Verify* reports any errors, Words should be rebuilt. (Use **AdoVerify** with ADO.)

Use the **Compare** utility to compare two Words. Generally, two Words will only pass the *Compare* tests when they are exactly the same. Words that have different table types may pass the test if all the words consist of standard characters (international characters may be treated differently by the table types and therefore cause differences).

Use the **Optimize** utility to determine the optimal size for *BlobFieldSize*, *BytesFieldSize*, or *CharFieldSize*. An important factor in determining the optimal size of these fields is the number of overhead bytes per record of blob storage. This value is entered in the Overhead edit box. After pressing the Optimize button, the optimal field size will be reported. Note that 'optimal' is the minimum table size, not optimal performance. All the table sizes reported are estimates and do not include any table overhead or slack space other than blob overhead.

After using *Optimize*, you may use **Convert** to change the structure of Words. This is most useful for changing the values of *BlobFieldSize*, *BytesFieldSize*, *CharFieldSize*, and the value of *SegmentSize*. You may also convert Words from the segmented structure to a non-segmented structure.

EditProp may be used to view and modify the properties stored in Words. Generally, properties should not be changed since any change will not be reflected in the words and indexes.

Use **Accept** to test various combinations of *TrbAccept* properties against a Words table. The program will show how many words were rejected.

The 8th utility is the **Server** utility.

Setup Programs

The following programs install to the `Rubicon\Setup` folder.

Use **RbcResetSampleData** to download sample data files for use with the demos. This program uses a freeware utility named `url2file` to dowload the relevant files from *http://data.rubicon.href.com/* and to save them to a local folder, `Rubicon\RBSampleData`, for you.

If you need to make path adjustments and rebuild your packages, run the BAT file in the `Setup` folder that starts **ZMAdmin.exe**. ZMAdmin is a configuration maintenance utility, and when run from the provided BAT file, it loads a custom panel for Rubicon configuration. You can toggle database bridges off and add you can customize their search paths. This may be necessary when a vendor updates their libraries after you have installed Rubicon. After making any such changes, and keeping Delphi closed, run three BAT files in sequence to launch **RbcAssistant.exe** with suitable parameters for rebuilding your packages and (re)installing them to the IDE. Of course, you can not activate a Rubicon database bridge that is not installed; you would have to re-run the Rubicon Setup to achieve that.

Use **Hunt4Packages.exe** to quickly search your disk(s) in relevant places for a particular package (if you see the Fatal Required Package Not Found error).

FREE Rubicon Editions and Their Limits

Evaluation Edition

The FREE evaluation version of Rubicon is ideal for anyone who needs to test database bridges other than ADO/FireDAC/IBExpress/dbExpress. There should be an evaluation edition for the two most recent Embarcadero compilers at *http://www.href.com/pub/rubicon*.

These are the differences between the Eval edition and the Paid edition:

- Source code is not included
- You must generate an unlock code for yourself at http://www.href.com/unlock
- You must use packages when compiling the demos, the utilities and your own test projects. (See: Operations chapter.)
- There is a time limit, after which the components will revert to Unregistered Lite mode.
- The Delphi IDE must be running for many, but not all, operations.

Lite Edition

The Lite edition uses the same files as the evaluation edition. The difference is that you do not need to unlock it (i.e. you do not need to tell us your email address). The lite edition is suitable for small, personal applications, education/training situations.

The limitations to the lite edition are:

- Source code is not included
- You must use one of the database bridges with native support within RAD Studio: ADO/FireDAC/IBExpress/dbExpress.
- You must compile with packages.
- The Words table may not contain more than 1500 words.
- There is a minimum delay of 1.5 seconds per search.*
- A small yellow form floats on the screen during operation.*

* The limits marked with an asterix can be removed immediately if you ask for a FREE Registered Lite license at http://www.href.com/unlock now.

Verifying Your License and its Limitations

After installing, you can look at the AboutLicense property on TrbSearch and other components in the Delphi IDE. You can also check the information at runtime with code such as the following:

```
procedure TForm1.Button1Click(Sender: TObject);
var
      x: TrbSearch;

begin
      x := nil;
      try
            x := TrbSearch.Create(Self);
            ShowMessage(x.AboutLicense); // License info shown here
      finally
            FreeAndNil(x);
      end;
end;
```

Rubicon End User License Agreement

This End User License Agreement ("EULA") contains the terms and conditions regarding your use of the SOFTWARE (as defined below). This EULA contains material limitations to your rights in that regard.

0. SOFTWARE. Rubicon Pro version 3 in one of three modes: registered (paid), evaluation (free with timeout), or lite (free with limited features).

1. LICENSE. HREF Tools Corp. ("HREF") hereby grants to you, the undersigned, and you accept, a nonexclusive, nontransferable license to install, use, and display the Licensed Software on unlimited computers, subject to restrictions which follow. You may use the Licensed Software for the purpose of developing your own software applications, in accordance with the terms of Section 2, below. A copy of the Licensed Software may be made for archival or backup purposes as long as it contains all the original Licensed Software proprietary notices. You may not sublicense, rent, distribute, lease or otherwise transfer or assign any or all of your rights in the Licensed Software. You may not distribute the Licensed Software as a stand-alone product. You may not reverse assemble, reverse compile, disassemble, or in any way reverse engineer the Licensed Software. You may not translate the Licensed Software without written permission from HREF. This license does not grant you any right to bug fixes, enhancements, updates or new versions, but if such are made available to and are obtained by you, then they shall become part of the Licensed Software and governed by the terms of this License. HREF reserves all rights not expressly granted to you in this License.

Evaluation/Lite users may not change, alter or modify the Licensed Software.

2. CREATING APPLICATIONS. Your rights to create and distribute your own software applications that use the Licensed Software as a runtime component ("Applications") is based on Rubicon being designated a ROYALTY-FREE product.

2(a). ROYALTY-FREE LICENSES. The following terms apply:

You may freely distribute your own Applications that use Licensed Software as a runtime component without payment to HREF, if and only if the Licensed Software used by such Applications is not marked as a Free Evaluation Version, and the Applications: (a) contain no modifications to the Licensed Software (including alterations to the original proprietary notices); and (b) are in compiled, executable form; and (c) do not provide substantially the same functionality as the Licensed Software or have as one of their purposes to build other software that would compete with the Licensed Software; and (d) do not reproduce or distribute any portion of the documentation for the Licensed Software or document the Applications in a manner that identifies the programmatic interface to the callable routines in the Licensed Software; and (e) are subject to a license agreement that (i)limits Applications end-users use of the Licensed Software to a run-time component, (ii) restricts the Applications end-user from changing, altering or modifying the Licensed Software, creating derivative works, translations, reverse assembling, reverse compiling, disassembling, or in any way reverse engineering the Licensed Software, and (iii) prevents the Applications end-user

from sublicensing, renting, distributing, leasing or otherwise transferring or assigning any portion of the Licensed Software other than as specifically permitted in this Section 2, you may not create any derivative works of the Licensed Software.

2(b). EVALUATION LICENSES. If the product you have downloaded or otherwise obtained is unlocked with an EVALUTION license, the following terms apply: you may install one copy of the Licensed Software for development and testing purposes until the date stated in the unlock code (typically 14 calendar days) ("Evaluation Time"). Upon expiration of the Evaluation Time, the Licensed Software must be erased from the computer it was installed on and all copies destroyed. Under no circumstances should evaluation software be used for commercial purposes. Evaluation software may contain mechanisms that inhibit its ability to function at a later date.

HREF offers evaluation unlock codes at http://www.href.com/unlock in exchange for a valid email address.

2(c). LITE LICENSES. If the product you have downloaded or otherwise obtained is unlocked with a LITE license, your usage is governed by 2(a) and you must respect the feature limitations of the LITE edition.

2(d). FILES MARKED WITH ALTERNATE LICENSES. At HREF's sole discretion, the SOFTWARE may be distributed with files which are individually licensed under Creative Commons, "MIT", or other generous licenses which carry fewer restrictions than 2(a). Any such files are clearly marked with a licensing comment at the beginning of the file. You may use any such individual file according to the information in that file (for example, under Creative Commons, you may distribute that file as long as you keep the credits intact).

3. RIGHTS IN LICENSED SOFTWARE. You acknowledge that the Licensed Software and any copies, regardless of the form or media in which the original or copies may exist, are the sole and exclusive property of HREF; by accepting this License, you do not become the owner of the Licensed Software recorded on the media. You further acknowledge that the Licensed Software, including the code, logic and structure of the Licensed Software, contain valuable trade secrets belonging to HREF. You agree to secure and protect the Licensed Software consistent with the maintenance of HREF's rights in the Licensed Software, as set forth in this License. You agree that HREF can use the information provided during the purchase of the Licensed Software to deliver and confirm your purchase, in the marketing or promotion of the Licensed Software, or for other relevant purposes, as well as contact you again about other products, services, or offers.

4. THIRD PARTIES. You acknowledge and agree that the Licensed Software may be used to connect to or integrate with software and other technology owned and controlled by third parties. In order to connect to or integrate with any and all other such third party software or technology you may be subject to a license agreement with that third party. You acknowledge and agree that you will look solely to the applicable third party and not to HREF to enforce any of your rights with regard to such third party software or technology.

5. COPIES. The Licensed Software is copyrighted under the laws of the United States and

international treaty provisions. Notwithstanding the copyright, the Licensed Software contains trade secrets and confidential information of HREF. You agree not to disclose or otherwise make available any part of the Licensed Software to any third party on any basis, other than as set forth in Section 2. You agree not to distribute any copies of the documentation of the Licensed Software.

6. TERM. This License shall be perpetual unless you fail to observe any of its terms, in which case it shall terminate immediately, and without additional prior notice, provided, however, that copies of the run-time component of the Licensed Software that are part of the Applications licensed to third parties may be retained by such licensed third parties in accordance with this Agreement. Upon termination or expiration of this Agreement, you shall return the original and all copies, complete or partial, of the Licensed Software to HREF, and shall not access such media for the purpose of recovering any of the Licensed Software from any copies that may exist with respect to media containing regular backups of your computer or computer system. The terms of Sections 3, 4, 5, 7, 8, 9, 10 and 11 shall survive termination of this Agreement.

7. DISCLAIMER OF WARRANTY. THE LICENSED SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. FURTHER, HREF SPECIFICALLY DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE LICENSED SOFTWARE OR WRITTEN MATERIALS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY HREF OR ITS EMPLOYEES SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY, AND YOU MAY NOT RELY ON ANY SUCH INFORMATION OR ADVICE.

8. LIMITATION ON LIABILITY. The Licensed Software may produce inaccurate results because of a failure or inaccuracy in the performance of the software, because you input incorrect data, or for many other reasons. You assume full and sole responsibility for any use you make of the output from the Licensed Software, and you bear the entire risks of there being an error in the output. You agree that regardless of the cause of any error or the form of any claim, YOUR SOLE REMEDY AND HREF'S SOLE OBLIGATION SHALL BE GOVERNED BY THIS AGREEMENT AND IN NO EVENT SHALL HREF'S LIABILITY EXCEED THE PRICE PAID TO HREF FOR THE LICENSED SOFTWARE. YOU EXPRESSLY AGREE THAT IN NO EVENT SHALL HREF BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES ARISING FROM THIRD PARTY SOFTWARE OR TECHNOLOGY, BREACH OF WARRANTY, BREACH OF CONTRACT, NEGLIGENCE, OR ANY OTHER LEGAL THEORY, WHETHER IN TORT OR CONTRACT, EVEN IF HREF HAS BEEN APPRAISED OF THE LIKELIHOOD OF SUCH DAMAGES OCCURRING, INCLUDING WITHOUT LIMITATION DAMAGES FROM INTERRUPTION OF BUSINESS, LOSS OF USE OF SOFTWARE, LOSS OF DATA, COST OF RECREATING DATA, COST OF CAPITAL, COST OF ANY SUBSTITUTE SOFTWARE, OR LOSSES CAUSED BY DELAY. HREF shall not be responsible for any damages or expenses

resulting from alteration or unauthorized use of the Licensed Software, or from the unintended and unforeseen results obtained by you resulting from such use.

9. INDEMNIFICATION. You hereby agree to indemnify HREF and its officers, directors, employees, agents, and representatives from each and every demand, claim, loss, liability, or damage of any kind, including actual attorneys fees, whether in tort or contract, that it or any of them may incur by reason of, or arising out of, any claim which is made by any third party with respect any breach or violation of this Agreement by you or any claims based on the Applications and the Licensed Software included therein.

10. U.S. GOVERNMENT RESTRICTED RIGHTS. The Licensed Software is Commercial Computer Software provided with RESTRICTED RIGHTS under Federal Acquisition Regulations and agency supplements to them. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subsection (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFAR 255.227-7013 et. seq. or 252.211-7015, or subsections (a) through (d) of the Commercial Computer Software Restricted Rights at FAR 52.227-19, as applicable, or similar clauses in the NASA FAR Supplement. Contractor/manufacturer is HREF Tools Corp., 1275 Fourth Street #109, Santa Rosa, CA 95404.

11. EXPORT CONTROLS. None of the Licensed Software, or underlying information may be exported, directly or indirectly, without the prior written consent, if required, by the office of Export Administration of the United States, Department of Commerce, nor to any country to which the U.S. has embargoed goods, to any person on the U.S. Treasury Department's list of Specially Designated Nations or the U.S. Commerce Department's Table of Denials. By consenting to this License you warrant that you are not located in, under the control of, or a national or resident of any such country or appear on any such list and further warrant that you will not distribute the run-time version of the Licensed Software to any entity that is located in, under the control of, or a national or resident of any such country or appears on any such list.

12. ENTIRE AGREEMENT. YOU ACKNOWLEDGE THAT YOU HAVE READ THIS LICENSE, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS LICENSE IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN HREF AND YOU, WHICH SUPERSEDES ANY PROPOSAL, PRIOR AGREEMENT, OR LICENSE, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS RELATING TO THE SUBJECT MATTER OF THIS LICENSE. This License shall be construed in accordance with the internal laws of California and all disputes shall have exclusive venue in the federal and state courts in Sonoma County, California, and both parties consent to the jurisdiction of these courts. If any term of this License shall be found invalid, the term shall be modified or omitted to the extent necessary, and the remainder of the License shall continue in full effect.

By using the SOFTWARE, you agree to this EULA. If you do not agree, immediately return this product for a refund.

Glossary

Bridge
Same as a driver.

Dictionary
Alternative name for Words. More commonly used in Rubicon 1.

Driver
VCL component that provides the interface between a core Rubicon component and a storage device (usually a database)

Efficient index
Rubicon keeps track of each word in the text by using a position or location value. An efficient index is one where each position in a given range is uses or has very few gaps. An inefficient index has large gaps between locations which results in wasted memory and disk space.

IndexMode
The manner in which *TrbTextDataSetLink* identifies a record's location. Rubicon 2 and 3 alway assumes this is done by using an *IndexFieldName*.

Link
Same as a driver.

Location
An ordinal value used to uniquely identify a group of text.

Match table
The table created with *TrbMatchMaker*. Does not have to be a physical table. Instead, it could be a *TClientDataSet*.

Ordinal
A SmallInt, Word, Integer, or LongInt field type.

Search table
The table (Text) to be searched.

Segmented
When working with large amounts of text, or a huge number of unique words, it may be necessary to logically divide the indexing task across several logical segments. When this is done, the Words are considered segmented.

Target table
The table (Text) to be searched.

Text
The text being indexed, usually in the form of a table.

Text table
Same as above.

TextLink
Set this property to the text driver being used to access the Text.

Words
A compilation of the unique words and their indexes, usually in the form of a table.

Words table
Same as above.

WordsLink
Set this property to the words driver being used to access the Words.

PLANNING

In this chapter, we cover activities that can be done prior to downloading and installing the software.

System Requirements

Rubicon requires 32-bit or 64-bit Windows, Delphi or C++ Builder, and approximately 15mb of hard disk space. Rubicon 4 fully supports Delphi 2009+ with Unicode strings.

Common Questions

Q: Does Rubicon handle memo fields?

Yes, Rubicon handles all standard field types. Memo fields are limited to 64k in 16 bit applications. Nonstandard fields such as ftBlob, ftVarBytes, and 16 bit memo fields exceeding 64k can be handled via the OnProcessField event handler.

Q: Can the same cache be used to cache indexes from different Words tables?

No, the TrbCache component can only cache indexes from one Words table.

Q: How can I expand acronyms?

Use the OnProcessField event.

Q: Does the Words table or Match table have to be the same table type as the table being searched?

No. For instance, when indexing a dBase table, you may want to use a Paradox Words table in order to take advantage of the binary data field. In a client/server situation, you may not want to create a Match table on the server and instead create it on the client side using a local table type.

Q: When using the BDE, which local table format is best for the Words table?

Paradox... since it natively supports the ftBytes field type (see BytesFieldSize) and has very little overhead for blob fields.

Q: Does the AltMemMgr option replace the standard Delphi memory manager?

No, it merely supplements the standard memory manager during GetMem and FreeMem

calls, and only is used for cache memory.

Q: Can binary field (BytesFieldSize) be used with InterBase tables?

No. Despite the fact that InterBase supports the ftBytes field type, testing shows that Rubicon is not compatible with the way this field type has been implemented.

Paradox and dBase Options

When using the *TrbCustomTextBDELink* derived components with Paradox and dBase tables, the *IndexFieldName* property need not be set. As explained below, there are many advantages to using the *IndexFieldName* property, but in some situations it can be safely avoided.

When the *IndexFieldName* property is not set, Rubicon uses the Paradox sequence number or the dBase record number as the location. The main drawback to record based locations is that they restrict the ways updates can be performed and can dictate which index is used.

Using the *IndexFieldName* property has the following advantages:

- Best choice when a primary or unique secondary key consisting of a single ordinal field is available
- Available for all table types
- Scales easily to SQL
- Most compatible with dynamic updating
- Works with some filters (see *Filters* and Ranges)
- When *IndexFieldName* is not used for dBase tables:
- Updates may only consist only of appends and edits allowed, deletions allowed only if the table is not packed, and no insertions are permitted
- No range limits may be in place
- Works poorly with filters
- Table must be open in natural (record number) order
- Packing the table after the creation of the dictionary requires a rebuild of the dictionary

When *IndexFieldName* is not used for Paradox tables:

- Updates may only consist only of appends and edits, deletions and insertions are not per-

mitted

- Does not support filters or ranges
- During searches and updates, the table must use the same index that was in place when processed by TrbMake

Rubicon 1: A blank IndexFieldName property has the same affect as setting the IndexMode to imRecordNo or imSeqNo for dBase and Paradox tables, respectively.

Component Hierarcy

See also: Docs\Model.zip for the model files for Rubicon. If you do not have this file, re-install Rubicon, and be sure to select the Model.zip option under Documentation.

Third party drivers are not included in the hierarchy, but usually descend from *TrbText-DataSetLink* and *TrbWordsDataSetLink*.

```
TObject
  TPersistent
    TComponent
      TrbBase
        TrbMatchMaker
        TrbTextLink
          TrbTextDataSetLink
            TrbTextBaseTableLink
              TrbCustomTextTableLink
                TrbMakeTextTableLink
                TrbTextTableLink
            TrbTextSQLLink
              TrbCustomTextQueryLink
                TrbMakeTextQueryLink
                TrbTextQueryLink
        TrbWordsLink
```

TrbWordsDataSetLink

TrbWordsBaseTableLink

TrbCustomWordsTableLink

TrbMakeWordsTableLink

TrbWordsTableLink

TrbWordsSQLLink

TrbCustomWordsQueryLink

TrbMakeWordsQueryLink

TrbWordsQueryLink

TrbBaseCache

TrbCustomCache

TrbCache

TrbEngine

TrbCustomMake

TrbMake

TrbCustomConvert

TrbConvert

TrbReadEngine

TrbBaseUpdate

TrbCustomUpdate

TrbUpdate

TrbServerUpdate

TrbClientUpdate

TrbBasicSearch

TrbModeSearch

TrbWildSearch

TrbProximitySearch

TrbLogicSearch

TrbNavSearch

TrbRankSearch

TrbSearch

TDataSet

TDBDataSet

TTable

TrbTable

TrbProgressDialog

TrbController

TrbUpdateDialog

TWinControl

TCustomControl

TCustomGrid

TDrawGrid

TStringGrid

TrbHints

TCustomEdit

TCustomMemo

TCustomRichEdit

TrbRichEdit

INSTALLATION

Download

When you purchase a Rubicon license, HREF Tools Customer Service will email you instructions for downloading the Rubicon Setup file (the installer). You should end up with a file with a name such as `Rubicon_v4.031_Setup.exe`.

Alternatively you may download a FREE Lite or Evaluation edition of Rubicon.

The FREE **Lite Setup** is recommended for obtaining a quick overview, using the latest Delphi or C++Builder compiler and the four database bridges supported by Embarcadero (ADO, dbExpress, FireDAC and IBExpress). This edition is intentionally is limited and simpler to use. The Lite Setup is generally available in Code Central, at http://cc.embarcadero.com/.

The **Evaluation Setup** is also FREE; installers are published for Delphi 2009+ at *http://www.href.com/pub/rubicon/*. You can find the exact download location by starting at the Rubicon product information page which is *http://www.href.com/rubicon*.

Unusual Features of the Installer

The Rubicon Setup program includes some unusual features:

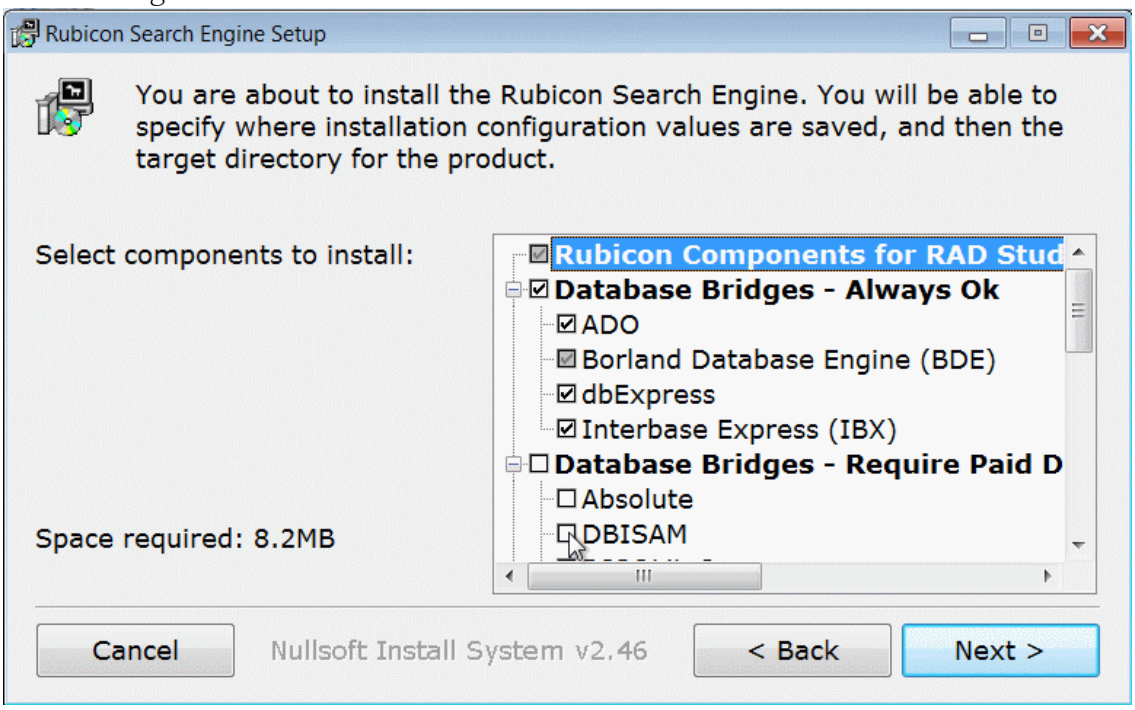
- It uses ZaphodsMap to remember your configuration settings, instead of the registry. More information about ZaphodsMap is available at *www.ZaphodsMap.com* and in the CodeRage Presentation Archive, searchable at *http://www.codenewsfast.com*; search for keyword "zaphodsmap". ZaphodsMap is an open-source cross-platform configuration system licensed under Creative Commons and maintained by HREF Tools Corp.

- it writes custom BAT files and runs them in order to compile all packages. These BAT files run in big black scary DOS windows, but they are quite okay. They will leave a .log file documenting what they compiled and any fatal errors. Look for log files under `Rubicon\Packages\*.log`.
NB: for evaluation and lite users, there is no need nor ability to recompile the core Rubicon packages.

- By default, the installer will delete any old source, package and utility files prior to installing, so that you can confidently run the installer repeatedly (without uninstalling in between).

Running Setup

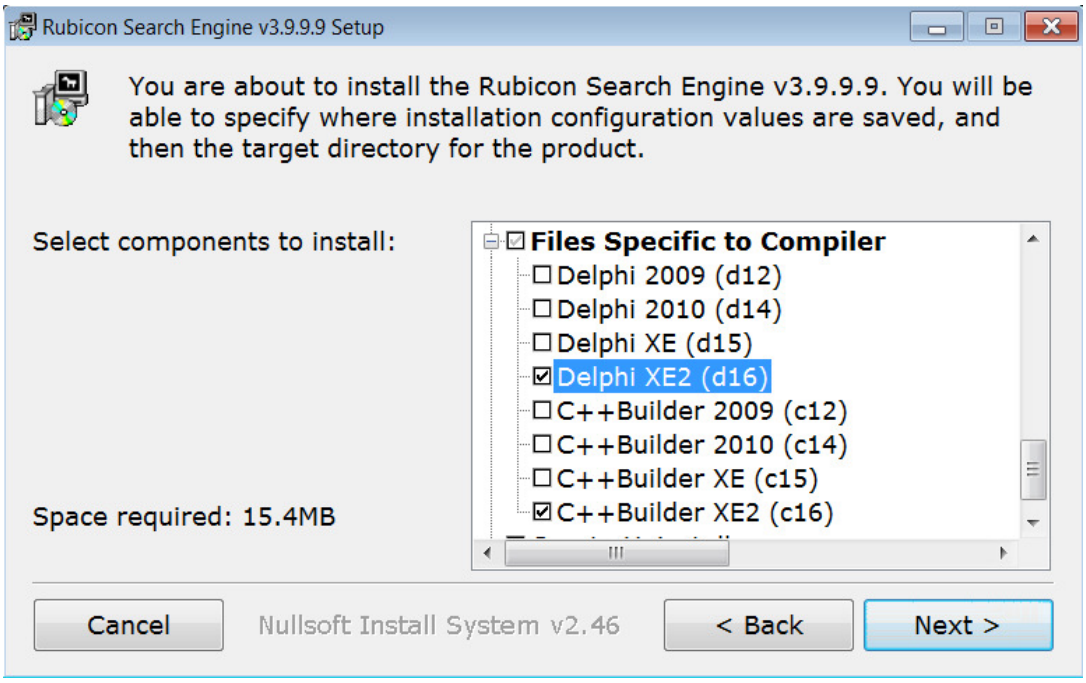
When you run the Setup program, you will be shown a license agreement and a readme file. On the next page, you will be able to select product components to install, as shown in the following screenshot.



Setup: Select Components to Install

Be sure to scroll down to see all options. The list of database bridges from third parties (i.e. other than Embarcadero) are available in the Paid and Evaluation Setup programs.

The list of compilers is at the end. You may install files for Delphi and C++Builder at the same time if you wish.



Setup: Select Compiler(s)

Errors about Third-Party Bridges

Based on your choice of compilers and database bridges, a utility named `RbcAssistant` will generate a series of dccdpk.bat files to compile all necessary packages during setup. There are three steps to this:

1. Write the dccdpk.bat files under `Rubicon\Packages\pkg_d17_win32` etc.

2. Run the dccdpk.bat files and display any errors

3. Install the packages to the RAD Studio IDE

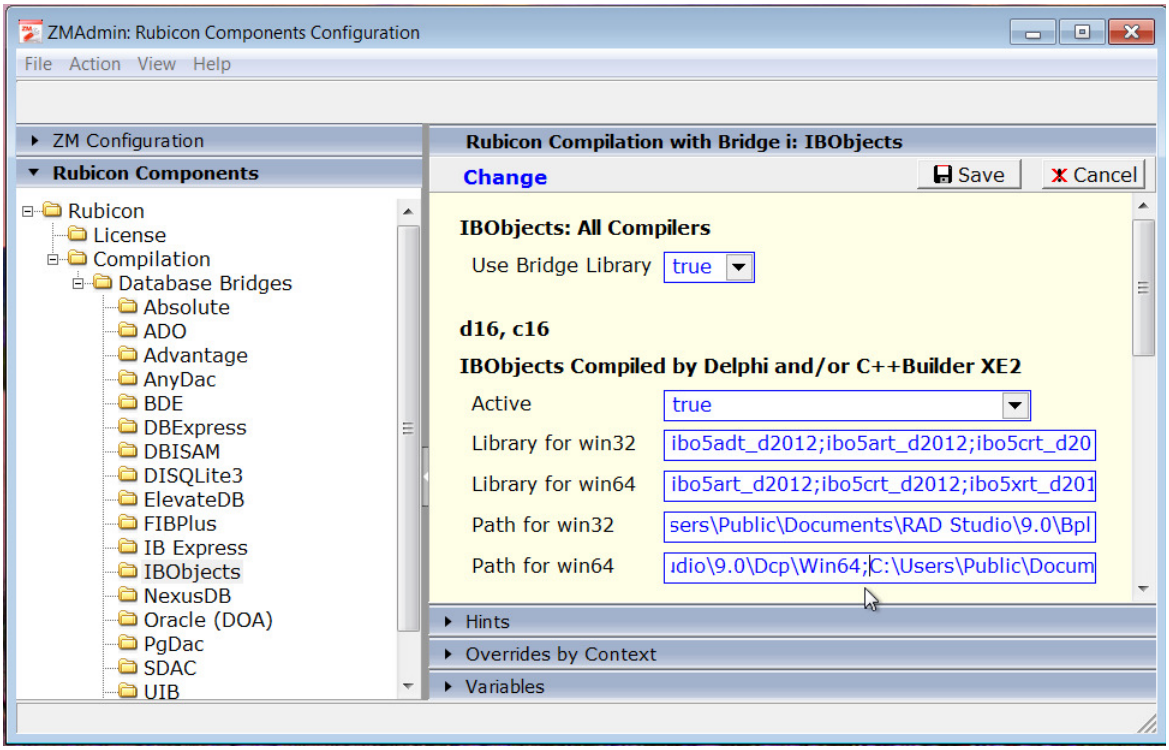
Errors are generally caused by:

a) the third party files are actually missing from disk (forgot to install them first)

b) Rubicon does not know where the files are

If (a), just install the third party product and then re-run the Rubicon Setup.

If (b), follow the advice in the error log file. When you **Start > All Programs > HREF Tools > Rubicon > ZMAdmin**, you will be able to enter the correct search path(s) for your bridges as shown in the upcoming screen shot. After correcting the paths, you need to repeat steps 1-2-3. You can run them from the `Rubicon\Setup` folder or via **Start > All Programs > HREF Tools > Rubicon** shortcuts.

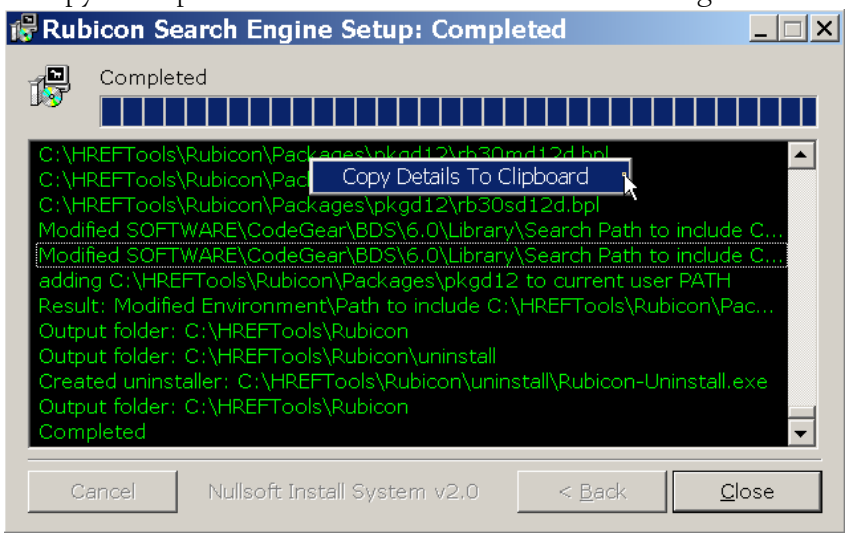


Use ZMAdmin for Rubicon to Correct Library Names and Paths

Detailed Installation Log Available

Just before the Rubicon Setup finishes - before you [Close] the dialog - you have the option of displaying a detailed log. This log contains information which will be very helpful in troubleshooting any installation problems you may experience. To save it, right-click to

bring up the “Copy to Clipboard” feature as shown in the following screenshot.



Setup Print Details

Extra Sample Database Files

As of Rubicon v4.0.1, we are no longer using subversion to maintain the archive of sample database files. Instead a normal http address is used, with no login, and a new utility is provided to make it very easy to get started.

Just run `Setup\RbcResetSampleData.exe`. You should do this before trying to run any of the demo programs.

If you wish to see the files with a web browser, visit *http://data.rubicon.href.com/RBSampleData*.

Third Party Drivers

Please note that changes made by third party vendors to their components after the release of Rubicon may require minor changes in the related Rubicon packages and/or recompilation of the packages.

In general, if you wish to recompile your Rubicon packages, exit Delphi, run **ZMAdmin for Rubicon** to correct any settings, and then run all three steps in `Rubicon\Setup` to (1) create `dccdpk.bat`; (2) run `dccdpk.bat`, and (3) install the resulting packages into the Delphi Integrated Development Environment (“IDE”).

C++ Builder Package Installation

Please remember to select C++Builder as a compiler during installation if you want packages installed for it.

Files will be installed to

Packages\pkg_c18_win32	for C++Builder XE4
Packages\pkg_c17_win32	for C++Builder XE3
Packages\pkg_c17_win64	runtime only, for C++Builder XE3
Packages\pkg_c16_win32	for C++Builder XE2
Packages\pkg_c16_win64	runtime only, for C++Builder XE2
Packages\pkg_c15_win32	for C++Builder XE
Packages\pkg_c14_win32	for C++Builder 2010

Package Naming Conventions

The package naming convention is as follows:

rbc40_B_LNG_UUU_NNN_WIN32.ext

- 40 is the Rubicon version number
- B represents the database bridge (the driver) and is detailed below
- LNG is 'pas' for Delphi/Pascal or 'cpp' for C++Builder
- UUU is 'lib' for runtime library or 'ide' for integrated design environment
- NNN is the target compiler and is detailed below, e.g. d17 for Delphi XE3
- ext is the appropriate extension (DCP or BPL)

Table: Database Bridge codes used in Rubicon Package Filenames

Bridge For	Code
Advantage Database	A
Borland Database Engine	B
SDAC	C
DBISAM	D
Interbase Express	E
FIBPlux	F
UIB	G
Interbase Objects	I
Apollo	J
AnyDAC (FireDAC)	K
ElevateDB	L
ADO	M
NexusDB	N
Direct Oracle Access	O
PgDAC	P
DISQLite3	Q
Rubicon Core	R
dbExpress	S
Absolute Database	T
UniDAC	U

Table: Compiler codes used in Rubicon Package Filenames

Compiler	Code
Delphi 2010	d14
Delphi XE	d15
Delphi XE2	d16
Delphi XE3	d17
Delphi XE4	d18
C++Builder 2010	c14
C++Builder XE	c15
C++Builder XE2	c16
C++Builder XE3	c17
C++Builder XE4	c18

Demo/Example Programs

The `Rubicon\RBDemos` subdirectory contains examples for each database bridge. Use the program `Rubicon\Setup\RbcResetSampleData.exe` to download sample data, or browse for files at *http://data.rubicon.href.com/RBSampleData/*

Primary Example Programs

These examples are provided for ALL database bridges:

ExMake:
Builds a Words table. Uses the TrbProgressDialog component. This example must be run before any of the other examples!

ExSrch:
Perform searches and creates a match dataset. Try using different property values for SearchLogic and RankMode.

ExUpd:
Uses TrbUpdate and TrbTable to perform updates to the table. Try making a change, then click the 'Words' and see the change.

ExAppend:
Example of TrbAppend.

Secondary Example Programs

These examples are provided for selected bridges. Many of these were writtenfor the BDE

and assume a local “table” or “briefcase” model database.

ExNav:
Demonstrates the use of the FindFirst, FindNext, FindPrior, and FindLast methods.

ExFltr:
Uses the Matches method to filter the records. 32 bit only.

ExRange:
Demonstrates how to perform searches when a range or filter is in place.

ExQuery:
Shows how to combine the full text search capabilities of Rubicon with a TQuery that operates on numeric/date fields.

ExCntrlr:
Example of TrbController.

ExClient:
This is a network version of ExUpd. Before running ExClient, start the Server application (look in the Utils folder), press the Process button, then start ExClient and try making the same change as described in ExUpd.

ExHints:
Demonstrates the use of the TrbHints component.

ExHtml:
Example of how to search external files – in this case HTML files.

ExRTF:
Similar to ExHTML, but processes text and RTF files.

ExSgmnt:
Shows how two TrbMake’s can build a single Words table using the FirstSegment and LastSegment properties. See comments in the Button1Click procedure. This example creates a Words table identical to the one created in ExMake. Segmentation is necessary when working with extremely large dictionaries.

Resource Definition

In this chapter we cover making adjustments in the environment or associated technology needed to accommodate Rubicon.

Hint for Evaluation and Lite Editions

When you compile the demos, utilities, or your own projects, you MUST **compile with packages**.

For example, your runtime package list might look like this when compiling the ExMake demo with IB Express (bridge "E"):

```
rbc40_core_free_pas_lib;rbc40_core_pas_lib;rbc40_e_pas_lib;
```

When compiling with the FireDAC (AnyDAC) bridge "K" it would be:

```
rbc40_core_free_pas_lib;rbc40_core_pas_lib;rbc40_k_pas_lib;
```

If you do not compile with packages, Delphi will complain that it can not find files such as rbdefine.inc. Purchase a license to obtain full object pascal source.

See page 49 for detailed instructions.

Non-English Text; UTF-8; Unicode

For most of Rubicon's history, Rubicon was used to index content in English plus a few Western-European languages, all using Ansi Code Page 1252.

Rubicon v2.134 was the first edition to let you take advantage of alternate character sets.

Rubicon v4.0 fully supports Unicode content.

Enable Indexing of Unicode Content

As of Rubicon v4, you can index Unicode content by setting one flag and defining your alphabet. The flag that you must set is

```
rbMake1.Ansi := False;
```

Customize the Alphabet Symbols

To define your "alphabet" to be other than English A..Z, set the following property:

```
rbAccept1.Alpha := 'ABC...'; // use your alphabet symbols here
```

If you wish to generate the alphabet symbols programmatically, refer to this small Code Central project #28624 (full Delphi source), *http://cc.embarcadero.com/Item/28624*

IB Express

Be sure to set the UTF8 character set on your database connection component:

```
IBDatabase1.Params.Values['lc_ctype']:='UTF8';
```

Note that even though you set the character type to UTF8, your data, in Delphi, will be UnicodeString not UTF8String. The .AsString method will give you a UnicodeString.

Compiling Third-Party Data Bridges

Packages for all requested data bridges *should* install and compile automatically for you...

It is absolutely necessary to install third-party database components before Rubicon. For example, if you are using NexusDB, make sure that you install NexusDB before installing Rubicon, or else re-run the Rubicon installer again after installing NexusDB.

If a compilation error is reported when you run the Rubicon Setup, look for your rbc40*.bpl file under the `Packages\pkg___` folder. (You will need to memorize the Rubicon package code for your bridge in order to know the exact BPL file to look for, e.g. N for NexusDB. There is a complete list of package codes in the Installation chapter, under "Package Naming Conventions".)

If you do not find the BPL file, there is a problem.

Why Bridges Fail to Compile

You can determine the exact reason for the problem by opening the LOG file with a name such as `Rubicon\Packages\dccDPK-d17_win32.log` and looking for the word "fatal."

Configure Library Names, etc.

To fix problems relating to library names or search path, run **Start > All Programs > HREF Tools > Rubicon > ZMAdmin for Rubicon**. In ZMAdmin, you can fix third party library names and/or the path to those libraries. Be sure to include the path for both the DCP and BPL libraries. Some vendors put both DCP and BPL files in a single path. Other bridges, include IBObjects, have DCP and BPL files in separate folders so you must reference both locations.

Rubicon supports some advanced compiler flags. You can enter these in ZMAdmin for Rubicon. Within ZMAdmin, navigate to **Rubicon Components > Rubicon > Compilation**. On the right-side panel, you should see a place to enter Compiler Defines.

Within ZMAdmin, you can display Hints. Open those and read them for specific help with any setting.

Recompiling after Fixing Library Names and/or Paths

New in Rubicon v4.0

After you fix library names and paths within ZMAdmin, regenerate everything by running three BAT files. The following are in the `Rubicon\Setup` folder:

```
RbcAssistant_01_Create-Bat-Files.bat

RbcAssistant_02_Run-Bat-Files.bat

RbcAssistant_03_Install-to-IDE.bat
```

Shortcuts to the same three BAT files are located under **Start > All Programs > HREF Tools > Rubicon**.

If you are stuck on step 02, feel free to use our contact page to send your detailed LOG file to technical support (you can paste it in as part of your message). Use *http://www.href.com/contact* or ask for help through the Rubicon support newsgroup *http://www.href.com/newsgroups*.

Download Model Files (JPGs)

If you are interested in viewing the model files which show the object oriented structure of the Rubicon components, you may download them separately from the Setup installer.

There is a 10mb ZIP file waiting at the following web address: *http://www.href.com/pub/rubicon/Rubicon3/docs/RBModel.zip*

Borland Database Engine

Embarcadero never shipped a 64-bit version of the BDE. As of Delphi XE4, Embarcadero is marking the BDE deprecated.

Although the BDE is officially obsolete, it remains free and useful for those who know its quirks. If you need to install the latest 32-bit BDE without installing Delphi, there is a standalone installer available for download from *http://www.href.com/pub/sw*.

Rubicon Demos/Examples Use Shared Config,Yet...

In order to run the Rubicon demos, you will need to connect to some sample data. The example programs jump through some hoops to figure out the database name, login user, etc for the content type (e.g. “parts,” “rail accident,” or “rubicon help” content.

You are more than welcome to ignore all that and test against your own databases!!!

Look in the INC file that is included in your ExMake and ExSrch demo program. It will be connecting up to a database and doing some configuration. Comment that out and put in the details for your own database and tables.

OPERATION

How to Compile with Free Rubicon Components

If you have a Lite or Evaluation license, you must build all projects with packages. The details vary, depending on which database bridge you are using.

For example, to build with Delphi XE4 and IB Express, the runtime package list would be

rbc40_**e**_pas_lib_d18_win32;rbc40_core_pas_lib_d18_win32;rbc40_core_free_pas_lib_d18_win32

i.e. bridge code "E" for IB Express and 'pas' for Pascal
d18_win32 for Delphi XE4 on Windows 32-bit
core units for Rubicon itself
core library to enable the free license

whereas for use with C++Builder XE3 and dbExpress, the package list would be:

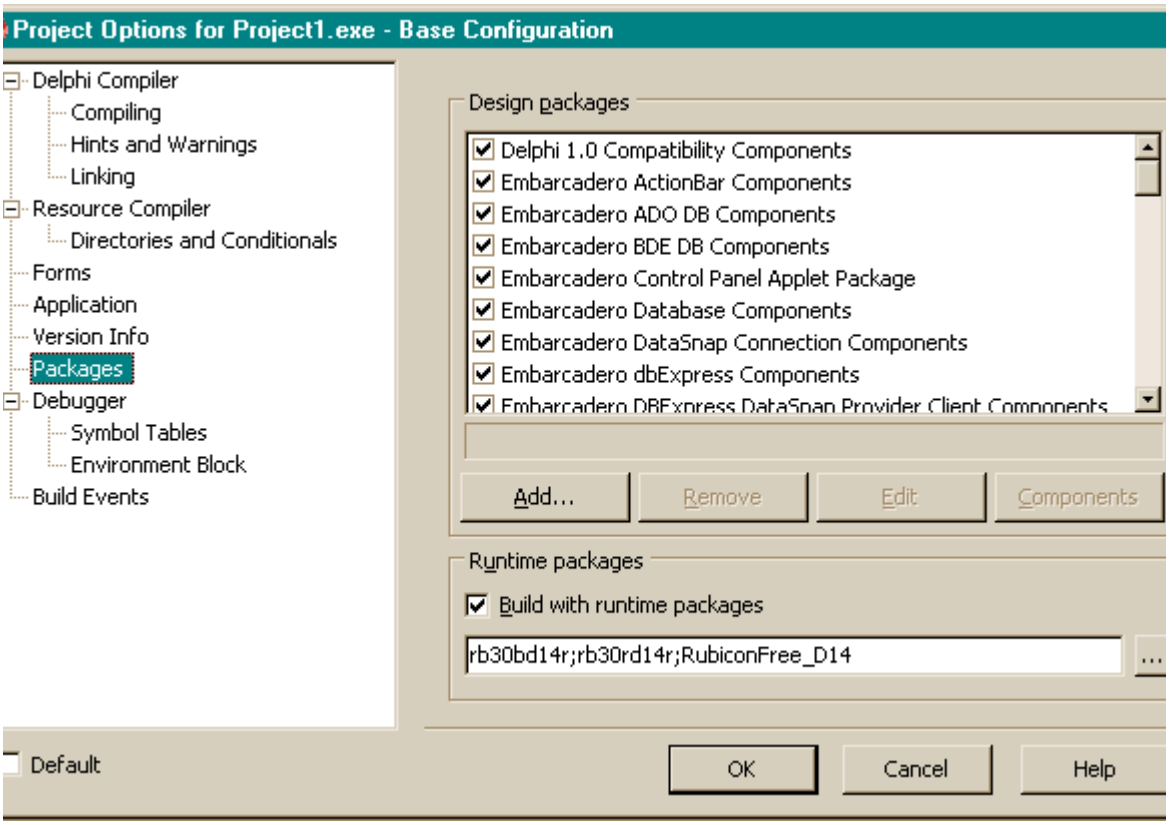
rbc40_**s**_cpp_lib_c17_win32;rbc40_core_cpp_lib_c17_win32;rbc40_core_free_cpp_lib_c17_win32

i.e. bridge code "S" for dbExpress and 'cpp' for C++Builder;

c17_win32 for C++Builder XE3 on Windows 32-bit
core units for Rubicon itself
core library to enable the free license

If you do not already know your compiler and database bridge codes, please see the charts in the Installation chapter.

If you are using the **paid** version of Rubicon, you may still build with packages if you wish. The difference is that you **never** need the third one, rbc40_core_free*.bpl.



To set the list of packages for your project, use the menu in the Delphi IDE: **Project > Options, Packages**. You can include more packages than the ones shown above, but not fewer.

Package filenames have changed since the above screenshot was taken.

How to Make

Building or making Rubicon indexes requires that Rubicon read each record in the target dataset, process the fields designated to be indexed, and then save the words and their indexes to the dataset.

Required Components

- *TrbMake*
- *TrbCache*
- *TrbTextLink* descendent

- *TrbWordsLink* descendent
- *TDataSet* descendent for Text
- *TDataSet* descendent for Words

Optional Components

- TrbProgressDialog

Operation

TrbMake uses the *TrbTextLink* to read the text from the *TDataSet* descendent being indexed, builds the indexes for each unique word in memory (*TrbCache*), and then writes the indexes via the *TrbWordsLink* descendent to a *TDataSet* descendent. A *TrbProgressDialog* can be used to monitor the process of this often lengthy operation.

Note: The TrbTextLink and TrbWordsLink descendents used by TrbMake should include "Make" in their names (e.g. TrbMakeTextTableLink). These versions of the components expose properties that are Make specific.

The fields included in the index are controlled by the *TrbMake FieldNames* property. The parsing of Words is defined by the *MinWordLen* and *WordDelims* properties.

In order to identify where a record came from, each record must have a unique integer field (*IndexFieldName*).

Rubicon can try to read the entire table in one step, or it can divide the process into segments. The latter approach is used with larger table where it is simply impossible to process the whole table in one pass. When using segments, the resulting Words table is called segmented, otherwise it is referred to as non-segmented. The *SegmentSize* property controls segmentation.

There is no hard and fast rule as to when to use the segmented approach. The factors to be considered include the number of records, the number of unique words, the efficiency of the index, and the amount of physical memory installed. A good indication that the process would benefit from segmentation is if the build process on a non segmented Words table slows down significantly and/or there appears to be a great deal of virtual memory disk swapping.

The amount of memory to devote to the build is controlled by the *TrbCache MemoryLimit* property. *MemoryLimit* should not be set any higher than the point that virtual memory starts being used. This threshold varies from system to system, but as a general starting point for Win 9x systems, use physical memory less 8mb, for Win NT, use half of physical memory.

Non-English Text and Unicode Support

As of v4, you can enable full Unicode support by compiling with Delphi 2010+ and controlling two properties:

```
rbMake1.Ansi := False;
rbAccept1.Alpha := 'ABCDEFGHIJKLMNOPQ'; // letters to accept
```

By default, the letters of your current locale are accepted.

Step By Step

1 Using the BDE Configuration applet in the Control Panel, check that a Rubicon4 alias exists pointing to RBExamples\data

2 Open a new application

3 Click on the Data Access tab in the component palette

4 Add a *TTable* (Table1), set the *DatabaseName* to Rubicon4, the *TableName* to help.db, the *IndexFieldNames* to HelpNo, the *Active* property to True

5 Add a second *TTable* (Table2), set the *DatabaseName* to Rubicon4, and *TableName* to Words. Do not try to set *Active* to True! This table will later be created by rbMake1.

6 Click on the R2 Drivers tab in the component palette

7 Add a *TrbMakeTextTableLink* (rbMakeTextTableLink1) to the form, set the *Table* property to Table1, click on the *FieldNames* property and add all the fields except HelpNo and Parent, and set *IndexFieldName* to HelpNo

8 Add a *TrbMakeWordsTableLink* (rbMakeWordsTableLink1) to the form and set the *Table* property to Table2

9 Click on the Rubicon4 tab in the component palette

10 Add a *TrbCache* (rbCache1) to the form and set *MemoryLimit* to 2000 (2mb)

11 Add a *TrbMake* (rbMake1) to the form, set *Cache* to rbCache1, set *TextLink* to rbMakeTextTableLink1, and set *WordsLink* to rbMakeWordsTableLink1

12 Add a *TrbProgressDialog* to the form. It will automatically configure itself.

13 Add a *TButton* (Button1) to the form, set the caption to Make, double click on the button and add the following code: rbMake1.Execute

14 To view the tables, add *TDataSource* and *TDBGrid* components

15 Run the application and press the Make button

16 You may wish to experiment with the *TrbMake MinWordLen* and *WordDelimiters* parameters and see how they affect the Words table.

Troubleshooting

Since the *TrbCache* unit is used by several Rubicon components, the *MemoryLimit* property has a default value of 1mb which is appropriate when used with *TrbSearch*. However, 1mb is usually not enough memory when used with *TrbMake*. Be sure to increase this value as

described in the previous paragraph.

Extraneous characters can often appear in the Words table. This usually occurs because the source text has been downloaded from the internet or OCR'd. To filter out these characters, simply add them to the *WordDelims* property. See *WordDelims* in the reference section on how to filter out all characters less than #30 or greater than #127.

To index lookup or linked fields, simply add a calculated field to the table and included the calculated field in the *FieldNames* property. If the link is to a memo field, an *OnProcessField* event handler will need to be written that performs the lookup and passes the memo to the Rubicon engine.

When working with large tables, it is often helpful to test all the property values by setting the *CounterLimit* property to 1000. This will build a Words table based on the first 1000 records. Check the words in the table to see if they are properly delineated. When the text includes international characters, also check that these characters appear correctly in the table (if not, see International Character Issues).

Example "Make" Project

`ExMake.dpr` (driver specific version located in each `RBExamples` subdirectory)

How to Update (Single User)

In order to keep the Words table current, Rubicon needs to be notified before and after each record is deleted, edited, or inserted. Rubicon then compares the state of the record before and after the change, determines which words have been added or deleted from the record, and makes the appropriate changes to the indexes.

Required Components

- TrbUpdate
- *TrbTextLink* descendent
- *TrbWordsLink* descendent
- *TDataSet* descendent for Text
- *TDataSet* descendent for Words

Optional Components

- *TrbTable* (recommended)
- *TrbCache* (recommended)
- *TrbUpdateDialog*

Operation

As mentioned above, *TrbUpdate* needs to be notified before and after a change takes place. The notification process is build-in to the *TrbTable* component, so if all changes to the table are processed through *TrbTable*, the Words will be kept current.

Note: There are dataset specific versions of TrbTable for most third party database engines.

Updating performance can be significantly improved by using the *TrbCache* component and setting the *TrbUpdate DelayedWrites* property to `True`. Under these conditions, changes are not immediately written to disk and thus reduce disk I/O. Since updates are held in cache, other applications performing searches against the same Words table will not see the changes until they are written to disk by calling either *FlushCache* or *WriteCache* methods (if the other applications are using caches in their searches, then those caches would also have to be flushed).

Within the same application, the cache can be shared between a *TrbUpdate* and *TrbSearch* that are accessing the same non segmented Words table. This would insure that *TrbSearch* would have access to the latest changes without forcing those changes to be written to disk. If the Words table is segmented, then the cache may not be shared.

When updates consist of just appending new records, other approaches not requiring a *TrbTable* (or equivalent) may be used. The *TrbUpdate.BatchAdd* method and *TrbAppend* components are alternatives.

Troubleshooting

Records may be inserted and deleted as long as their location is greater than *MinIndex*. If *MinIndex* is 1000, then any record whose location is greater than 1000 may be inserted or deleted. Record 1000 may not be deleted, nor may any record be added with a location less than 1000. Any record may be edited. Certain additional restrictions apply when using drivers that support a blank *IndexFieldName*.

In order to be able to delete the first record, or be able to add records in front of the first record, a number of locations have to be reserved at the front of the index. To do this, add an *OnMinIndex* event handler and set the value of *MinIndex* to a lower value. Using the above example, setting *MinIndex* in the *OnMinIndex* event handler to 500 would enable the application to delete the record with a location of 1000 and add records as long as the location was greater than or equal to 500.

Warning: The OnMinIndex event cannot be used only in TrbUpdate. The event must be used consistently be each Rubicon component that uses the Words table. This event cannot be added after the Words table has been built. The Words table must be built with the event enabled.

Since there should be only one *TrbUpdate* per Words table, the *Cache* property may not be shared with another *TrbUpdate* component. If the Words table is segmented, the *Cache* may not be shared with a *TrbSearch* component.

Because the *Cache* is a separate from the update engine and *WordsLink*, be sure to check that all the records have been written before shutting down the application. This can be done by calling the *WriteCache* method.

The *TrbUpdateDialog* component may be used during development to monitor the performance of the updating process. Among other things, the dialog box displays how many indexes are held in cache, and of those how many have been modified and not yet written to disk. These statistics are useful in tuning the performance of the application.

Example

`ExUpd.dpr` (driver specific version located in most `Rubicon\DBDemos` subdirectories)

How to Update (Multi User)

Multi user updates work much the same way as single user updates, except that the work is divided between one or more clients and a single server application. Like a single user application, the client application notifies Rubicon before and after a change is processed. Rubicon then calculates which words have been added or deleted. Since other clients may be updating the same indexes, the client application cannot process changes to the indexes itself. Instead, the client notifies the server application of the changes by writing those changes to a common *NetDataSet*. The server application reads and processes the changes posted to the *NetDataSet* and updates the Words table.

Required Components

- Client Application
- *TrbClientUpdate*
- *TrbTextLink* descendent
- *TrbWordsLink* descendent
- *TDataSet* descendent for Text
- *TDataSet* descendent for Words
- *TDataSet* descendent for NetDataSet
- Server Application
- *TrbServerUpdate*
- *TrbTextLink* descendent
- *TrbWordsLink* descendent
- *TDataSet* descendent for Text
- *TDataSet* descendent for Words
- *TDataSet* descendent for NetDataSet

Optional Components

- Server Application

- *TrbCache* (recommended)
- TrbUpdateDialog
- Client Application
- *TrbTable* (recommended)

Operation

Multi user updates perform all the same steps as a single user update, but where the work is performed varies. Setting up the client application is nearly identical with the exception that a *TrbClientUpdate* is used instead of a *TrbUpdate*. *TrbClientUpdate* does not use a cache and contains the additional *NetDataSet* property.

The server application also shares many similarities to a single user update application, but uses the *TrbServerUpdate* component which also contains a *NetDataSet* property. *TrbServerUpdate* does make use of a cache so that during periods of high activity that changes can be held in memory and later written to disk.

The server application must be running before any client as it creates the *NetDataSet*. However, once the *NetDataSet* has been created, the server application need not be running (when the server application is restarted, it will process any changes posted to the *NetDataSet*).

The server application should be running on the system where the tables are physically located. If this is not possible (e.g. with a NetWare file server), then it should be run on a dedicated client with a fast connection to the file server.

When processing a large number of changes, the server application can fall behind the activity of the clients. The clients may test to see whether the server has caught up by using the *IsCurrent* method.

Troubleshooting

Multi user updates share all the issues with single user updates, but require a bit more coordination as more applications are involved. Since the *NetDataSet* is used to transfer changes between the applications, it must be correctly identified by all the clients and the server. In addition, the appropriate multi user rights must be set in the database engine. For BDE applications using local tables, this means setting Local Share to `True`.

If searches are being simultaneously executed against the Words table, the search applications should periodically flush their cache so that they will be able to read the most recent version of the indexes.

Server Application

The server application, `Server.dpr`, is located in the `Rubicon\utils` subdirectory. It is a

ready to run server application. You can start it by a shortcut found under **Start > Programs > HREF Tools > Rubicon > Utility - Server**.

•The application does not have to be run on the server, although doing so increases performance and reduces network traffic
•Configure the application by filling in the edit boxes on the configuration tab. The *Text* table and *Words* table must already exist. The *NetDataSet* will be created if it does not already exist.
•Server.exe must be running before any of the client apps (press Process button) if the *NetDataSet* does not exist
•Changes are cached in memory until its cache is full or when it reaches the end of the *NetDataSet* (at which point it is waiting for new records)
•When running, the target table grid does not automatically refresh itself. Do this manually. Best to leave this grid visible as it does not redraw itself, thereby leaving more time to processing.
•When it is processing, navigation of the *NetDataSet* *is disabled*

Example

```
Rubicon\utils\Server.dpr
Rubicon\RBDemos\demo_b_ttable\ExClient.dpr
```

How to Search

Searching consists of prompting the user for a query, parsing the query into words, reading the words and indexes from the Words table, and calculating the result. For proximity searches, ranked searches, and searches using the *SubFieldNames* property, there is an added step of reading the text itself.

Note: Many of the basic concepts of searching are discussed in the Architecture section.

Required Components

- *TrbSearch*
- *TrbTextLink* descendent
- *TrbWordsLink* descendent
- *TDataSet* descendent for Text
- *TDataSet* descendent for Words

Optional Components

- *TrbCache* (highly recommended)
- *TrbMatchMaker*

Operation

Searching always requires setting the *SearchFor* property and calling *Execute*. Searches are not case sensitive. The number of matching locations or records is returned in the *MatchCount* property, while the words matched by the search can be accessed by calling the *MatchingWords* methods (very useful when wildcards are used). A list of matching locations can be accessed by calling the *MatchingLocations* method.

Searches are record specific, not field specific. This means that a search for the word 'jackson' could find records that contain 'Mr. Jackson' in the Name field, '125 Jackson Street' in the Address field, and 'Jackson Hole' in the City field.

Note: proximity searches do not cross field boundaries.

While not requiring the user to specify individual fields to search is generally a plus, there may be instances when the search should be restricted to a subset of fields. In these cases, there are two options: one is to construct a second Words table that is limited to the subset of fields. The limitation here is that the subset would need to be known ahead of time so that the dictionary could be pre-built (for small databases, this may not be an issue). When using this approach, the application should have a *TrbSearch* devoted to each variation of the Words table. The alternative would be to simply switch the table assigned to *Words* table, but this technique incurs a penalty because the component is forced to reinitialize itself each time the switch occurs.

The second option would be to use *TrbTextDataSetLink SubFieldNames* property to limit a

search to a subset of the fields represented in the Words table. Using *SubFieldNames* forces the component to read the Text during each search, and is therefore slower than the previous approach. The only records read are those that match the search criteria before applying *SubFieldNames*. During the reading process, the *SubFieldNames* are checked to ensure that they meet search criteria.

Rubicon 1: SubFieldNames was part of TSearchDictionary. It now resides in the TrbTextDataSetLink component.

The *FindFirst*, *FindNext*, *FindPrior*, and *FindLast* methods may be used to navigate to matching records. When ranking is not used, navigation is in *IndexFieldName* order. When ranking is enabled by setting the *RankMode* property, navigation is from highest (*FindFirst*) to lowest (*FindLast*) ranking.

Providing a *Cache* component reduces database reads as indexes that are already in cache are not re-read. If updating is being performed simultaneously, the cache should periodically be flushed so that it may access the latest changes to the indexes.

A copy of the matching records may be placed in a separate dataset by using the *TrbMatchMaker* component. This component creates a new dataset, reads the matching records in the Text and copies them to the new dataset.

Rubicon 1: The functionality of TSearchDictionary CreateMatchTable method now resides in TrbMatchMaker.

Step By Step

Note: Be sure to have completed the Make Step By Step instructions before beginning here.

1 Open a new application

2 Click on the Data Access tab on the component palette

3 Add a *TTable* (Table1), set the *DatabaseName* to Rubicon, the *TableName* to help.db, the *IndexFieldNames* to HelpNo, and the *Active* property to True

4 Add a second *TTable* (Table2), set the *DatabaseName* to Rubicon, *TableName* to Words, and the *Active* property to True

5 Add a third *TTable* (Table3), set the *DatabaseName* to Rubicon, *TableName* to Match, but leave the *Active* property False

6 Click on the R2 Drivers tab on the component palette

7 Add a *TrbTextTableLink* (rbTextTableLink1) to the form and set the *Table* property to Table1

8 Add a *TrbWordsTableLink* (rbWordsTableLink1) to the form and set the *Table* property to Table2

9 Click on the Rubicon4 tab on the component palette

10 Add a *TrbCache* (rbCache1) to the form

11 Add a *TrbSearch* (rbSearch1) to the form, set the *Cache* to rbCache1, *TextLink* to rbTextTableLink1, and *WordsLink* to rbWordsTableLink1

12 Add a *TrbMatchMaker* (rbMatchMaker1) to the form, set the *DataSet* property to Table3, and set the *Searcher* property to rbSearch1

13 Add a *TEdit* (Edit1) and clear the *Text* property

14 Add a *TButton* (Button1), set the *Caption* to Search, double click on the button and add the following code:

```
with rbSearch1 do
begin
SearchFor := Edit1.Text;
Execute;
if MatchCount > 0 then rbMatchMaker1.Execute
end;
```

15 Add a *TDataSource* and *TDBGrid* components and connect them to Table3

16 Run the application

17 Move to *Edit1* and enter class

18 Press the Search button and the grid is filled with matching records. For some searches you may try, the matching words may appear in the memo field (you may want to add a *TDBMemo* control to the form to make memo field(s) visible). If you are using your own table and no records appear in the grid, either enter a more specific search or set the *TrbSearch RecordLimit* property to a higher value.

19 Exit the program, select rbSearch1, change the *SearchLogic* to slExpression and change the *RankMode* to rmCount

20 Run the application again, enter class and make into Edit1, and press the Search button. The matching records are returned in rank order and a rank field has been added as the rightmost column.

Troubleshooting

Wildcard and proximity searches can be time consuming (e.g. the user searches for '*'). While you may try to validate the search request before passing it on to Rubicon, it is probably just better to set the TimeLimit property. Once TimeLimit is exceeded, the search will abort.

Ranking can also be time consuming, so before enabling ranking, you may wish to check the MatchCount property to see if there are a reasonable number of matching records.

SearchLogic of slNot should not be used when SearchMode is smSearch because any gaps between index values will be reported as false matches. It is safe to use these results with subsequent searches to narrow or widen the scope of the search as long as a SearchLogic

other than slNot is used at least once.

If the Words table is segmented, the Cache may not be shared with a TrbUpdate component.

Example

`ExSrch.dpr` (driver specific version located in most `Rubicon\DBDemos` subdirectories)

How to Use a TClientDataSet

The Client/Server editions of Delphi 3 and above and C++ Builder 3 and above support the *TClientDataSet*. This dataset may be used with the *rbMatchMaker* by simply including the *rbCDS* unit in uses clause of the application or unit. The primary advantage of a *TClientDataSet* over a *TTable* is speed – *TClientDataSet* is an in-memory table which can be created much faster than a *TTable*.

Required Components

TrbSearch

TrbTextLink descendent

TrbWordsLink descendent

TDataSet descendent for Text

TDataSet descendent for Words

TrbCache (highly recommended)

TrbMatchMaker

TClientDataSet for *TrbMatchMaker*

Operation

The setup is exactly the same as a regular search that uses a *TrbMatchMaker* with the exception that a *TClientDataSet* is used instead of a table based *TDataSet* descendent. The application must include *rbCDS* in its *uses* statement.

Troubleshooting

If a "Dataset not supported" error is raised, then *rbCDS* has not been included the *uses* statement.

Example

DBExamples\ttable\ExCDS.dpr

How to Search Multiple Tables

Rubicon may be used to search multiple tables simultaneously. The tables being searched may or may not have the same field structure. Multi table searching is coordinated by the *TrbController* component. Rather than performing the search itself, *TrbController* simply coordinates the activities of multiple *TrbSearch* components and reports back a "consolidation" of results.

Required Components

- TrbController
- For each table being searched
- *TrbSearch*
- *TrbTextLink* descendent
- *TrbWordsLink* descendent
- *TDataSet* descendent for Text
- *TDataSet* descendent for Words
- *TrbCache* (highly recommended)

Optional Components

- *TrbMatchMaker*
- *TDataSet* descendent for *TrbMatchMaker*

Operation

As stated above, *TrbController* coordinates and consolidates the activities of multiple *TrbSearch* components. Before a *TrbController* can be used, each table being searched must be prepared by building a Words table and then adding the *TrbSearch*, link, and dataset components to the form or data module in the same fashion as you would when setting up a single table search.

When placed on the form or data module, the *TrbController* will coordinate the activities of all the other *TrbSearch* components on that form or data module. One of the *TrbSearch* components should be designated as the *MasterSearcher*. Set the *SearchFor*, *SearchLogic*, and *SearchMode* properties of the *MasterSearcher* to the desired values. When *TrbController.Execute* is called, it will apply the properties of the *MasterSearcher* to the other search components, execute the search, and consolidate the results.

To navigate through the matching records, call the *TrbController FindFirst*, *FindLast*, *FindNext*, and *FindLast* methods. Since *FindNext* may move from the last matching record of one

table to the first matching record of the next table, it is important to be able to distinguish which table the navigation has moved to. Use the *Searcher* property to determine which table the *FindXxxx* routines relate to.

To create a match table, assign a *TrbMatchMaker* to the *TrbController.MatchMaker* property and call *TrbController.CreateMatchDataSet* (and not *TrbMatchMaker.Execute*)

The tables being searched may have the same or different record structures. More precisely, if the field names, types, and sizes are the same for the *TrbSearch.FieldNames* across all the search components, then a call to *TrbController.CreateMatchDataSet* produces a table just like *TrbMatchMaker.Execute* would.

If the field names, types, or sizes in *TrbSearch.FieldNames* are different, then when *TrbController* creates a match table and stores all the field values in a memo field.

In either case, *CreateMatchDataSet* will also add fields to each record that indicate which table the matching record originated from. These fields consist of DatabaseName, TableName, and Location.

Use the *OnAcceptSearch* to control which *TrbSearch* components are included in the search. By using this event, tables may be selectively included or excluded from the search.

Troubleshooting

Since *TrbController* coordinates multiple *TrbSearch* components, you need to be sure that each individual *TrbSearch* component is setup correctly in order for *TrbController* to work properly. During development, this may be accomplished by performing a search with each *TrbSearch*.

Example

`Rubicon\RBDemos\demo_b_ttable\ExCntrlr.dpr`

Working with Huge Tables

Working with large amounts of text does not vary significantly from working with 'normal' amounts of text. The following sections cover some special considerations.

Testing

Before indexing a large amount of text, it is a good idea to perform a test build by setting *CounterLimit* to about 4000, then inspect the Words to see if there are any obvious characters and/or words which should be excluded from the build.

Configuration

Indexing performance will largely be a function of the number of unique words, the *IndexRange*, segmentation, and available memory.

Use the *WordDelims*, *MinWordLen*, *OmitList*, and *FieldNames* properties to limit the number of indexed. The *OnAcceptWord* event may also be used to further limit the number of words indexed.

The *IndexRange* is the difference between the maximum and minimum location values. It is important that this index be efficient so that memory will be used efficiently. An efficient index is one that has few gaps between index values.

Since *TrbMake* performs indexing in memory, the amount of physical memory may limit the capacity of the component. The first option to consider is segmentation. To enable segmentation, set the *SegmentSize* property to some fraction of the *IndexRange*. Segmentation simply tells *TrbMake* to divide the job up into more manageable chunks which are more likely to fit into available memory. Try varying the *SegmentSize* to find the optimal value.

Set *MemoryLimit* equal to the amount of physical RAM minus 4 to 8 mb for Win9x or half of physical RAM for NT (later, you may wish to experiment with this setting – it is not a hard and fast rule!). If it appears that the operating system is excessively using virtual memory and you do not which to decrease the *SegmentSize* any further, then *MemoryLimit* is set too low or more physical memory needs to be added.

Other configuration options include:

Set tables' *Exclusive* property to True or turn off Local Share (single user BDE applications using local tables)

Enable *Transactions* and/or use *TQuery* based drivers when using SQL tables

If the table type natively supports *ftBytes* fields, use the *BytesFieldSize* property, else consider using the *CharFieldSize* property

Set the *Ansi* property to False, if possible

Enable the dbiRead and *dbiWrite* options in *TrbCustomXxxxBDELink* based components. These options are valid when using Paradox, Local InterBase, InterBase 4 or higher, or other servers that supports 32 bit integers

See RBDEFINE.INC for other performance options

Indexing

While the application is indexing, shut down other applications

Build the Words on multiple machines or processors using the *FirstSegment* and *LastSegment* properties of *TrbMake*

Updating

Updates will generally need to be performed a more powerful system (with ample memory). It is recommended that a backup copy of the original Rubicon indexes be made before performing the update. If records are only being added to the table, then the *BatchAdd* approach to updating should be considered. If records are only being added and the Rubicon indexes are not segmented, then *TrbAppend* may be used.

Searching

There are no special considerations to take into account.

Filters and Ranges

Ranges may be used with Rubicon under certain restrictive conditions. The same is true with filters with the exception of using search results to filter the view of the Text table (discussed later in this section).

The Rubicon components keep track of which records words appear in by indexing their locations. The location is a base plus offset calculation, where the base is the *MinIndex* and the maximum is the *MaxIndex*. The difference between these two is the *IndexRange*, and this value is used to determine how much memory is required by each uncompressed index.

When applied to the Text table, filters and ranges artificially change the *MinIndex* and *IndexRange* of the table, thereby invalidating any existing word indexes. This is because the components assume that the *MinIndex* and *IndexRange* are fixed *TrbUpdate* is aware of adds and deletes). This assumption is made for performance reasons since recalculating the *IndexRange* takes time (this is especially important for SQL tables). In addition, the calculation of *IndexRange* is not performed until the property is first used. This behavior has important consequences as we will see in the following section.

The key to working with filters and ranges is carefully controlling the values of *MinIndex* and *IndexRange* along with setting the filters or ranges at the right time. Another approach is to devote a separate unfiltered and 'unranged' or range free dataset to the Rubicon component. These methods will be discussed in the following sections. The last alternative is to clear the filter or range before any Rubicon operation (before a search or change to the table). Since this can be time consuming, this may not be a practical alternative.

Before going into detail on these, it may be useful to clarify when filters and ranges will not work:

Filters and ranges may never be used with *TrbMake* as it requires an unrestricted view of the Text

Filters and ranges may never be used when using a *TrbCustomTextBDELink* with a blank *IndexFieldName* and the source table is Paradox. Under these conditions, the location is a sequence number, and filters and ranges (along with secondary indexes) change the value of the sequence number (which is the index location), and thereby always invalidate the word index

Controlling MinIndex and IndexRange

MinIndex and *IndexRange* remain uninitialized until they are first used. For *TrbUpdate* and *TrbServerUpdate*, they are first used when *Initialize* is called, while *TrbSearch* waits until the first search is performed. Since an application may already have a filter or range in place, this poses a problem. The workaround is to explicitly call *Initialize* before any filters or ranges are put into place. By doing so, the *IndexRange* is initialized before the filter or range takes affect and therefore has an unrestricted view of the Text dataset.

IndexRange will also become uninitialized after setting a new DataSet, or *IndexFieldName*, and after the Text dataset becomes active.

For the technique used above to work, filters or ranges must not have been enabled from within the IDE. These are the only steps required for *TrbUpdate* and *TrbServerUpdate*.

For *TrbSearch*, searches may be performed on a filtered or range restricted Text dataset as long as the search does not use *slNear* or *slPhrase SearchLogic* (or use an *slExpression* that uses NEAR or phrases), nor may *SubFieldNames* or other action that requires reading the Text dataset.

Searches always apply the search to the entire database and the results never reflect any filters or ranges that may be in place. The *FindXxxx* routines always navigate based on the search results, and so they too are unaware of filters or ranges. This essentially restricts you to using the *TrbSearch RankArray* and *MatchBits* properties. For these reasons, it is much better to use the approach described in the next section.

Using a Separate TTable

Another approach to this problem is to devote a separate *TTable* to the Rubicon component and apply the filters and/or ranges to another *TTable* that the user views on screen. All the capabilities of *TrbSearch* may be used, but search results again will be relative to the entire table, not the filtered or range restricted records. Also, the *FindXxxx* routines will navigate the table in the *DataSet*.

Most of the above restrictions can be overcome by using the *CheckMatchResults* procedure. This procedure checks to see whether the records it has located appear in the table with the range or filter in place. If the record cannot be found, it is removed from the match results.

IndexFieldName

When the *IndexFieldName* is in use, the first call to *IndexRange* will force the component to temporarily switch indexes (assuming the table is not open on *IndexFieldName*). Switching

indexes will often clear any existing filter or ranges. For this reason, it is better to devote a separate *TTable* to *TrbSearch*.

Using Search Results as a Filter

While it may be difficult to apply a filter or range to search results, it is easy to use the results of a search to act as a filter. In Delphi 2.0 and higher and C++ Builder, it is simply a matter of using the *OnFilterRecord* event:

```
procedure TMainForm.SearchTable1FilterRecord(DataSet: TDataSet;
var Accept: Boolean);
begin
Accept := rbSearch1.Matches
end;
```

This example assumes that *DataSet* is the same as *rbSearch1*.TextLink.*DataSet* (this technique is used in the Rubicon demo). Of course, you may wish to add additional conditions to the above code to further restrict the view of the table.

The disadvantage of this kind of filter is that a search may only produce a few matching records from a table of tens of thousands of records or more. You will want to carefully test this approach to ensure it meets you performance requirements.

Warning: Some Text drivers do not require setting the IndexFieldName. If the IndexFieldName is not set, filtering will probably not work.

CUSTOMIZATION

Optional Compiler Directives

Rubicon source may be compiled with a number of options which are listed below.

Generally speaking, you should not need to worry about any of these options. They are listed here for the exceptionally curious developer.

To use any of these, set them under **Project > Options** for your own projects, and/or set them in the definition of `compflags` within ZMAdmin.

Additional information about these flags can be found in the comments in this file: `Rubicon\Source\inc\RBDEFINE.INC`.

- AltMemMgr
- DBISAM2
- DBISAM2x
- DebugMode
- DETECTMEMLEAKS
- FASTMM
- HaveInfoPower
- HStrings
- ManualLeakReportingControl
- MemoryLogging
- ODBCExpress6
- ThreadSafe
- USE_JEDI_JCL
- USE_MDX
- UseAdvantage

- UseApollo
- UseAutoInc
- UseCodeBase
- UseDBISAM
- UseFlashFiler
- UseGotoKey
- UseNexusDB
- UseRubiconRichEdit
- UseTable
- VendorTab

  The VendorTab complier flag determines whether special Rubicon components install in the IDE onto a palette named Rubicon4 or onto palettes named based on the name of the vendor of the particular database bridge, e.g. Advantage.

- xDebugMode

Evaluation and "lite" users can not take advantage of these flags.

Example: Customizing the Multi-Table Demo

This section explains when and how to use the multi-table query bridge, `rubicon\source\rbBridge_0_query.pas`. This bridge is new in Rubicon v4.031.

The use-case for this bridge is when you have data-to-search in multiple linked tables and (a) your Rubicon bridge is table-based not query-based and (b) your database system does not provide a SQL "view" of the joined tables. One example of this situation is a Delphi XE project using the Apollo bridge. As long as your database vendor (e.g. Apollo) has its query component derived from TDataSet, this generic query bridge will work for you. It works because it uses the IProviderSupport interface defined by Embarcadero in recent versions of Delphi and C++Builder.

The two core examples, make" and "search," are provided for two bridges: IBExpress and FireDAC. If you compile with XE4, use the FireDAC files. Otherwise, use the IBExpress files.

A suitable database for this example needs to contain at least 2 tables whose records can be joined by a common field. The "nutrition" database published by the U.S. government fits

this requirement. Unfortunately it is published in Access file format and uses non-integer fields for its keys. This has been remedied in a Firebird 2.5 conversion of the data. The documentation for the nutrition database is here: *http://data.rubicon.href.com/RBSampleData/Access/Nutrition/sr25_doc.pdf* and page 32 of that PDF contains the entity relationship diagram.

Compiling and Running the Example Projects

If you want to test out this example, you will need to download suitable data and compile the projects. As of 28-May-2013, the sample data is available in Firebird 2.5 format. Download either an. fbk file (to be restored with gbak) or a ready-to-go .fdb file from *http://data.rubicon.href.com/RBSampleData/FirebirdSQL/*. The data is also available in Access format but the key fields have not been converted to Integer and will not work with Rubicon.

Note: you do not need to be able to run these examples in order to benefit from them. Instructions (below) explain how to modify the example files to work with your own database and your own bridge (e.g. Apollo). It will help if you can open the source files without errors about missing components, which is why they were developed with IBExpress and FireDAC, bridges available to any Embarcadero developer.

The two project files are:

Make: ExMakeMultiTable.dpr

Search: ExSrchMultiTable.dpr

First compile and run the Make project to create the NtrWords table (nutrition words index) in the database. Then run the Search project to test the search features.

Modifications for Your Own Database

These instructions assume you are using the IBExpress copy of the files and that your desired bridge is Apollo.

This process will be easiest if you run two copies of Delphi, loading the example into one and making your own project in another.

The example projects use a shared ancestor form and shared units for configuration. You will not need any of that in your own project. So, assume that your own project starts out as nothing more than a **File > New > VCL Forms Application**. (The menu choice varies slightly by version of Delphi.)

The "Make" Example

Copy and paste the following components all at once, as a group, from ExMakeMultiTableU.pas to your main form: rbMake1, rbAccept1, rbCache1, rbProgress1.

For sql-based bridges, add 2 query components and name them Query1 and Query2. For

table-based bridge such as Apollo, add 1 TApolloQuery named Query1 and 1 TApolloTable named Table2. (You can adjust the naming to your own satisfaction at the end, once you see how everything is linked up.) For now, query #1 is for Text and query/table#2 is for the Words index.

On the rbMake1 component, connect its TextLink property to Query1. Connect its WordsLink property Query2/Table2.

Add whatever "database/connection" and "transaction" components you would ordinarily have for your bridge, for example, add a TApolloDatabase named ApolloDatabase1. Connect Query1 and Query2/Table2 to the database component. If you want to store the Words index in a location away from your core data, use a second database component for Query2/Table2.

Copy the information about rbMakeTextIPSLink1: TrbMakeTextIPSLink; and rbMakeWordsLink1: TrbMakeWordsIBXLink; from the example to your project. There is a private declaration plus code in OnCreate and OnDestroy and OnActivate. Keep all references to TrbMakeTextIPSLink intact. That is the link based on IProviderSupport ("IPS") which uses rbBridge_0_query.pas. Search and replace all references from 'TrbMakeWordsIBXLink' to 'TrbMakeWordsApolloLink.' Because the Words table is a single table, using the normal vendor-specific link works well.

In TForm1.FormActivate, adjust the lines of code that connect the components. For example, `Query1.Database := Connection1` may need to be changed slightly so that it compiles with your database access components.

Make sure that rbMakeWordsLink1.DataSet points to Table2 (for Apollo) or Query2 (for sql-based bridges).

In TForm1.FormActivate, note that you do NOT need the 6 linse of code that create Query1 and Query2 on the fly because you have placed them as components from the palette. In TForm1.FormCreate, you do NOT need the 2 lines that initialize them to nil, and in TForm1.FormDestroy you do NOT need the 2 lines that FreeAndNil Query1 and Query2.

You may have noticed that the Name property of each component created at runtime is set in the PAS code. This Name property is not strictly required but it often helps when debugging.

Note: quite a few lines of code compile conditionally when CodeSite is used. CodeSite Express is included free with recent versions of RAD Studio. A paid version is available with full source from Raize Software, and that can be used with Delphi XE. There is no requirement to use CodeSite; it can be very helpful during development and troubleshooting.

Set rbCache1 AltMemMgr to False. Set rbCache1 MemoryLimit to 1000.

Set rbMake1 Segment size to 0 or MaxInt.

You will need code for TForm1.SQLBlock but we will come back to that.

You will not need ButtonNextClick per se. However you will probably want some Button or menu item that triggers the “make” process to start. In that OnClick event, you should roughly do the equivalent of these methods, in this order:

`Example_InitializeDB_and_Words_IBExpress`

The purpose here is to configure the database/connection component. You do NOT need to use any of the example code. Instead you can set the database properties in the Delphi IDE as you normally would. Usually this means defining the database location, username, password and character set. Details vary depending on the database system.

You should set the Words Tablename to a name that is not already in use. In the nutrition database, the tablename is ‘NtrWords’.

`Make sure OnMinIndex and OnMaxIndex are defined`

These are event handlers on the rbMakeTextIPSLink1 object. Copy and paste the declarations from the example but re-implement them. You want the OnMinIndex event handler to return the lowest unique integer identifier, which is usually 1. You want the OnMaxIndex to return the highest unique integer. The simplest way to find out the highest number is to run a ‘select max(mykey) from mytable’ query. The fastest way (in Firebird or Interbase) is to select the value from the generator associated with the primary key.

`Example_InitializeText_MultiTable`

This code is in uInitExample_multitable.pas. You will need the equivalents of all the “shared” functions, e.g. SharedMultiTableSQL, SharedFieldNames, SharedIndexFieldName, SharedTableName. However, as you are only implementing 1 system, your code will be simpler. It is still worth putting your answers into a unit that can be shared because you will need these again when you get to searching.

SharedTableName: this will be your main child/detail table. In the nutrition database, it is the Nut_Data table.

SharedIndexFieldName: this is the name of the field containing the unique integer.

SharedFieldNames: this is the list of fields that have text-to-be-indexed. It may help to copy the SharedFieldNames function, omitting the InDatabaseContentType parameter. Copy the example for dctNutrition because that is where you can see the complexity of pulling fields from multiple tables. You should keep the InSeparator parameter because some situations require sLineBreak and others require a comma.

SharedMultiTableSQL: this is the SELECT syntax that pulls together your tables. You should select your unique integer field plus other linking fields plus all text fields containing data-to-be-searched.

SharedMultiTableSQLWhereClause: it is helpful to separate this part of the SQL which joins the tables together.

Make sure that you implement each part of function Example_InitializeText_MultiTable, setting the SQL.Text, the FieldNames of TextQueryLink, the IndexFieldName of TextQueryLink, and the TableName of TextQueryLink.

As a last step, copy and adjust TForm1.SQLBlock so that it is suitable for your database. This code is hooked up via

```
rbMakeTextIPSLink1.OnPrepareQueryBlock := SQLBlock; // essential
```

With all of the above in place and compiling, your Button OnClick event should reach a point where it can open the TextQueryLink. If you can open your TextQueryLink without an exception, you are ready to build the Words index with the following single line of code:

```
rbMake1.Execute;
```

This tiny statement builds the index. It will create the Words table by itself. It will pop up a progress dialog for you and indicate progress as it reads your text and writes to your Words index. If everything completes without exceptions, you will have a Words index table in your database.

Review the contents of the Words table and see whether you are happy with the list of words. If it includes too many invalid words, you can use any of the following to assist:

- MinWordLen property on rbMake1: change to a value longer than 2
- WordDelims property on rbMake1: change the list of delimiters
- Properties on rbAccept1: adjust
- OnAcceptWord event of rbMake1: here you can put in completely custom rules to accept or reject words.

The Search Example

Usually the search code is added to an existing client application. Initially you may find it helpful to test with a separate new Vcl Forms Application.

As above, copy and paste the components from the search example into your own project. You will need rbSearch1 and rbCache1. You might not need rbMatchMaker1, ClientDataSet1, DataSource1.

Configure everything exactly the same way that you did during the Make process.

The main difference is that when you are ready for action, instead of rbMake1.execute, you will have these two lines:

```
rbSearch1.SearchFor := Edit1.Text; // user input of the search term(s)
rbSearch1.Execute;
```

The result of that search will initially be in an array in memory. The array contains a list of integers corresponding to the records that matched the search question. To get the array,

create a TList (e.g. FLastSearchKeys:= TList.Create) and then call

```
rbSearch1.MatchingLocations(FLastSearchKeys);
```

For the price of a little more configuration, it is possible (but not necessary) to use the TrbMatchMaker component to retrieve the data that corresponds to those keys, storing the result in a TClientDataSet or other TDataSet derivative. The right approach varies by application; please discuss this with technical support and/or study the examples for ideas. the implementation of the displlay of the search results is in `rbdemos\common\ExSrch-Form.pas`.

Example: Customizing Append and Make Components

The following example was contributed by a customer and is published with permission. It shows how you can extend the default behavior of the TrbAppend and TrbMake components.

Delphi source follows.

```
unit ebRbMO;

interface

uses
      rbAppend, rbMake;

type

  TrbAppendMySample = class(TrbAppend)

  private

    FStopLocation: LongInt;

  public

    procedure Execute;

  published

    property StopLocation: LongInt read FStopLocation write FStopLoca-
tion;

  end;


type

  TrbMakeMySample = class(TrbMake)
```

```
  private
    FStopLocation: LongInt;
    procedure SetStopLocation(const Value: LongInt);
    procedure SetCounterLimit(const Value: LongInt);
  public
    procedure Execute;
  published
    property CounterLimit : LongInt read FCounterLimit write SetCounter-
Limit;
    property StopLocation: LongInt read FStopLocation write SetStopLoca-
tion;
  end;


procedure Register;


implementation


uses Classes, rbBase, rbConst, rbFile, rbUpdate;


{ TrbAppendMySample }


procedure TrbAppendMySample.Execute;
var       SaveMinIndex : LongInt;
begin
 CheckNil(Cache,rbeNilCache);
 CheckNil(TextLink,rbeNilText);
 CheckNil(WordsLink,rbeNilWords);
{TextLink.Open;}
 WordsLink.ResetCounters;
```

```
{ReadProperties;}

{FStartIndex := (FStartLocation - 1) and $fffffff8 + TextLink.MinIndex
and 7;}

{TextLink.SetIndexRange(FStartIndex,Undefined);}

 State := [esUpdate];

 try

  SaveMinIndex := TextLink.MinIndex;

  FStartIndex := (FStartLocation - SaveMinIndex) and $fffffff8 +

                 TextLink.MinIndex;

  if FStartIndex < SaveMinIndex then

   RaiseRubiconError(rbeInvalidParameter);

  { Force TextLink.MaxIndex to be refreshed }

  TextLink.SetIndexRange(FStartIndex,Undefined);

  TextLink.SetIndexRange(FStartIndex,TextLink.MaxIndex);

  Bits.Capacity := TextLink.IndexRange + 1;

  Cache.DataSize := Bits.CopyBufferSize;

  if TextLink.GotoNearestLocation(FStartLocation) then

   begin

    State := State + [esReading];

    try

     Action := uaInsert;

     while not TextLink.EOF and

           ((FStopLocation = 0) or
            (TextLink.Location <= FStopLocation)) and (not Aborted)

      do

      begin

       Location := TextLink.Location;

       FCacheCounter := Cache.Counter;

       Process;
```

```
        DoAfterBatch;
        TextLink.Next;
       end;
      FAppendBits.Max := TextLink.IndexRange or 7;
     finally
      State := State - [esReading]
     end;
     TextLink.SetIndexRange(SaveMinIndex,TextLink.MaxIndex);
     Bits.Capacity := TextLink.IndexRange + 1;
     ResizeBuffer(Bits.CompressBufferSize);
     State := State + [esWriting];
     try
      Cache.Iterate(iAppend,False,True);
     finally
      FAppendBits.SwitchBack;
      FAppendBits.Max := 0;
      State := State - [esWriting]
     end
    end
  finally
   TextLink.SetIndexRange(Undefined,Undefined);
   State := State - [esUpdate]
  end
end;


{ TrbMakeMySample }
```

```
procedure TrbMakeMySample.Execute;

begin

 State := [esMake];

 try

  State := State + [esReading];

  if WordsLink.Segmented and

     (FFirstSegment > 0) then

   TextLink.GotoNearestLocation(TextLink.MinIndex + FFirstSegment * Seg-
mentSize)

  else

   TextLink.Lowest;

  Location := TextLink.Location;

  while not TextLink.EOF and

        ((FStopLocation = 0) or (TextLink.Location <= FStopLocation)) and

        not Aborted and

        ((FCounterLimit = 0) or (FCounter < FCounterLimit)) and

        (Segment <= FLastSegment) do

   begin

    Inc(FCounter);

    FCacheCounter := Cache.Counter;

    TextLink.Process(Self);

    TextLink.Next;

    {* Update the value of Location/Segment so while loop test is current
*}

    if Segment = FLastSegment then

     Location := TextLink.Location;

end;

  if Cache.Count > 0 then

   Post(True)
```

```
 finally

  State := State - [esMake]

 end

end;


procedure TrbMakeMySample.SetCounterLimit(const Value: LongInt);
begin
      FCounterLimit := Value;
      if Value <> 0 then
            FStopLocation := 0;
end;

procedure TrbMakeMySample.SetStopLocation(const Value: LongInt);
begin
      FStopLocation := Value;
      if Value <> 0 then
            FCounterLimit := 0;
end;

procedure Register;
begin
      RegisterComponents('ebUtil',[TrbAppendMySample, TrbMakeMySample]);
end;

end.
```

Program Service

This chapter helps you assess and solve technical problems. It provides advice in a question and answer format, as if our support department were talking with you on the phone. Read the questions carefully as they should help you quickly diagnose your trouble.

Common Issues with Solutions

Q: I am trying to compile in Delphi XE3 with Rubicon Lite. I am using runtime Packages. I get this error about a DCP file not being found when I build my project:

```
Done building project "ExUpSr.dproj" -- FAILED._ Build FAILED._

D:\Program Files\HREFTools\Rubi-
con\RBDemos\demo_m_ado\ExUpSrU.pas(18,18):

error F1026: F1026 File not found: 'rbc40_core_pas_lib.dcp'
```

Answer: Here is a workaround. You can fix the DCP error with a trick:

Tools > Options, Environment > Delphi > Library.

Make ANY change to the search path, such as delete an invalid path or add a fake path, regardless of whether that is fake or real. Save.

Project > Build

You may also find that it helps to go in the registry and disable Delphi's package cache.

Q: I have a paid license. I ran the installer but I do not have any Rubicon\Source files nor any Packages!

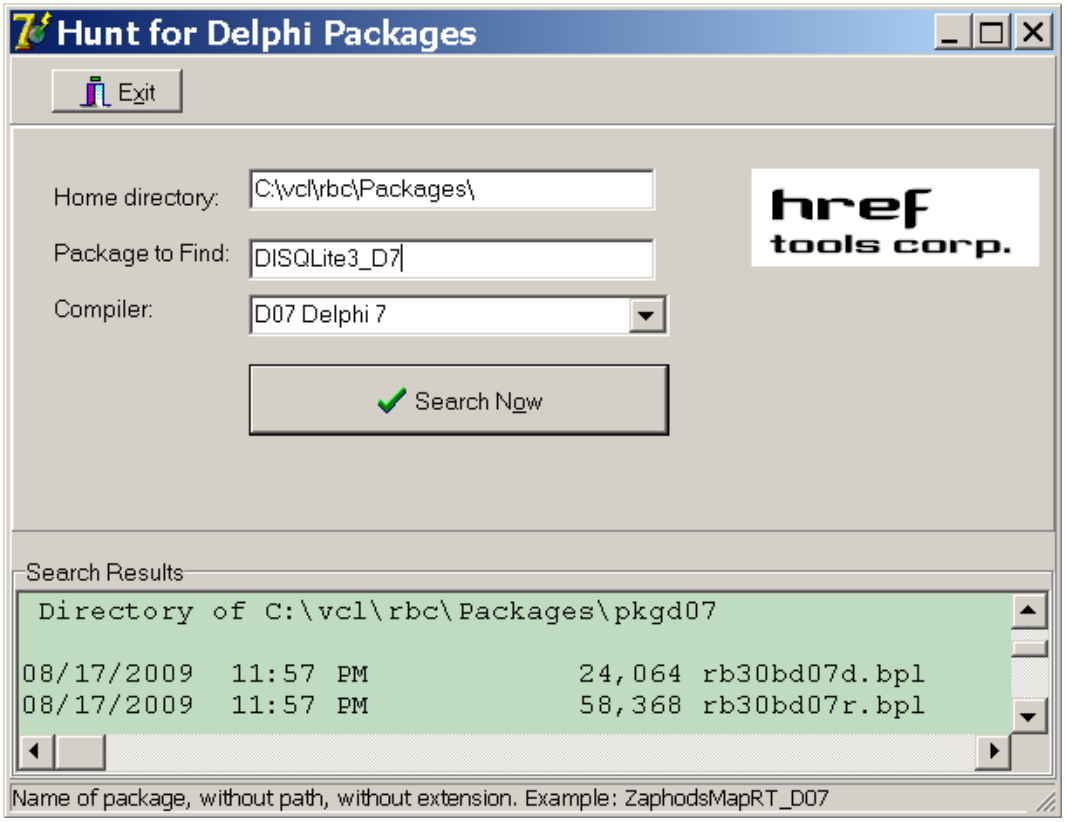
Re-run the Setup program and be very careful when you get to the screen with all the checkmarks (selecting components of the product). The first option is the most important one - Source - and can be accidentally unchecked, leading to the condition you describe.

Q: When I try to compile my Rubicon packages using my customized DCCDPK BAT file, I find a fatal error in the resulting LOG file. The message, "Fatal: Required package '___' not found," talks about a package than I have on disk! Why is that file not found by the compiler?

First, check your configuration in ZMAdmin and make sure that there is no trailing slash at the end of your path.

Second, make sure that have included the path to BOTH the DCP and BPL files for the package in question.

Third, if you suspect a typo but cannot see it, confirm the filenames by running Hunt4Packages (use the shortcut under **Start > Programs > HREF Tools > Rubicon > Hunt for Packages**).



- **Home directory**
  this is filled in for you. It should be the Rubicon\Packages folder.
- **Package to find**
  you should paste in the name of the package mentioned in the fatal error.
- **Compiler**
  select the compiler

Click [Search Now] and you should end up with a report listing all relevant details. Copy that report into a text editor such as Notepad for easier searching. Now you can quickly confirm the location of the package file.

If the report confirms that the package is on disk, the solution is either (a) customize the moresrc setting in the DCCDPK BAT file (only if you have discovered an error - please tell technical support) or (b) use shorter paths for the compiler and/or the database bridge. For example, instead of installing to c:\Program Files\Embarcadero\RAD Studio\11.0\ you could install to c:\Apps\Embarcadero\Delphi\XE4\ and for a database bridge, you could install to c:\vcl\nexus\ . You could install Rubicon to c:\vcl\rbc\. Avoid spaces in your paths whenever possible.

Q: Can I reduce the size of Words table by setting WordFieldSize to a lower value?

Yes, but you run the risk of increasing the number of truncated words, which leads to false matches during updates and ambiguous search results.

Q: How can I limit a dictionary build to a maximum amount of RAM?

In order to conserve RAM, be sure to set CacheMemoryLimit to the desired value. Then in the TextLink.AfterProcess event handler, include code that calls rbMake1.Abort when the maximum amount of RAM is exceeded.

Q: When using the query based links, what should be included in the SQL property?

Nothing. Rubicon will manage the SQL property.

Q: How can I use TrbSearch as a filter for my DataSet?

In Delphi 2.0 or higher and C++ Builder, you may simply define an OnFilterRecord event handler and test whether the current record matches the search criteria by calling RbSearch1Matches.

Q: MemoryUsage includes what kinds of memory?

It is primarily made up of the memory used to cache the indexes. If AltMemMgr is True, it also includes any memory in the memory pool. Some internal buffers are also included. It does not include the memory used by the container classes themselves, various TLists, and other ancillary data structures.

Q: The blob portion of the Words seems excessively large

Check to see if the table type being used has a default or minimum blob size. If so, see if the default size can be reduced to 32 or 64 bytes. For dBase tables, the MEMO FILE BLOCK SIZE is often set to 1024 or 2048 in BDEAdmin.exe, which is excessively high.

Q: My application seems to stall while using TrbUpdate. Can this be avoided?

If you have set DelayedWrites to True, TrbUpdate will write records to disk when the cache is full. You may use the OnWrite event to do some processing while the cache is being compressed. You should not interrupt this process. Calls to WriteCache or FlushCache may also cause delays. Here, you may abort the process and then resume it later.

Q: (With NexusDB) If I use a pre-made Words table that was previously working with Rubicon v2 + NexusDB v2, then I get the following errors:

Using TRBNXTable:
Service failed on execute: NexusDB: rbNXReqWords: Could not find object. [$2208/8712]

Using TnxTable:
Service failed on execute: NexusDB: rbNXReqWords2: Could not find object. [$2208/8712]

Those are specifically in a Windows service that runs in the background and checks if any updates need to be done. If I even try to add new words to the database, I get the exact same error, "Could not find object. [$2208/8712]".

A: The "Could not find object" error that is being thrown is because of the name of the stream used in the NexusDB Words table (reference rbnx.pas). Version 2 of Rubicon used "RUBICON2" as the name of the stream. Rubicon 4 uses “RUBICON4” instead. To solve this, you have to manually modify the name of the stream in the Words table from "RUBICON2" to "RUBICON4" using the NexusDB Enterprise Manager application.

Troubleshooting

Q: What does the ‘Decompress buffer too small’ error mean?

The error means that there was not enough memory allocated to decompress an index. The allocation of this memory is handled internally and is not affected by MemoryLimit or the amount of installed memory on the computer. This condition is usually caused by one of the following:

1 When updates and searches are occurring simultaneously, the search application may not have allocated enough memory to hold the word indexes (which may have grown in size). In this case, see the code in Button1Click in `RBDemos\demo_b_ttable\ExSrch.dpr` which handles this exception.

2 If the application is placing ranges or filters on the Text table, this will cause the error. In this case, call rbUpdate1.Initialize or rbSearch1.Initialize before setting any filters or

ranges. Also, read the "Filters and Ranges" section in the documentation.

3 This error may result from records being deleted from the Text table without Rubicon being notified. Rebuilding the Words table should solve the problem.

4 This error may result when different settings (e.g. SegmentSize) are used for building versus appending to the Words table.

Q: The first search takes longer than subsequent searches

When the first search is executed, *TrbSearch* needs to initialize itself as well as perform the search. You may minimize the impact of this by performing the initialization earlier by calling *Initialize*. If you are working with SQL tables, you will want to read the section Working with SQL Tables.

Q: A “Dataset not supported” error is raised when using a *TClientDataSet*

Need to include *rbCDS* in the *uses* statement.

Q: Words seem to be missing or incorrectly associated in the dictionary

If the length of the words in question exceeds the *WordFieldSize* property, increase *WordFieldSize* and rebuild the dictionary.

Q: A match table record has a rank of zero.

If *RankMode* is *rmPercent*, it is possible that the rank value is being rounded to zero. This may be checked by changing the *RankMode* to *rmCount* and rerun the search.

Also check to see if the word or words that should have been ranked have a length that exceeds the *WordFieldSize* property. Typically this occurs when using a wildcard in a search. For instance, a search for WaitFor* might locate a record containing WaitForMultipleObjects, but the word found in the index and listed in the *MatchingWords* property is WaitForMultipleObjects. The solution here is to increase the value of *WordFieldSize*.

Q: Searches are not finding the correct records

If a *TrbTextBDELink* is being used with a blank *IndexFieldName*, then a record may have been inserted or deleted in the middle of the table. This will corrupt all the Words table index values and does not generate an error unless the change was made when *TrbUpdate* was running. Otherwise, check for table corruption and use the Verify utility to check the Words table.

Q: A *slOr* search on '*' followed by a *slNot* search should return zero matches, but doesn't

What is being returned are the gaps between index values. Since these records don't really exist, a call to *TrbMatchMaker.Execute* will return an empty table. The correct way to perform the above search is to follow the *slOr* search followed by a *slNot NarrowSearch*.

Q: All the values for *WordCount* and *BlobSize* are zero in my Words table

This should only occur when using *TrbCustomWordsBDELink* components. This indicates that the database format of Words table does not support 32 bit integers. Check to be sure that the *dbiWrite* property is set to False and rebuild the Words table.

Q: Words at the end of memos are not indexed

16 bit applications are limited to memo lengths of 64kb. If possible, compile your application with Delphi 2 or higher.

Q: The *Matches* method does not seem to be working

Matches returns a value that indicates whether the current record in the *DataSet* meets the search criteria. You may have to call *UpdateCursorPos* before calling *Matches*. In addition, when using a *TrbTextBDELink* with a blank *IndexFieldName* and *Matches* is called from within a filter, it may not be possible to synchronize the *DataSet* to the physical record number.

Q: Number of Words table records varies with table type

Normally, the number of unique words should not vary with table type. Differences can arise when the source table(s) contain nonstandard characters that are treated differently by the table types, and therefore result in key violations that cause a word to be excluded from the table. For instance, one table may interpret Canada and Cañada as two different words, the other may treat them as the same (and thus one would be excluded because of a key violation).

Q: Processing *TrbMake.Execute* slows down exponentially

There may be insufficient memory to complete the operation. Check the value of the cache's *MemoryLimit* property. You may need to increase it from the default value of 4mb (advise setting it to the amount of physical memory installed minus 4 to 8 mb for Win9x, half of physical memory for NT). If using Delphi 2 or higher, you may have run into the memory fragmentation bug. Set the *AltMemMgr* property to True. See Memory Fragmentation for more details.

Q: Trouble installing the components; conflict with prior version

If you have installed a prior or trial run version of Rubicon, make sure you have deleted all the old files, especially the dcu/dpl/bpl files. You may want to confirm this by using the Windows 9x or NT 4.0 find utility and search for Rubicon files.

Human Assistance

Do you need further assistance?

Contact HREF Tools Corp. technical support via our newsgroups. You will need to authenticate; generate an account for yourself at *http://www.href.com/newsgroups*. The newsgroups to subscribe to are:

hreftools.public.announce
hreftools.public.rubicon.install
hreftools.public.rubicon.support

You can also contact Technical Support using the form at *http://www.href.com/contact* but we prefer that you ask in the newsgroups because that helps build a public knowledgebase. Newsgroup content is searchable at *http://www.CodeNewsFast.com*.

End Use

This section of the documentation covers performing specific occupational tasks and activities, peculiar to the customers' profession or assignment.

We present a series of real-world solutions that our support department has collected.

Query Based Links

There are two broad groups of Text and Words links: one uses TTable-like components while the other uses TQuery-like components. As a general rule, the first group is best applied to local tables (“briefcase model”), while the second is best suited for SQL tables and views.

When using query based links, Rubicon will manage the SQLText property of your query components, so there is no need to provide any SQL statements.

Query based links behave much the same way as table based links except when it comes to performing wildcard searches. Table based links simply perform a scan, while query based links execute a LIKE query. The difference is that the table scan can be easily aborted, while the LIKE query may not be aborted.

In order to avoid an excessively long query wildcard search, the app may use the OnPreviewWord event to catch vague wildcard searches (e.g. a*). Another approach is to check whether the query component (e.g. TQuery, TIBQuery, etc.) supports a max rows feature. For BDE applications, the dbiSetProp procedure can be used to set the curMAXROWS setting in the TQuery.PrepareCursor method.

Using TrbServerUpdate and TrbSearch Simultaneously

When using TrbSearch while other clients are updating the Text/Words via TrbClientUpdate and TrbServerUpdate, it is possible that the Words will not be in sync with Text as it takes some time for all the changes to be processed. This lag will vary depending on the volume of transactions and "word density" (the number of words changed per transaction -- an edit may have very few changes while an add or a delete changes every word), the bandwidth of the network, hardware, etc.

If your app requires that users be able to search on the most recent changes then:

Set the rbSearch1.Cache to nil
Do not allow searches to be widened or narrowed as old search results may be out of

date
Use the IsCurrent property to determine when the Words is in sync with the Text
If you app does not require access to the most recent changes then
rbSearch1.Cache may be used
Probably want to periodically call rbSearch1FlushCache

Working with Link, Lookup, or Normalized Tables

Performing a text search on a set of linked tables generally requires searching a field in a lookup table, grabbing the index value, returning to the master table, changing the index, finding the index value, etc., etc. Now try performing a complex multi-field search!

Rubicon eliminates this complexity by allowing you to build the Words with a DataSet that contains all the lookups. Just use the Delphi field editor to define the relationships and process the table with TrbMake, TrbSearch, and/or TrbUpdate. Now you can search for any word in any field regardless of whether the field is in the master table or in a detail table.

Working with SQL Tables

Working with SQL tables differs with local tables in only two respects: the choice of drivers and the time required to calculate the IndexRange.

Local table are almost always accessed with TTable-equivalent components. While this approach also works with SQL tables, performance usually benefits from switching to TQuery based components.

All the core Rubicon components need to calculate the IndexRange (the difference between the maximum and minimum locations) before they can perform any work so that they can determine how much memory is needed to hold an individual index. On local tables this simply requires calling TTable.First and TTable.Last, and reading the index field values. The TQuery based components do not use that approach. Instead, they execute a MAX() and MIN() query. For very large tables, this process may be lengthy. In addition, if the field is populated based on a generator (or equivalent) value, it would be much faster to query the generator. This can be done by using the OnMaxIndex and OnMinIndex events and supplying the appropriate values. If you are using a database system such as Microsoft SQL Server which does not use generators, it will be worthwhile storing your minimum and maximum primary key values in a separate table, and selecting those values in OnMinIndex and OnMaxIndex. If your primary key starts at 1, then you can simply hard-code that number as the minimum.

Alert! The same OnMaxIndex and OnMinIndex event handlers should be used for all Rubicon components that access the Text, otherwise there is a risk that the IndexRange will be calculated differently, leading to index corruption.

Using TrbSearch in Conjunction with a TQuery or TClientDataSet

A TQuery may be used in conjunction with TrbSearch. The strength of TrbSearch is its full text search capabilities. However, it is unable to search for text and limit the results to a certain date or numerical range. In order to do this, you simply need to perform the text portion of the search with TrbSearch, create a match table with TrbMatchMaker, and then execute a TQuery against the match table.

If you use this approach, you may wish to index the match table before executing the TQuery in order to improve performance. Remember that all the TrbSearch methods and properties will be unaware of the results of the TQuery (e.g. MatchCount will report the number of matching records of the text portion of the search, not the number of records in the TQuery).

To avoid using the BDE, you can use TClientDataSet instead of TQuery.

Custom Ordering of Search Results

In some instances it may be desirable to order search results in a certain order, but there may not be enough time to rank or sort the match results (e.g. a web app performing numerous searches per minute).

Unless ranked, search results are returned in IndexFieldName order, lowest to highest. If the application is searching a customer table that is indexed on CustNo, and CustNo roughly corresponds to how long a person or company has been a customer (e.g. the lower the CustNo, the longer they have been a customer), then the oldest customers will be displayed first and the most recent customers shown last. To reverse the order, the application could just call FindLast and work backwards through the matching locations as it displayed the results.

But what if the matching records needed to be ordered by the customer's dollar volume over the past year? Without doing extra processing as the search was performed, the only alternative is to add a new index to the Text table and populate with values that correspond to each customer's sales volume, then use this index as the IndexFieldName. Assuming that the customer with the highest sales volume was given and index value of 1, the second highest 2, and so on, search results will always be displayed in sales volume order.

The disadvantage of this approach is that when the ranking by sales volume changes, the index has to be regenerated and the table re-indexed.

Web Applications

Rubicon may be used in web applications in a similar fashion to single user apps. Web search applications do vary in two respects: there are more likely to be multiple versions of the app running simultaneously and web apps often need to save the state of the surfer.

When running multiple copies of a Rubicon search app, each copy is going to have their

own cache, so the host computer should have enough physical memory available for all the apps. If all the apps are searching the same table, then the caches could be shared between them.

Saving the surfer's state is usually the responsibility of the web framework being used. These frameworks often provide a mechanism for saving standard data types. To save the state of TrbSearch, the app will have to save at least the SearchFor property and the matching locations (unless the search is re-executed). Since SearchFor is a string, most frameworks can readily handle saving this property, whereas saving the matching locations requires a bit more planning.

The easiest way to save the matching locations is to use the TrbSearch.MatchBits.AsText property. This will return the matching bits as a compressed UUEncoded string. To reset the matching locations, just assign the string back to AsText. MatchBits also supports a CommaText property which returns the matching locations as a string of comma delimited locations. Because CommaText tends to return much longer strings, using AsText is the preferred approach.

International Character Issues

International characters require some special consideration. When dealing with international characters, the sort order and character set of the Words table is very important.

An excellent overview of character sets as used in Firebird SQL is online here: *http://www.destructor.de/firebird/charsets.htm*.

In the Borland Database Engine, sorting and character set are controlled by the table's language driver. Regardless of which database you use, sorting international content is tricky. Using the BDE as an example, depending on the language driver, a table may be sorted as follows

International	AscII
RESUME	RESUME
RÉSUMÉ	REVIEW
REVIEW	RINK
RINK	RÉSUMÉ

The terms used to described these sort orders vary, but they will be referred to here as Intl and AscII. Both sort orders are compatible with Rubicon, but they do affect the search results in many cases.

The table's character set often determines how characters #128 and above appear in the table. In some instances, characters #128 and above are converted into another character.

To check for this, open the Words table after it has been built and make sure that international characters appear correctly in the table. If international characters have been converted, you must select another character set or language driver.

Regardless of the sort order, using Rubicon's default configuration, a search for a specific word will find that word. For instance, if resume and résumé are in the text, then a search for resume will find resume, and a search for résumé will find résumé.

While it is always preferable that the user enter their search using international characters (if any), this may not always be the case. In the previous example, the user may enter resume, but really be looking for résumé. Since Rubicon is not telepathic, the best it can do is return both words (if the user enters résumé, it will return resume as well). In order to enable this behavior, the sort order must be Intl and the TrbSearch International property must be set to True.

The International property will also affect how wildcard searches are conducted. When False, a search for re* would return resume, and review while a search for ré* would only return résumé. When International is set to True, either search would return all three words.

When the International property is set, Rubicon normalizes the words internally. If the search is for ré*, it would be normalized to RE*. Likewise, the word RÉSUMÉ (it is already uppercase in the Words table) would be normalized to RESUME. The normalization process is handled by the Normalize procedure in rbUtils.pas. Some character sets may require custom normalization, and this can be implemented by reinitializing the NormalTable.

Searching External Files

Data outside of a database – typically files on a hard disk drive – may be indexed and searched if a database can be created that references this 'external' data and a routine is supplied that can read and parse the data (i.e. remove any formatting codes).

The ExHTML.dpr application in the `Rubicon\RBDemos\demo_b_ttable` subdirectory demonstrates how to do this for HTML documents. (You must install the BDE bridge to install these example projects. You can then save-as the PAS files to your own directory and modify them to use your own bridge.) See the comments in ExHTMLU.pas for details. ExRTF.dpr is a very similar program that works with text and RTF files.

The approach used in the application is to create a table whose records reference all the files to be included in the search. If the files are small, it may be easiest to simply copy the file into a memo field and then use the Rubicon components in a normal fashion. Usually, it is better just to reference the file by filename (including drive and path). The remainder of this discussion assumes the latter.

Once a table of filenames has been created and populated, it needs to be processed by Rubicon. In the TrbCustomTextLink component, the field name containing the filename is added to the FieldNames property. Of course, the field itself does not contain the text that needs to be indexed, so an OnProcessField event needs to be supplied that reads and parses the file. For a text file, the event would look something like:

```
procedure TForm1.Form1ProcessField(Sender: TObject; Engine : TrbEngine;
Field: TField);
var List : TStrings;
begin
      List := TStringList.Create;
      try
            {* Field contains the filename *}
            List.LoadFromFile(Field.AsString);
            Engine.ProcessList(List,True);
      finally
            List.Free
      end
end;
```

Warning: Any time an OnProcessField event is used, it must be connected to all TexLinks that access the text in order to insure consistent treatment of the text.

When the table is processed with TrbMake each record will be indexed with the words contained in the external file.

As the files change, the database will have to be updated. If the size of the database is small, it is probably best to simply rebuild it from scratch. However, this may not be practical for large databases. Updating presents a problem because unlike a conventional database, TrbUpdate will not be notified when a file changes, and thus it cannot keep track of which words were added/deleted to a record/file. This forces TrbUpdate to assume that all the words in the previous version of the file were deleted and all the current words in the file were added.

Fortunately, the routine ClearLocations makes this complicated process pretty fast and straightforward. All the application needs to do is identify which files were deleted or changed, pass this information to ClearLocations. The method will then scan though all the words and indexes in Words and remove any references to the deleted or changed locations. All that is left to do is process files that have changed or been added with BatchAdd.

Converting Words

Rubicon includes a Convert method and a Convert utility that may used to change the structure of the Words. This is most useful for changing the values of BlobFieldSize, BytesFieldSize, CharFieldSize, and SegmentSize. You may also convert the Words table from the segmented structure to a non-segmented structure (to do this, set the SegmentSize to zero), and visa versa.

Changing the values of BlobFieldSize, BytesFieldSize, or CharFieldSize changes the amount of index information held in the record structure versus blob storage. Generally, if the entire index can be stored in the record structure, access time for writing and reading the index is reduced because there is no need to read the blob data. This improves performance for index creation, updating, and searching. It may also reduce the overall size of Words table, especially if there is a large amount of overhead for blob storage.

Changing the value of SegmentSize can change the number of segments in the Words table. When the number of segments is reduced, the overall size of the Words table will also be reduced and search performance will improve slightly (but probably not enough to be noticed). However, fewer segments will degrade update performance, so the tradeoff may not be worthwhile. Increasing the number of segments by decreasing SegmentSize has the opposite effect: table size will increase but update performance will improve.

Changing the SegmentSize to zero converts a segmented Words table to the non-segmented structure. This will reduce the Words table file size by 20-30% and slightly improve search performance, but updates should not be performed on the converted table. Conversion makes sense when the Text is relatively static and storage space is at a premium.

A non-segmented Words table may be upsized to the segmented architecture by changing the SegmentSize from zero to a positive value.

All the conversions discussed in this section will have no affect on the memory resources required to perform a search. Changes that reduce the Words table size will reduce the amount of data read from disk, network, or server.

Searching Short and Omitted Words

For searches using slAnd, slOr, and slNot SearchLogic, words with fewer than MinWordLen characters and words in the OmitList are ignored. If MinWordLen is 4, then a search for "red carpet" would be the equivalent to searching for "carpet". Since Rubicon cannot tell whether "red" appears in the matching records, the word "red" is not included in MatchingWords.

For slPhrase and slNear searches, all the words entered in the search are used regardless of their length or whether they are in the OmitList. Since Rubicon does check to see that all the words are present, all the words will appear in MatchingWords.

When words rejected by an OnAcceptWord event are used in a search, the search will fail. If OnAcceptWord rejected the word "only", then any search except slLike would find zero matches, including slPhrase and slNear. Since OnAcceptWord is geared toward rejecting noise words, the chances of one of these words being used in a search are low. However, if you wish to have a word rejected by OnAcceptWord treated the same way as words rejected by MinWordLen and OmitList, then test each word in OnPreviewWord, and set the length of a rejected word to zero.

Searching without a Text DataSet

Under some circumstances it may be desirable to be able to perform searches, but not actually connect TrbSearch to the Text dataset. To do this, TrbSearch must still be connected to a TrbTextDataSetLink descendent, but the DataSet property (usually Table or Query) is left nil. The OnMinIndex and OnMaxIndex events must be used so that TrbSearch can calculate the size of the indexes. Under this configuration, no proximity searches may be performed nor may the SubFieldNames property be used else an rbeInvalidTextDataSet error is raised.

An example use-case for this feature follows.

In some cases, stepping thru results can be slow because for each step a query is executed (select * (or just the indexed field) from TextTable where ref = x) and this may give you only a small part of what you really need. For example, you may need to join all the resulting records to other tables before displaying them to the user and/or sequence the results. Thus what you can do is step thru the results and insert all the keys into a global temporary table (do all the insertions within a transaction for speed), then join that temp table to all the other relevant tables with an order by clause. This can be orders of magnitude faster.

There is a search results retrieval method which can be used to quickly obtain the list of matching keys for this sort of solution: `rbSearch.MatchingLocations`.

Usage example:

```
var
      list: TList;
      int: Integer;
begin
      rbSearch.Execute;

      list := nil;
      try
            list := TList.Create;
            rbSearch.MatchingLocations( list);
            int := Integer(list[0]);
      finally
            FreeAndNil(list);
      end;
end;
```

Memory Issues

The only Rubicon component that uses a significant amount of memory is TrbCache. The amount of memory it uses can, in most cases, be capped by using the MemoryLimit property.

TrbCache can be connected to TrbMake, TrbUpdate, TrbServerUpdate, and TrbSearch. Of these, only TrbMake requires a cache be assigned (although it is recommended for the other components) because it builds indexes entirely in-memory. As a result, TrbMake is the most memory intensive Rubicon component. The other core components use relatively little memory, however performance will benefit if additional memory is made available to cache indexes.

TrbMake Memory Requirements

The amount of cache memory required to build the Words is approximately:

# of unique words * index size * (1 - compression rate) / 8

With non-segmented Words, index size is the IndexRange (the difference between the lowest and highest location values). When segmentation is used, the index size is the lesser of the IndexRange or the SegmentSize.

Applying the formula to a table composed of one million records and 5,000 unique words using an efficient index and a 97% compression rate would require 18.75mb of virtual memory.

Since the amount of virtual memory available is not always clear, TrbMake will keep consuming memory as it needs it until it runs out. If you want to set an absolute limit on the

amount of memory available to the component, add an AfterProcess event handler to monitor MemoryUsage and abort the process once the memory threshold has been exceeded.

32 Bit Memory Fragmentation

Rubicon caches and compresses indexes in memory in order to minimize disk/network activity. In doing so, it is frequently disposing large blocks of memory for small ones, or visa versa. Unfortunately, this pattern of behavior is the Achilles' heel of the 32 bit memory suballocator and eventually leads to massive memory fragmentation which will grind the application (but not the system) to a halt.

Fragmentation usually does not become a problem unless the Text has more than 50,000 locations and 15,000 unique words. This is an approximate threshold, and will vary with the amount RAM devoted to caching. The problem is most likely to affect TrbMake since it goes through the most compress and decompress cycles during execution. TrbUpdate may be affected if a very large number of records are updated during execution and caching is enabled. TrbSearch should not be affected even if caching is enabled since the number or records cached is likely to be very small.

Tools such as MemorySleuth 1.x do not catch this bug. The Windows 9x System Monitor or NT Task Manager will. You may use one of these tools to determine whether your application is being affected by fragmentation. If the tool shows memory use increasing even after the component has reached its MemoryLimit and the performance of the application is degrading, then fragmentation is the likely cause.

Fragmentation does not lead to a memory leak. All memory used by the components are returned to the system when they are freed or done processing.

Rubicon provides an alternative memory manager which works around this bug. It uses an algorithm that is optimized to work with the TrbMake pattern of memory use. To use this option, the AltMemMgr compiler directive in RBDEFINE.INC must be enabled and the AltMemMgr property must be set to True. This option does not replace the existing memory manager (i.e. it does not call SetMemoryManager), but rather supplements GetMem and FreeMem.

Unlike TrbMake, TrbUpdate has a much more unpredictable pattern of memory use so it is more difficult to assure that the alternative memory manager will not also defragment memory. If you are processing a large number of changes to a table and are using caching, then you may wish to call FlushCache periodically.

The alternative memory manager eliminates the fragmentation problem by creating a list of pointers available for reuse (a memory pool). When execution begins, this list is empty and requests for memory are passed to GetMem. As execution proceeds, any memory that is released is saved in the memory pool. Subsequent requests for memory first check the memory pool to see if there is a pointer available of the appropriate size. If one exists, it is used, otherwise GetMem is called.

When the alternative memory manager is used, MemoryUsage may exceed MemoryLimit

by a large amount. MemoryUsage is largely made up of the memory used to hold data structures and the memory pool. The MemoryLimit is compared to only the portion of MemoryUsage that is actually holding data, and thus the memory pool portion is excluded.