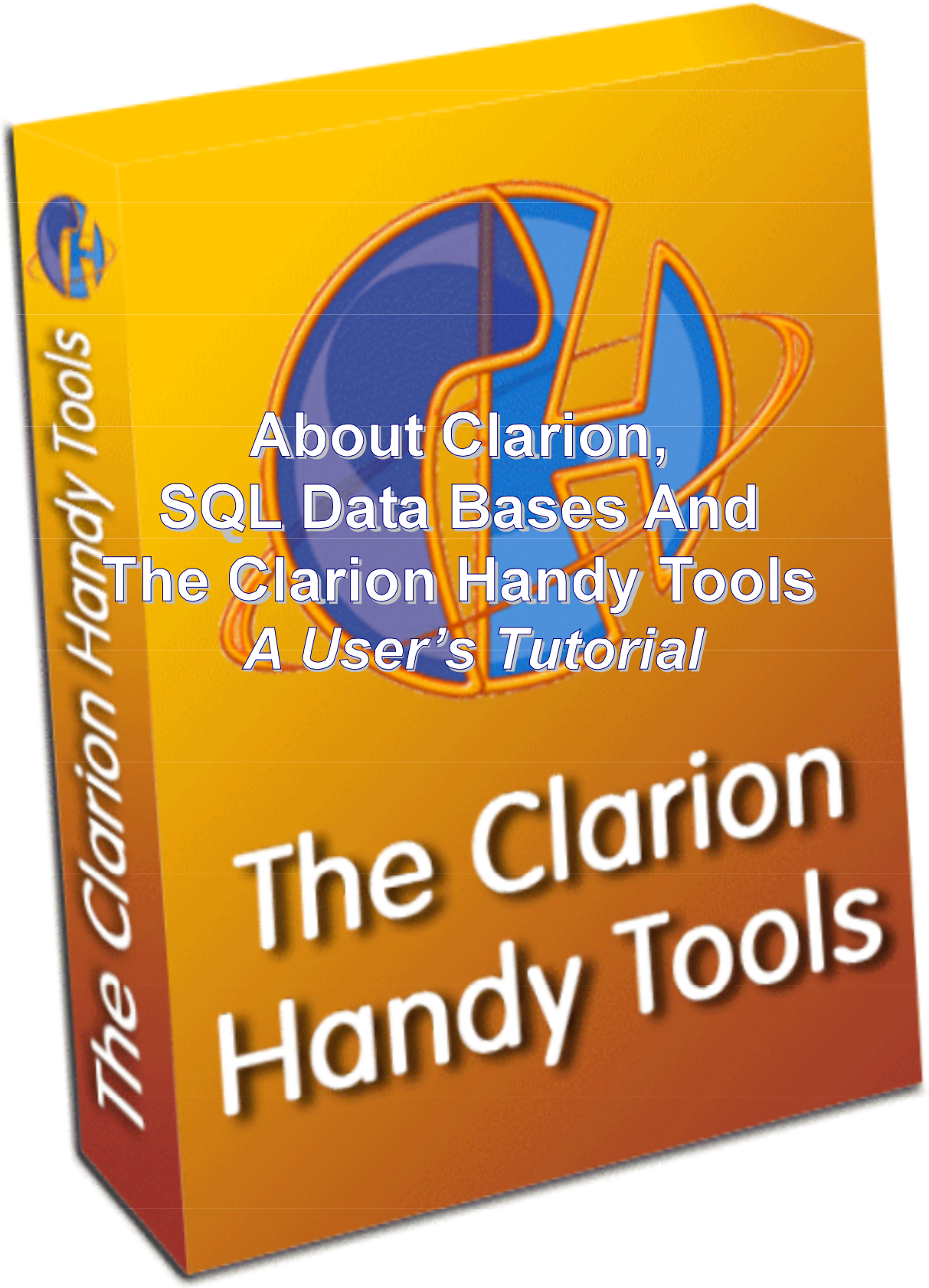

The Clarion Handy Tools
About Clarion,
SQL Data Bases And
The Clarion Handy Tools
A User's Tutorial
The Clarion
Handy Tools

The Purpose Of this Application (HNDACCES.APP)

In really general terms, this demonstration application, like all CHT demos, is intended to illustrate some aspect of The Clarion Handy Tools templates and classes. It's important to grasp that the CHT tool kit is an extension of Clarion ABC. It integrates with ABC and works and behaves like ABC. The CHT templates, in and of themselves - just like the ABC templates - don't do anything much more than generate interface code to the underlying CHT and ABC classes. When you embed into an ABC method inside one of your procedures using the Clarion Embeditor, the template derives the class containing that method in such a way that the original ABC class is now customized, or behavior-modified, by your embed. The Clarion Handy Tools templates and classes work exactly the same way. The CHT templates, again like ABC, provide an interface layer between the developer and the underlying classes. They ask you questions in the form of template prompts and generate code resulting from the information garnered from you. That generated code is almost always some kind of property-initialization code that prepares the class-to-be-called, for work to be done, followed by one or more procedure/function calls into the target class.

Since what the templates do is write custom interface-to-classes code on your behalf, based upon your answers to template prompts, it stands to reason that you can write this code yourself too, via embedding. HNDACCES.APP provides some examples of embed code as well.

HNDACCES.APP illustrates Clarion plus an SQL data base plus The Clarion Handy Tools working together. It is not intended to be an example of good window design. In fact, the window designs are, in most cases impractical for everyday purposes. They're intended to illustrate SQL concepts and CHT features, nothing more. Before we launch you into the specifics of the application here is some background about Clarion and SQL data bases.

Clarion Has SQL Capability Built In

One of the many things The Clarion Handy Tools can help you do is work more efficiently and easily with SQL data bases. This tutorial, in conjunction with the demonstration application that accompanies it, discusses a variety of ways in which you can use The Clarion Handy Tools to help you produce better Clarion applications that utilize SQL back ends. First, some background on what Clarion can do *alone* with SQL data bases.

Clarion, in and of itself has the capability to communicate with SQL data bases built in, so that even without The Clarion Handy Tools, you can build good, solid SQL applications. But there are a few things you may not be aware of that Clarion applications do, as regards SQL back ends, that are not necessarily optimum SQL practice.

If you're new to SQL you may not be aware that SQL works by request and response to and from a back end data base. That is to say, you send a request - an SQL select statement - to the data base and you get back a response (some data). The response is the set of records matching your request. By the way, if you *are* new to SQL, here's a website that teaches SQL syntax that may prove to be a helpful reference: http://www.1keydata.com/sql/sql.html

When you use Clarion and you create a "filter" on an orders browse to say that you only want to see orders for a specific customer that you've selected in the customer browse, that relationship can be expressed by a simple filter such as this: ORD:CustomerID = CUS:CustomerID. This kind of filter is often referred to as a "join" or a "join statement". It's also a very efficient filter that can readily be handled very smoothly by all data bases ISAM or SQL.

This join statement says to the orders browse, show me only the orders where the condition ORD:CustomerID = CUS:CustomerID is true. When you select a record in the customer browse, the right side of the join statement is "resolved" to the actual value of CUS:CustomerID for the selected customer record, so that a query in fact might be: Ord:CustomerID = 9342342. In order for this condition to result in some records that display in your orders browse a request is sent to the SQL data base for a batch of fields and records. That request could look something like this once translated into the SQL "language".

```
SELECT
ORD.`ORDERID`,ORD.`CUSTOMERID`,ORD.`EMPLOYEEID`,ORD.`ORDERDATE`,ORD.`SHIPPEDDATE`,
ORD.`SHIPVIA`,ORD.`SHIPPOSTALCODE`,DET.`ORDERID`,DET.`PRODUCTID`,DET.`UNITPRICE`,
DET.`QUANTITY`,DET.`DISCOUNT`,PRO.`PRODUCTID`, PRO.`PRODUCTNAME`
FROM `ORDERS` ORD, `ORDER DETAILS` DET, `PRODUCTS` PRO
WHERE ( ORD.`CUSTOMERID` = 9342342 )
ORDER BY {fn UCASE(PRO.`PRODUCTNAME`)}
```

Components Of An SQL Data Request

This SQL data request has four components, a SELECT component, a FROM component, a WHERE component and an ORDER BY component. Only the SELECT and FROM are normally required. Leaving out the WHERE clause results in receiving the entire data set. Leaving out the ORDER BY clause results in an unordered data set. Notice that the filtering and ordering operations of an SQL statement are separate. This is a critical difference between SQL and ISAM that very often necessitates significant data design changes between ISAM and SQL applications that appear to do the same thing. Many of the changes have to do with how the tables are keyed. A lot of the time SQL keys can be less complex, for the very reason that ORDER and WHERE can be handled separately.

The SELECT component enumerates the fields that you want to see in your browse. Clarion does this very well and automatically for you based on the fields that you've populated on your browse.

The FROM component enumerates the data bases from which the selected fields will come. Clarion does this very well and automatically for you based on the TABLE relationships that you design in your dictionary and the TABLE SCHEMATIC you lay out when you create your browse.

The ORDER BY component is normally determined by the key placed on the browse tab that's currently in use. Clarion does this reasonably well, and automatically based on the keys declared in the dictionary. On the other hand, SQL data bases are perfectly capable of returning ordered data sets even though there may not be a key for the requested data order. Hence, the one-key-for-every-browse-sort-order-used principle applied by Clarion browse templates is more of a hang-over from the ISAM files days than a SQL requirement.

The WHERE component is the "filter" aspect of your browse. When you determine that your orders browse should display orders for a specific customer selected in a "parent" browse, you'll use what's known as a "join" (the customer browse and the orders browse are joined via a common field in both data tables). Joins are expressed in SQL in the WHERE portion of an SQL request string. It's the WHERE clause of an SQL request that stops the data base from sending you the entire data set when you only want to see a specific subset. It's the WHERE clause that often determines the speed of your browses, reports and processes. Bad WHERE clauses cause slow access, when they cause the data base to send you more records than you need to fulfil a specific requirement.

Often, Clarion - right out of the box - does this WHERE stuff very well, and very automatically for you based on the Clarion filter string that you apply to your child browse via the browse template. Clarion uses an ABC browse class function called SetFilter() to apply a filter to the browse and then you apply a "reset field" on the browse to indicate that the browse should be reset when the

value in the reset field changes. So, when you select a new customer record in the parent browse, the child browse realizes it must reset itself because the value of the reset field (CUS:CustomerID) has changed. A new request goes out to the data base and a new data set comes back and is displayed in your orders browse.

About Joins, Filters And Range Limiters

If you're new to SQL it may not even have occurred to you that you can "join" two data bases like this via a filter and a reset field. Coming from ISAM data bases - TPS files, for example - your first inclination is to do this via a "Range Limiter". If that's the case, be aware that a range limiter, just like a filter, ends up being part of an SQL request string with the range limit portion resulting in a WHERE clause. The ABC templates and classes, in conjunction with the file driver eventually translate the range limit request into an SQL WHERE clause which is sent to the data base as part of a data request string or "select statement". So there is seldom, if ever, an advantage to using a range limiter in an SQL application. In fact, as we will explain below, range limiters can encourage keying practices that are, at best, unnecessary with SQL data bases and, at worst, result in bad SQL request strings.

In Clarion applications that use SQL data bases there's no guarantee, that the application of a range limiter is going to be faster than using a simple filter. The speed comes from how easily your range limiter or your filter statement can be translated into the SQL language by the Clarion file driver. If the range limiter and the key definition on which it is based are too complex or the filter statement you've written cannot be directly translated into SQL, bad things can happen to good applications. When this situation arises, your filters or range limiters of this type will still work, *but the end result can often be unsatisfying slow*.

Here you have a case, then, of Clarion being *too powerful*, or perhaps *too nice* to its developers to tell them that what they're doing is going to result in badly formed SQL statements and speed-crippled applications. In a worst case situation where the Clarion file driver realizes it hasn't a prayer in translating your range limiter or filter into a proper SQL statement, the file driver simply requests the entire data table and then filters it locally (client side). While that may achieve the end-result you want, it's truly ill advised to let this scenario unfold in high-use situations or situations where data base sizes are greater than a few thousand records. SQL data bases are, in effect, intelligent "data servers" and they are perfectly capable of filtering data sets before they're sent across the network or the internet to the connected client - as long as the SQL request strings are well formed with good WHERE clauses. It's folly to design applications that filter at the client side, unless you're using ISAM files where there's no other viable option.

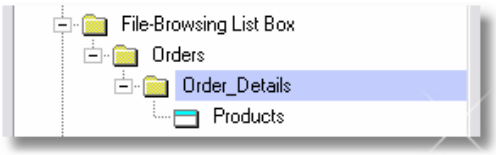
About INNER JOINS and OUTER JOINS

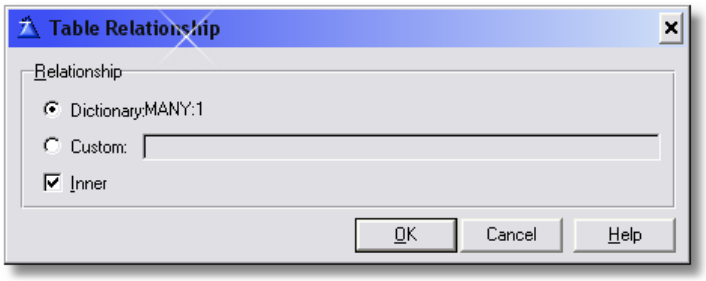
With keyless columns and dynamic, on-the-fly sorts, certain problems can arise when two or more tables are joined in the Clarion dictionary and are presented together in one view. Your browse presents a single view with columns displayed from, say, three different tables acting as if they were one table. In that case, an internal join statement, generated by the Clarion file drivers, is appended into the SQL SELECT. By default, Clarion browses always use an OUTER JOIN to accomplish this. However, to resolve the elements of an OUTER JOIN, the Clarion file driver relies heavily on the current key in place on the column, and you haven't supplied a key on at least some of the browse columns because we've told you that CHT applies sort orders dynamically – whether a key is available or not.

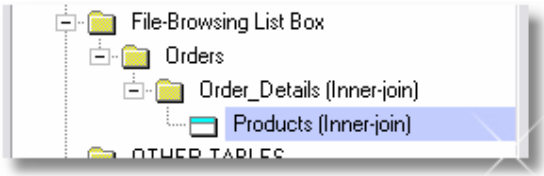
Consequently, the SQL code generated by the Clarion file driver to express this conjunction-of-three-tables-as-one can end up being incorrect. To further compound the problem and the possibility of the file driver making an error, the syntax for OUTER JOIN expressions varies vastly from one flavor of SQL to another, so much so that one wonders why INNER JOINS are not the

default, with OUTER JOINS applied where necessary, by developer intervention. In some data base back ends OUTER JOINS are expressed as part of the WHERE clause while in others they are separate expressions.

Suffice to say that where such issues arise, you can in many cases simply suggest to the ABC browse template's table-schematic that it should use an INNER JOIN. Whether you *need* to do this or not may depend entirely on which data base you're using. But using an INNER join can solve a multitude of what appear to be unrelated problems, from duplicate browse records to queries that won't resolve into all of your tables, to update forms that throw errors when new records are inserted. Here in pictures, is how to apply an INNER JOIN.





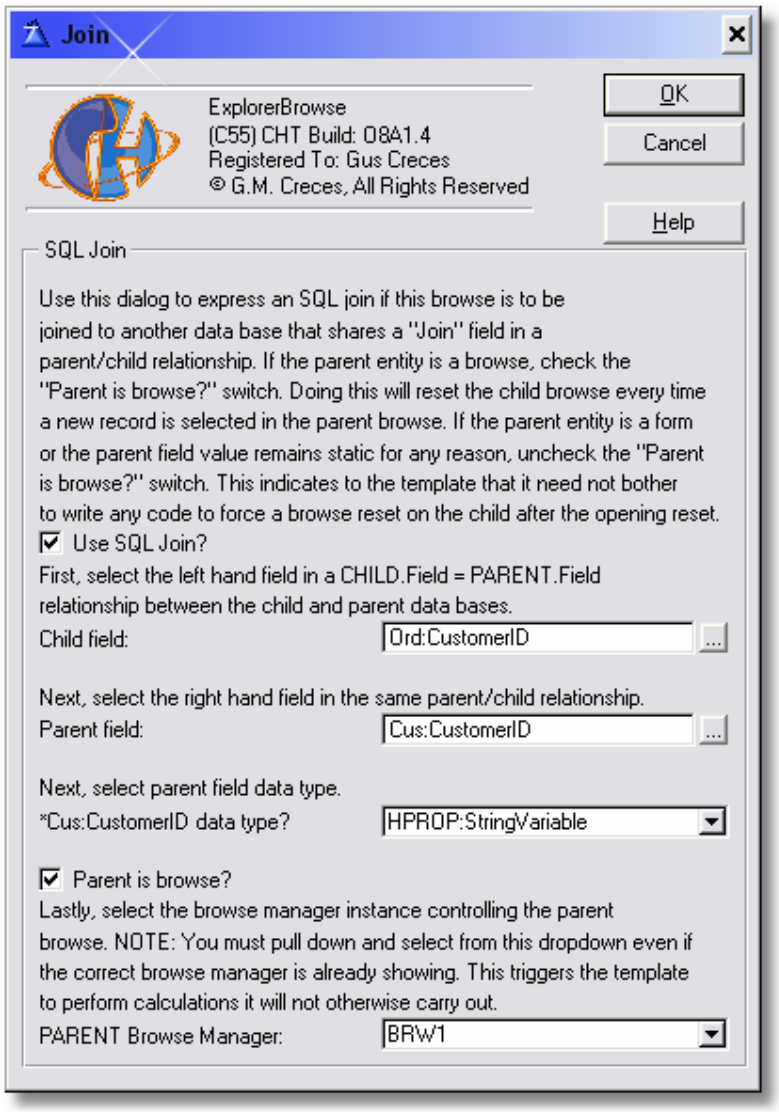


Joins That Keep Separate Browses In Synch

You need to keep two separate browses displayed on the same window in synch as illustrated by HNDACCES.APP. The browse procedure entitled **SQLFilter_ExplorerBrowse** provides a real-life example of this. A customer browse is used to select or locate a specific customer and an orders browse displays that customer's orders, including detail lines incorporating product descriptions and quantities. To make this happen, some mechanism must be put in place to keep the CHILD browse, orders, in synch with the PARENT browse, customers, any time a new customer record is selected.

CHT provides you with a one-stop dialog where you can describe the relationship established between these two separate browses. This, of course, assumes you've actually set up a relationship in your data design. The CHT template dialog to consult in this case is called "Join". Here, again illustrated with pictures, are the join settings on **SQLFilter_ExplorerBrowse**.

See also illustration, next page.



That results in the CHT template writing the following Clarion code into our HBrwx.AutoInit() method where CHT browse object initializations occur.

```
SELF.SetSQLJoinFilter(Cus:CustomerID,'Ord:CustomerID','Cus:CustomerID',|
HPROP:StringVariable)
```

An SQL trace of a SELECT statement on this browse looks like the one below (next page). Note that the section illustrated in orange text, below, is caused by the join settings illustrated above. It's productive to point out also that the **AND** portion of this WHERE clause was written by the Clarion file driver based on the INNER JOIN described between the tables ORDERS, DETAILS and PRODUCTS. In the background Clarion looks after keeping your all-in-one-view tables in synch while y*our* join concerns are with displaying only orders where ORD:CustomerID = "ALFKI" (The resolved right-hand-side of this expression is dependent on which customer is selected in the customer browse).

```
SELECT
ORD.`ORDERID`,ORD.`CUSTOMERID`,ORD.`EMPLOYEEID`,ORD.`ORDERDATE`,
ORD.`SHIPPEDDATE`,ORD.`SHIPVIA`,ORD.`SHIPPOSTALCODE`,DET.`ORDERID`,
DET.`PRODUCTID`,DET.`UNITPRICE`,DET.`QUANTITY`,DET.`DISCOUNT`,
PRO.`PRODUCTID`,PRO.`PRODUCTNAME`
FROM `ORDERS` ORD, `ORDER DETAILS` DET, `PRODUCTS` PRO
WHERE ( UCASE(ORD.`CUSTOMERID`) = 'ALFKI' )
AND ORD.`ORDERID`=DET.`ORDERID` AND DET.`PRODUCTID`=PRO.`PRODUCTID`
ORDER BY {fn UCASE(PRO.`PRODUCTNAME`)} DESC
```

One other little detail that you don't have to think about and which CHT templates handle for you automatically as a result of your having completed the "Join" dialog, is resetting the child browse when the parent browse receives a new selection. In the BRWx.TakeNewSelection() function servicing the parent browse the following code is embedded by our template:

```
BRW1.TakeNewSelection PROCEDURE

  CODE
  PARENT.TakeNewSelection
  !--- Start: HNDTOOLS.TPL O8A1.4 © 2004 (ExplorerBrowse) ------------
  !Services the SQL join between this browse (the PARENT, named: BRW1 )
  !and a child HNDBrowse Instance named: HBrw8
  !The child browse resets whenever (Cus:CustomerID) changes value.
  HBrw8.SQLResetQueue(SELF.ILC.GetControl(),False)
  ! --- End:   HNDTOOLS.TPL O8A1.4 © 2004 (ExplorerBrowse) -----------
```

Our template-embedded comments pretty much say it all. Whenever CUS:CustomerID changes, because a new record has been selected in the PARENT browse, the CHILD browse is issued a forced reset. This causes the WHERE clause to be resolved differently.

```
WHERE ( UCASE(ORD.`CUSTOMERID`) = 'XXXXX' )
```

And results in a new set of order records being displayed. Total lines of embed code written so far to make all this happen, *zero*.

Data Design And The Danger Of Certain Assumptions And Practices

Over the years I've had many after-the-fact consulting encounters with companies who had just finished moving their commercial software application from an ISAM environment to an SQL environment only to find that the reworked-for-SQL application performed badly, and in some cases it didn't work at all. If that transition didn't drive them out of business altogether, as often as not, fingers were pointed squarely at Clarion as the source of the problem when in fact the problem was really a combination of things, including data design issues that could easily have been resolved:

1. With a long, hard look at the data dictionary.
2. With a few weekends spent reading an SQL book.
3. With some considerable time spent understanding how Clarion actually works and interacts with SQL back ends.

While it is accurate to say that Clarion works equally well with ISAM files and with SQL data bases, it is not equally accurate to say that well-designed ISAM applications necessarily work well, or at all, if configured to use an SQL data base back end.

There are any number of possible strategies developers might use to try and improve his/her SQL applications - better data design is perhaps the most important of these. If you're not prepared to change your data base design for SQL (where necessary), you should probably stay FAR AWAY from SQL because you'll most certainly screw it up. Once again, while it is accurate to say that good data design is important both for ISAM files and SQL data tables, it is not equally accurate

to say, or assume, that a good ISAM data design will work equally well in an SQL environment. One of the main rules of thumb you should follow with SQL is to use a numeric integer, unique ID field in EVERY data table and place a primary key on that field. That primary key should NEVER be a compound key. A corollary to that rule of thumb is to ALWAYS create your joins using this unique ID, and NEVER make joins using compound keys. Range limiters as implemented in Clarion's templates encourage complex, compound keys because with ISAM files, where the range-limiter idea originated, the concepts of joining and ordering child tables are often inextricably intertwined. Yet very simple relationship statements like Ord:CustomerID = Cus:CustomerID are all that are EVER required to express a join between two data tables when you separate the job of joining and ordering into two operations the way that SQL does with WHERE and ORDER BY.

The Merits Of View{PROP:SQL}

Another strategy often used by developers to make their Clarion SQL applications work better, is to intervene heavily in the normal operation of the Clarion ABC templates and classes. They write and send their own SQL select statements to the data base via View{PROP:SQL}. This is an acceptable strategy as far as it goes if you know what you're doing and have both strong SQL skills *and* strong Clarion embedding skills. Relying on this heavily kind of defeats the purpose of having a sophisticated code generation system like the Clarion IDE. If it has to come down to writing your own SQL embeds in a Clarion browse you had better know what you're doing or you may find yourself working at cross-purposes with the natural flow and placing embeds to work around ABC behavioral changes caused by your own embeds.

It's perfectly clear that while using View{PROP:SQL} is one way to guarantee the quality of the SQL (assuming you know good SQL) we've already outlined that the SELECT and FROM portion of an SQL data request are hardly ever the problem, while the WHERE clause is the most frequently the problem and the ORDER BY component can be used more flexibly than the ABC browse template lets on. In other words, inside ABC, resorting to View{PROP:SQL} is most of the time not necessary.

The Merits of View{PROP:SQLFilter}

A better strategy is to intervene only in the WHERE component of an SQL data request string. You can do this using View{PROP:SQLFilter}. In this case, the SELECT, FROM and ORDER BY components of the data request string are handled by the Clarion driver and the WHERE component is, at least partially, handled by the developer. Armed with this bit of knowledge you might set about improving your SQL application and ensuring that all data requests are filtered by the data base engine, server side, and not by your application at the client side of the data transaction. This assumes, of course that you know what a well-formed SQL WHERE clause looks like and you know where to tell the ABC templates and Classes to apply the View{PROP:SQLFilter} statement. This is the strategy used by The Clarion Handy Tools. If you ask them to, they'll apply View{PROP:SQLFilter} on your behalf, ensuring first, that the SQL WHERE clause being applied is optimum SQL and that it's done in a way that co-operates with the ABC Classes.

The Merits Of Clarion Handy Tools Query Language

Almost eight years ago, The Clarion Handy Tools introduced a query control template called *QueryParsingFilterControl* that works hand-in-hand with other CHT and ABC templates on all browses, processes and reports to easily provide optimized filtering that works seamlessly in ISAM or SQL browses. *(SQL capability was added more recently)*. CHT query language is a simplified, real language way of expressing a filter statement that can be adapted on the template to work in English, Dutch, Germany, Spanish, in fact in any language you want it to. Its chief

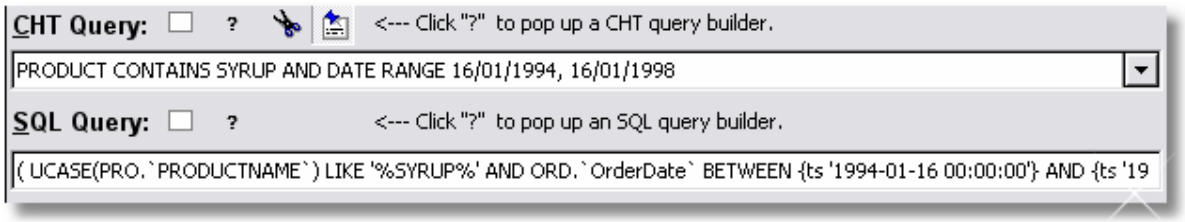
purpose is to provide a simplified end-user accessible way of writing dynamic filter statements in your applications.

Here is an example:

```
PRODUCT CONTAINS SYRUP AND DATE RANGE 16/01/1994, 16/01/1998
```

Translated to an SQL WHERE CLAUSE by our built-in CHT query parser this becomes:

```
WHERE (UCASE(PRO.`PRODUCTNAME`) LIKE '%SYRUP%' AND ORD.`ORDERDATE` BETWEEN {ts '1994-
01-16 00:00:00'} AND {ts '1998-01-16 23:59:59'})
```



Translated to a Clarion filter statement by our built-in CHT query parser this becomes:

```
MATCH(PRO:ProductName, '*SYRUP*', 17) AND ORD:OrderDate_Date >= 70511 AND
ORD:OrderDate_Date <= 71972
```

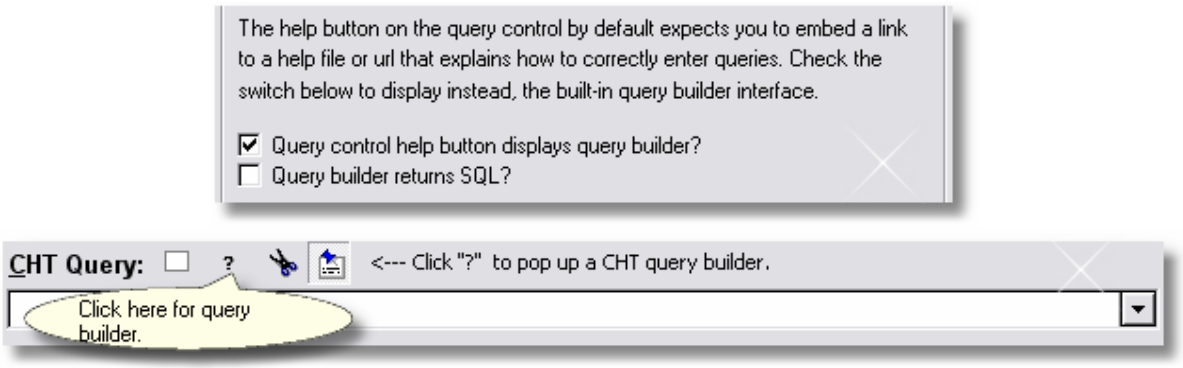
If we send this Clarion filter statement to an SQL data base via the Clarion ODBC file driver - the way you would with ABC.SetFilter() - the end result bears little or no resemblance to the SQL translation we gave you above. In fact what the driver does is something equivalent to requesting the entire orders data table. Then it filters the returned data locally, as if the data base were an ISAM file, using the Clarion filter string provided. Not all pure-Clarion filter statements sent to an SQL file driver would result in this ISAM-like behavior. In fact good data design, proper use of keys and care in the ordering of filter terms will help improve the chances of a good translation to SQL. But this is something you'd have to get used to doing. We're only pointing out, as we did above that what works fine in an ISAM application doesn't necessarily work well in an SQL application. One thing is certain, if you expect your users to be able to create queries for themselves, their chances of doing it properly are a lot higher with CHT query language than with either SQL statements or Clarion statements because CHT query statements resemble real language.

Real Language Queries VS Query By Example

CHT query language resembles the user's own language and because all human beings have built-in language processors, that pretty much guarantees that real-language queries are faster to produce, reuse and understand than clicking through several layers of a query wizard tool.
Here's a developer quote about his experience with just such a query tool:

> "I looked at Camtasia but it is overkill for this particular app. EZHelp works great, but needed something more to explain setting up a filter with Qxxxx Wxxxxx, which generates 98 per cent of the help calls. My clients are not well versed in computer logic."

Logic has nothing to do with it. It's experience. Everybody has experience with language. Few people have experience with multi-layered query tools. Having said that, there *are* times when an easy-to-use query builder can come in *handy*. ***In the O8A1.0 build we've given you a built-in query builder support tool for query-shy developers and end users.*** The question mark button populated by the *QueryParsingFilterControl* template can be made to pop up the CHT query builder by setting a switch on the template.



The tool is tightly based around the CHT query language, which it produces, and it provides memory assistance with query keywords and saves having to type column names. The ultimate question of any query-builder tool is: "*Can it write good SQL?*" If not, you haven't done a thing to improve the SQL performance of your application. The new CHT query-builder provides a dual benefit:

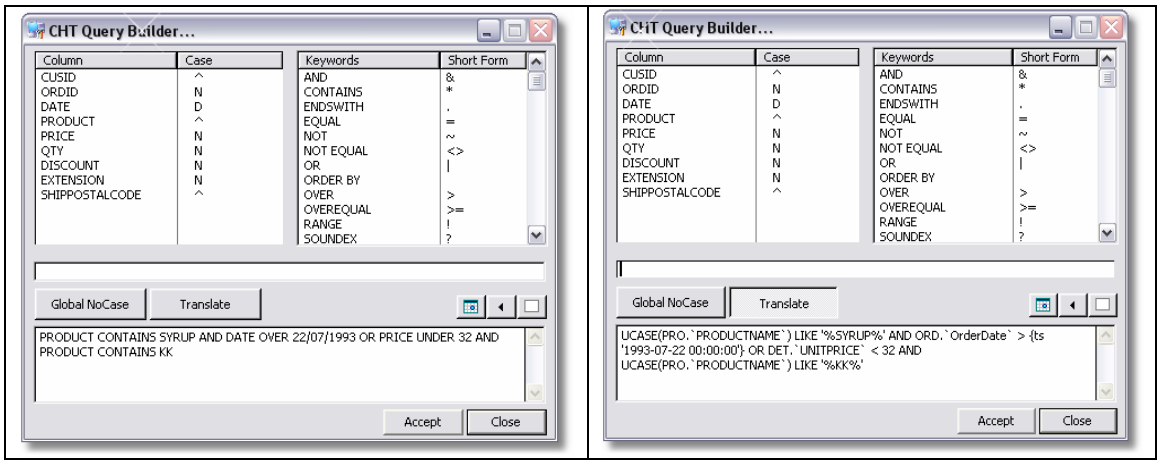
1. It creates CHT query language, which users can understand.
2. It produces top-notch, efficient SQL which your data base understands.

The end result is efficient WHERE clauses, faster-running applications and a lot less network (intranet or internet) traffic.

The New CHT Query Builder Tool

Remember how this all hangs together. The Clarion Handy Tools know where and how to apply View {PROP:SQLFilter} for maximum efficiency in an ABC environment. They know how to generate optimum SQL WHERE clauses starting from a user-friendly query language that our templates allow you to set up in any native language. And now, in build (O8A1.x) we've added a built-in easy-to-use tool that lets you or your end users design CHT-style queries via a point and click interface. Like CHT query language, the CHT query builder tool is a marvel of simplicity compared to most query wizard tools. It helps your users build real-language queries that *they* can understand and translates them into something your back end data bases (ISAM or SQL) can understand.

Here's an example CHT query applied on the SQL ExplorerBrowse example procedure in the demonstration application:



```
PRODUCT CONTAINS SYRUP AND DATE OVER 22/07/1993 OR PRICE UNDER 32 AND PRODUCT
CONTAINS KK
```

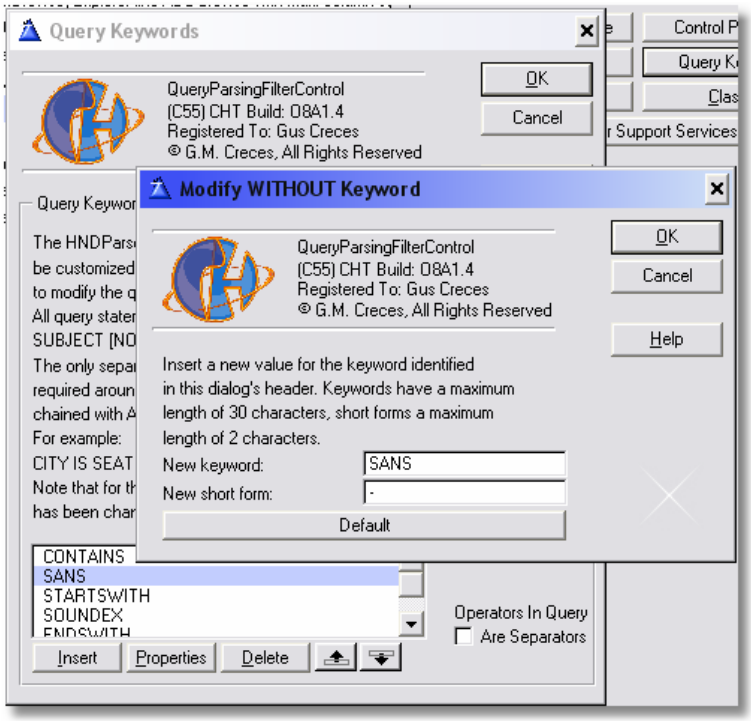
Translated to SQL for SQL back ends:

```
( UCASE(PRO.`PRODUCTNAME`) LIKE '%SYRUP%' AND ORD.`ORDERDATE` > {ts '1993-07-22
00:00:00'} OR DET.`UNITPRICE` < 32 AND UCASE(PRO.`PRODUCTNAME`) LIKE '%KK%')
```

Translated to Clarion Query Language for ISAM back ends:

```
MATCH(PRODUCT, '*SYRUP*',17) AND DATE > 70333 OR PRICE < 32 AND MATCH(PRODUCT,
'*KK*',17)
```

Normal behavior for the query tool is to let users build the query in the form of CHT query language - remember this can be made to resemble the end-user's own language using the CHT template interface called "Query Keywords".



This CHT query is passed to the browse class variable HBrwx.Query and is then translated to SQL or Clarion query statements with a call to HBrwx.PostQuery(). This single, automatic call made for you by the CHT classes, translates and posts the query to the back end and refreshes your browse. The resulting back end filter can be found in HBrwx.Filter. Given the CHT query shown above, that variable would contain one of the above translations depending on the state of the HBrwx.GetUseSQL() setting, a property of the CHT browse extension classes that can be set on the template or can be set at run-time with a call to HBrwx.SetUseSQL(True).

For example, template code like the following is triggered when the question mark button above the "CHT Query" control is clicked in the SQLFilter_ExplorerBrowse procedure.

```
!---------------------------------- Start: HNDTOOLS.TPL 08A1.4 © 2004 (QueryParsingFilterControl)
BrFilter2 = SELF.QBInterface(SELF.OQ)
IF NOT SELF.QBInterfaceGetCancelled() THEN
   SELF.PostQuery(EVENT:ScrollTop,True)
   DISPLAY(?QueryControl:2)
   !BEGIN "C) The Clarion Handy Tools - Embed QBInterface Accepted (Not Cancelled)."
   ! [Priority 5000]

   !FINISH"C) The Clarion Handy Tools - Embed QBInterface Accepted (Not Cancelled)."
END
!---------------------------------- End:   HNDTOOLS.TPL 08A1.4 © 2004 (QueryParsingFilterControl)
! [Priority 1300]
```

This code does the following:

1) It launches the query builder interface which returns a query (or nothing if cancelled) to a local, template-created variable, (i.e. the USE variable of the query control). This is the variable that's populated inside any CHT query control.

   ```
   BrwFilter2 = SELF.QBInterface(SELF.OQ)
   ```

2) This call asks you to create a query or cancel. Since SELF.Query, the filter class variable that transports the query is referenced by the template to the query control's USE variable, (in this case BrwFitler2) typing or selecting a query into the query control is the same as setting SELF.Query to the CHT filter that you're typing or selecting.

3) If the query is accepted as confirmed by SELF.QBInterfaceGetCancelled() the code issues SELF.PostQuery(). This among other things, validates the query, and posts an "Invalid Query" message if the query is syntactically incorrect. It also does a translation of the CHT query language version of your query to the requested back end value as determined by the filter class flag returned by SELF.GetUseSQL(). The translation is stored in SELF.Filter after the SELF.PostQuery() call. If SELF.GetUseSQL() is False, the CHT query is translated into a standard Clarion language filter like `MATCH(PRODUCT, '*SYRUP*',17)` for `PRODUCT CONTAINS SYRUP`. And if SELF.GetUseSQL() is True, then the CHT query is translated into an SQL language filter like `UCASE(PRO.`PRODUCTNAME`) LIKE '%SYRUP%'` for `PRODUCT CONTAINS SYRUP`.

Remember, the value of SELF.Query is not changed. It still appears in the query control as CHT Query Language. But a translated version of SELF.Query is now available in SELF.Filter. This SELF.Filter value is transferred to ABC.SetFilter() if SELF.GetUseSQL() is False. The value of SELF.Filter is passed to View{PROP:SQLFilter} if SELF.GetUseSQL() is True.

If you want to post a query in a CHT browse without the query control present you can still do so as follows, for example:

```
HBrwx.Query =  'COMPANY CONTAINS HORN'
HBrwx.PostQuery(EVENT:ScrollTop, True)
```

Or using the query builder interface:

```
HBrwx.Query = HBrwx.QBInterface(HBrwx.OQ)
IF NOT HBrwx.QBInterfaceGetCancelled() THEN
   HBrwx.PostQuery(EVENT:ScrollTop, True)
END
```

That's essentially what we've done by embedding under the *Inject Customer Query* button again on the SQLFilter_ExplorerBrowse procedure. The call to HBrwx.PostQuery() translates and resets the browse so that the query is executed. In that *Inject Customer Query* code, we're clearing HBrwx.Query right after the call to HBrwx.PostQuery() so that the next time the browse is reset - like when you click another header - the browse refills completely without the query value in place. If you provide HBrwx.PostQuery() a second parameter of True and include an Order By Clause in your query, browse order is reset to the column indicated in the Order By clause.

Here is the actual code embedded under the *Inject Customer Query* button in: SQLFilter_ExplorerBrowse().

```
!This calls the new in O8A1.0 query builder interface.
!This filter will clear as soon as you change browse order to another column.
HBrw5.Query = HBrw5.QBInterface(Hbrw5.OQ)
HBrw5.PostQuery(EVENT:ScrollTop, True) !Accept Order By clauses.
IF NOT HBrw5.QBInterfaceGetCancelled() AND HBrw5.Filter THEN
   !If the query resulted in an SQL filter.
   !This code is just to satisfy the code under ?ProcessButtonClarion
   !It saves the query for later use.
   HBrw5.Locator.Alt = HBrw5.Query
   !Query not retained after query posted and translated to SELF.Filter.
   !Hence the next browse reset clears the query.
   CLEAR(HBrw5.Query)
END
```

Note that we're saving the current query in HBrw5.Locator.Alt so that when you press the next button *CHT Query To Process* the query can be recovered and passed to a process. The code embedded to do that is:

```
IF NOT INSTRING('ORDER BY', HBrw5.Locator.Alt, 1,1 ) THEN
   !A CHT Query Language Query is able to pass through an ORDER BY statement.
   !Add it in if there isn't one already there.
    ProcessQuery = CLIP(HBrw5.Locator.Alt) & ' ORDER BY ' & |
    HBrw5.GetOrderColumnHeader()
 ELSE
    ProcessQuery = CLIP(HBrw5.Locator.Alt)
END
```

Here we're recovering the query stored in HBrw5.Locator.Alt and adding an ORDER BY clause (if needed) using the HBrwx.GetOrderColumnHeader(). When your query is in CHT Query language format, you can add an ORDER BY clause and the PostQuery() function will translate that correctly for the back end. The process procedure CHTQueryToHTMLCustomerProcess() is called with a parameter variable containing the CHT query language version of our query. We'll look closer at this process in a second HNDACCES.APP tutorial.

Programmer-Written SQL Filters

There will come a time in any design when you may need to extend query depth beyond what the CHT query control, CHT join setups and the Clarion driver-supplied joins, discussed above, can supply. And if you're fluent with SQL you may want to apply these additional, or replacement, WHERE components as SQL statements directly to the back end. To do that, the CHT parser classes, which are available whenever a CHT browse extension is applied to your browse, provide two direct-to-back-end filtering methods.

These are:

1. HBrwx.AppendSQLFilter('Your filter here') or HBrwx.AppendSQLFilter(YourFilterVar)
2. HBrwx.ReplaceSQLFilter('Your filter here') or HBrwx.AppendSQLFilter(YourFilterVar)

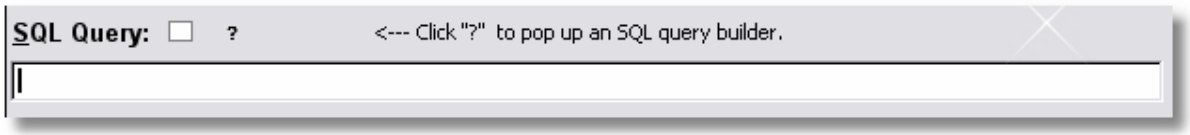
AppendSQLFilter does exactly as the name implies, it *appends* another condition to the WHERE clause using **AND**. ReplaceSQLFilter follows its namesake also and replaces all existing filters on the browse (except Clarion driver-inserted JOIN expressions) with the SQL statement that you provide it.

If you open any of the browse procedures in HNDACCES.APP, say, **SQLFilter_ExplorerBrowse** in the Clarion Embeditor, and search on DEVELOPER NOTE: you will encounter the following embed comment placed by the CHT browse extension template:

```
BRW1.ApplyFilter PROCEDURE

!BEGIN "Browser Method Data Section"
! [Priority 5000]

!FINISH"Browser Method Data Section"
  CODE
  !BEGIN "Browser Method Code Section"
  ! [Priority 2500]

  !---------------------------------- Start: HNDTOOLS.TPL 08A1.4 © 2004 (LocatorOverRideControl)
  !DEVELOPER NOTE:
  !Use HBrw5.AppendSQLFilter('Your filter here')  in the next embed to
  !append extra SQL filters using 'AND'
  !Use HBrw5.ReplaceSQLFilter('Your filter here') in the next embed to
  !replace all SQL filters with an SQL filter of your own.
  !You have chosen the Disable ABC.SetFilter() option. This option provides
  !greater control of the SQL statements sent to the back end since it stops
  !PARENT.ApplyFilter() from firing and multiple assignments to {PROP:SQLFilter}.
  HBrw5.ApplySQLFilter()
  ! [Priority 4950]

  RETURN
  !---------------------------------- End:   HNDTOOLS.TPL 08A1.4 © 2004 (LocatorOverRideControl)
```

Embedding a hard-coded constant filter is really not very flexible so we suggest you embed one or more HBrwx.AppendSQLFilter(YourFilterVar) statements instead. To capitalize on this, all you need to do is insert a legal SQL statement into your variable, **YourFilterVar** and reset the browse. *Where,* in the flow of your procedure, you change the value of this variable is very flexible and pretty much up to you. The variable could be populated right on the browse for the user to insert an SQL statement, or it could be populated in a dropdown selection list or a radio control. How ever you decide to insert SQL into your variable, once the statement changes, simply refresh the browse using **ThisWindow.Reset(True)** or **HBrwx.HardResetBrwQueue()**. Which of these reset methods you use depends on how much resetting needs to be done. The first resets the entire window including any other browses found there. The second only resets the browse addressed by the **HBrwx** instance.

We have an example of SQL-direct-to-back-end using AppendSQLFilter() in the HNDACCES.APP browse called **SQLFilter_ExplorerBrowse**. Note that on its window is a query control that looks like this:



We want to be able to enter real SQL queries on this control instead of CHT Query Language. To do that we've embedded a call to the query builder under the question mark (?) button and a call to refresh the browse whenever the query control receives an EVENT:Accepted.

In the BRWx.ApplyFilter() embed point we've embedded an HBrwx.AppendSQLFilter() statement as outlined above. So an SQL filter statement can be created with the query builder, or an SQL filter can be entered directly, by typing into the query control. Either of these actions causes the browse to be reset and the SQL filter to be applied. Since we've used HBrwx.AppendSQLFilter(), the filter we enter will not remove or disturb any of the existing filters we have in place on the browse. In other words, you are able to drill further down into the existing data set if that's what you want to do.

Under the question mark button a call to HBrwx.QBInterface() is hand-embedded with the second parameter set to True. This causes the query builder to translate all statements direct to SQL in the query builder instead of presenting and returning them as CHT Query Language. When the query builder returns, the browse is reset with Hbrwx.HardResetBrwQueue().

```
OF ?SQLButtonQueryHelp
  !BEGIN "Control Event Handling"
  ! [Priority 4499]
   !Ask for a query already translated to SQL.
   !Get the filter that QBInterface returns (if any) and reset. (Causes ABC.ApplyFilter() to fire)
   SQLFilter = HBrw8.QBInterface(HBrw8.OQ, True)
   HBrw8.HardResetBrwQueue()
```

Under the query control itself, we've embedded only the code to reset the browse since the query has been typed right into our query variable SQLFilter. The downside of allowing direct entry is that the SQL being entered is not pre-qualified and guaranteed to be accurate as it is by the using the query builder above. One way or another, the back end will tell you soon enough if your SQL statement is bogus.

```
OF ?SQLFilter
  !BEGIN "Control Event Handling"
  ! [Priority 4499]
   !(Causes ABC.ApplyFilter() to fire)
   HBrw8.HardResetBrwQueue()
  ! Generated Code
```

In the DEVELOPER NOTE: area of the ABC.ApplyFilter() method we've hand embedded a call to HBrwx.AppendSQLFilter(SQLFilter) to add our SQL filter statements without disturbing any other WHERE conditions already in place on the browse. If our filter is blank, it isn't applied so that a browse reset clears that portion of the WHERE statement that comes from our SQLFilter variable.

```
BRW7.ApplyFilter PROCEDURE

!BEGIN "Browser Method Data Section"
! [Priority 5000]

!FINISH"Browser Method Data Section"
  CODE
  !BEGIN "Browser Method Code Section"
  ! [Priority 2500]

  !---------------------------------- Start: HNDTOOLS.TPL 08A1.4 © 2004 (ExplorerBrowse)
  !DEVELOPER NOTE:
  !Use HBrw8.AppendSQLFilter('Your filter here')  in the next embed to append
  !extra SQL filters using 'AND'
  !Use HBrw8.ReplaceSQLFilter('Your filter here') in the next embed to replace
  !all SQL filters with an SQL filter of your own.
  !You have chosen the Disable ABC.SetFilter() option. This option provides
  !greater control of the SQL statements sent to the back end since it stops
  !PARENT.ApplyFilter() from firing and multiple assignments to {PROP:SQLFilter}.
  HBrw8.ApplySQLFilter()
  ! [Priority 4950]
  !Applies the SQLFilter value to any other filters applied by other means.
  IF SQLFilter THEN
     HBrw8.AppendSQLFilter(SQLFilter)
  END
  RETURN
  !---------------------------------- End:   HNDTOOLS.TPL 08A1.4 © 2004 (ExplorerBrowse)
```

Remember that with each browse reset, a fresh, full select statement including SELECT, FROM, WHERE and ORDER BY is sent to the back end. The WHERE clause is re-built from scratch, with each new reset applying the following information:

1. Any INNER/OUTER JOIN conditions described in the browse table schematic are added in from scratch by the Clarion file driver.
2. Any JOIN conditions described on the CHT "Join" dialog are added from scratch by CHT classes.
3. Any ABC.SetFilter() conditions are added in from scratch by the ABC Classes. *We suggest you disable ABC.SetFilter() as explained in this essay since this is prone to producing bad SQL due to the driver not always being able to translate complex Clarion filter statements to SQL.*
4. Any CHT Query control, CHT Query Language, conditions are automatically translated to optimal SQL and are added in from scratch.
5. Any HBrwx.AppendSQLFilter() conditions applied by your embeds are added in from scratch. A blank filter is not appended and therefore has no effect.

About ABC Page-Loaded and File-Loaded Browses

This is a good time to discuss, however briefly, another aspect of Clarion browses that Clarion users frequently misunderstand. An ABC browse has two fundamental settings that can have a huge impact on browse speed. These are the *page-loaded* and *file-loaded* settings. When to use which one is a critical piece of knowledge that can impact the apparent speed of an SQL application to an enormous extent.

Remember my earlier assertion that SQL data bases work by request and response. A request goes out to a data base table for a set of data records and a response - a subset of data meeting the request - comes back. This request/response model implies something you may not have given much thought to, but which *should* be kept foremost in your mind when designing any SQL application. The corollary implication of any SQL request is that the request is *for a subset of table records not the entire data table.*

Unless your ABC browse has a filter in place to limit the records requested when the browse initially opens, you're asking the data table for all the records *before* the browse user has been given a chance to indicate which records he/she actually wants to see. This is a hold-over from Clarion's ISAM days. ISAM data bases are filtered at the level of the client application, not the data base. That being the case, the implication was that there is no penalty for initially displaying (or at least appearing to display) the entire data base when the browse opens. Anyone who has used large TPS files across a LAN knows this assumption is a costly one in terms of performance and speed.

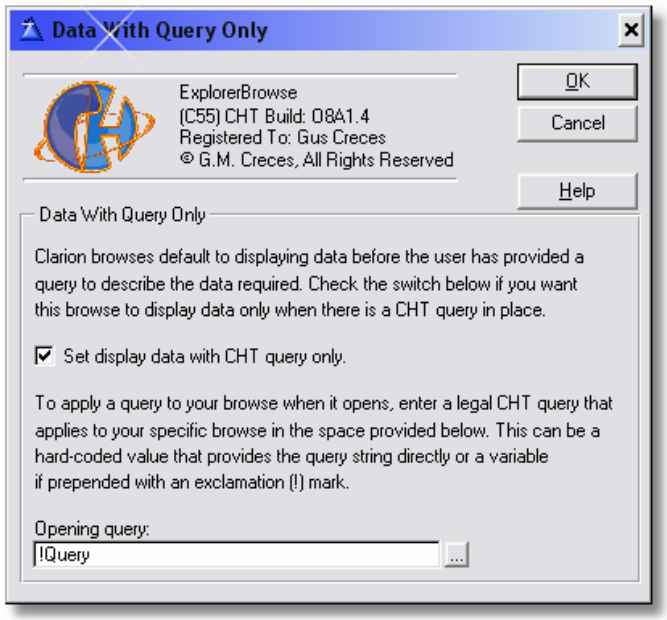
So the question remains, how really useful is this display-before-query behavior given what you must pay for it in terms of network file traffic and speed?

Page loaded browses, thus were introduced to chop the data requests up into bite-sized pieces. When an unfiltered, page-loaded browse initially opens it does not fetch all 100,000 records in the data table. It fetches the first 15-20 of them, the number depending on how may rows your browse is designed to display. This gives the illusion of speed. "Wow I loaded a 100,000 record Firebird data table in 2 seconds flat!" *Get a grip Kelvin*. You didn't load 100,000 records. You loaded a couple of dozen. Thankfully the remaining 99,976 records didn't have to trundle across the network cable to your PC only to be cast aside when you finally decide that you want to see them back-to-front or you only want to see orders pertaining to a specific city. Because it provides an illusion of speed, page-loading fosters the corollary illusion of efficiency. While efficiency *is most definitely gained*, because the entire result set of that initial non-query didn't travel across

the network, that doesn't mean that there was no impact on the data base back end which has been forced to set aside resources to accommodate what is in essence an *incomplete, bogus query encompassing the entire table* that must be handled until such time it is acted on or is simply cancelled and the resources and cursors released.

Given what I've said above, a page-loaded browse is not the most efficient browse model after all. A *filtered* file-loaded browse is *far more efficient* if you make the assumption that the filtered browse is fetching records that someone actually wants to see. At least in that case, the records that were moved across the network and the resources that were set aside by the SQL data base have a valid reason for being. A *filtered* page-loaded browse is perhaps still more efficient if you're not just "fishing" for data and paging aimlessly through the browse all the while causing hits on the back end. Page-loaded browses can impose numerous file hits for the same data set displayed in the same order on the same query. File-loaded browses don't need to go back to the data base under those conditions.

In most situations, a browse with no query is a waste of network and data base resources. That being the case, we've provided a new browse dialog in the O8A1 build that lets you create Clarion browses that do not display data unless there's a query in place. As soon as the browse query is cleared, the ABC browse mechanism no longer sends a WHERE-less select to the data base requesting all table records. The browse simply sits empty, waiting for a new query. With this feature enabled, on the "Data With Query Only" browse dialog, the only thing that can make your browse inefficient is incompetent queries that ask for more data than is actually required.

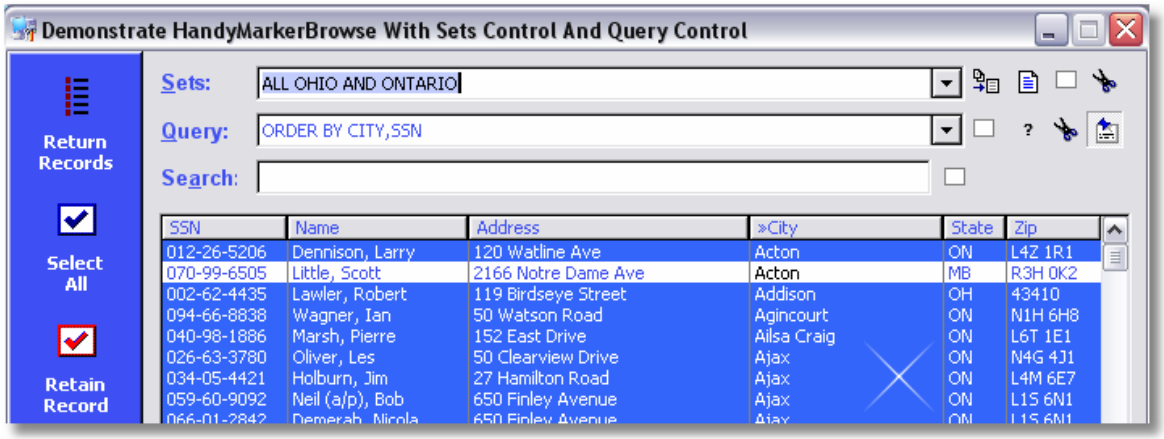


By way of a reminder, *ExplorerBrowse* and *LocatorOverRideControl* browses may be either page-loaded or file-loaded. The decision is entirely yours. *HandyMarkerBrowses,* on the other hand, are file-loaded only, *and for good reasons which we'll explain now!*

More than a few times, CHT users have asked me, "Why don't you introduce a page-loaded marker browse? I have a 100,000 record data base that I want to load and have users select and mark records in. File loading it is so slow!" Think about that question for a minute and realize what you're implying that the users of your application are going to do. *They are going to be asked to flip through 100,000 record data base to find and mark a few records to be sent to a print or process procedure.* "No", you vociferously proclaim, "I'm going to add a query control to the

browse. My users are going to narrow the number of records to a small subset of the file, and mark a few records within that subset." Well, now you're talking sense! In actual fact, the only real reason you asked for a page-loaded marker browse was to eliminate the initial load period while all 100,000 records loaded. Now you have a way to *reduce initial browse load time to zero seconds*. Create a *HandyMarkerBrowse* that loads no records until a query is entered. There'll be no more waiting or wasted network and data base resources and no more multiple file hits as the user pages through his data set, because all the records they've asked to examine are local.

If you do have large numbers of dissimilar records to mark, populate also on your browse a corollary template to HandyMarkerBrowse called **HandyMarkerBrowseSetsControl**. This lets your users accumulate sets of marked records by name, which can be recalled at will. There's an example of this in another CHT demo application called HNDLBXDM.APP in the procedure called HandyMarkerBrowseSelectorWithSets. Here's an illustration of that in action:



This essay has been expanded from its earlier version to include more information about browse-related issues that keep coming up on the CHT Web Group support server in the form of developer questions. The earlier essay also discussed issues relating to reports and processes and how CHT classes can increase the productivity and speed of ABC report and process templates. Those portions of this document have been moved to a second HNDACCES.APP essay which will be available from our download site or from the Application Tutorial menu in HNDACCES.APP.

Have fun!

Cheers...
Gus M. Creces
The Clarion Handy Tools Page
support@cwhandy.com