vb123.com

Smart Access

Solutions for Microsoft® Access® Developers

Making History

Doug Den Hoed

Doug Den Hoed is back with another new technology invention. This time, he introduces his technique to capture multiple on-demand "history time slices" of an application, allowing you to instantly flip back in time and view data from "back then." Doug exploits it all with some interesting tricks to see trends over time.

MOST of the database applications I build are transaction-based: Data comes in, is duly noted, and then is summarized and reported later. But one of the applications that I'm working on right now is different: It's a budgeting system for an oil exploration company. In the fall, my users distribute the wells that they're most likely to drill across several budgets in different geographical areas. In the winter, the ground freezes, allowing the heavy trucks to get to the target locations and start drilling. Every day, the drillers either hit or miss oil; either way, it gives the geologists better information about "what's down there, and how much there is, if any." So each week, the users meet, weigh the information, and *reassign* which projects will be drilled, constantly trying to increase the chance of hitting oil or gas—to maximize their return.

From a database perspective, this means that the wells assigned to the budgets are constantly changing over time. To gauge how well their decisions are working out, my users asked me for the ability to freeze the budgets at a point in time and, at a later date, let them compare against those previous budgets later. To them, it didn't seem hard: It was something they used to do with a couple of cut-and-pastes in the application's Excel-based predecessor. For me... it wasn't so easy.

I considered copying the data (an mdb file) every week, linking to it, and running reports against those linked databases. This would mean that

Continues on page 4

April 2002

Volume 10, Number 4

1 Making History *Doug Den Hoed*

3 Editorial: Toys, Changes, and Skills *Peter Vogel*

10 Stump the Expert: Resize-Recalc *Doug Den Hoed*

11 An Access Data Project: Designing the App *Martin Reid*

15 Reviewer's Corner: SPEED Ferret 4.0 *Danny J. Lesandrini*

18 Reviewer's Corner: FMS Total Access Speller 2002 *Danny J. Lesandrini*

21 Access Answers: AutoLookup and Output in Many Formats *Peter Vogel*

24 April 2002 Source Code

Applies to Access 95 | Applies to Access 97 | Applies to Access 2000 | Applies to Access 2002

www.vb123.com/smart

In code, an underscore (_) as the last character of a line indicates that the line has been wrapped for layout purposes. In Access 95 and up you can use the code as it appears, but in Access 2.0 you must recombine the wrapped lines.

Making History...

Continued from page 1

I'd have to duplicate (and maintain) the multi-layered queries behind some of my most complex reports so they'd also draw from the linked data. And that was only the start of the problems. What if the data structure changed over time, as it always does? Would I have to save the front end that "worked" at that time in case users wanted to go back to view something in the copy? Or would I retrofit new features into that copy? How could I prevent tampering? Once the data got big, how could I easily convert it all to Oracle or SQL Server? And once it was stored there, how could I then make a copy, since it's not just an mdb file anymore?

I found a better solution to my problem. On demand, my users can make a copy of the entire database "as it stands." At any time, they can instantly flip from the current data (editable) to any historic time slice (read-only) to view the data "back then." And, in certain forms and reports, they can view the variance between the current data and the historic time slice they've chosen.

My solution is also elegant for developers. I avoided duplicating any object; every query, form, and report that's used to show the current information can also show historic information. I minimized the impact on performance by keeping the current data separate from the history, and by structuring the historic information in a clustered, compact fashion to reduce data retrieval time. My solution will also scale to SQL Server and Oracle.

Finally, my solution is generic enough to add to any application. I intend to consider incorporating it into every future project. I even expect to retrofit this functionality to some of my existing applications. So you can use this solution too.

Take a gamble

In this month's Download file (available at www.smartaccessnewsletter.com), you'll find a

Figure 1. Sample data model with historic extensions.

sample application that that I wrote to explain my History technique. Figure 1 shows the data model of the sample application (PoolForm.mdb). It's a hockey pool. If nothing else, you can run your office gambling from the source code.

At the beginning of the pool, I created teams for my office cohorts (tblTeam), entered the players they picked (tblPlayer), and drafted them onto their respective teams (tblTeamPlayerXref). Each team can have only five players, although a player can be used across different teams. Each team can trade one player every two weeks to improve their point total. To run the game, I went to the www.nhl.com statistics page each week, updated the top players' points (in tblPlayer), and "Made History" (which I'll explain shortly). The highest team score on the first of each month and at the end of the game won prizes.

PoolData.mdb holds all the current data. The tables directly underneath (thstPlayer, thstTeamPlayerXref, and thstTeam) hold the historic data in the database PoolHist.mdb. To build the initial version of the historic tables, I copied PoolData.mdb and saved it as PoolHistory.mdb. In PoolHistory.mdb, I renamed each table with a "thst" prefix instead of the usual "tbl" and then cleared out all the data. In each "thst" table, I added a long integer field called HistoryID at the front of the primary key and converted the existing autonumber fields to long integers. I then created a new log table to track additions to the history table.

Putting the HistoryID at the front of the primary key gives me the best performance because most queries use a HistoryID when referring to a historical table, so Access will use the primary key as an index to speed up the query. I also removed the referential integrity from between the tables to avoid any parent/child issues when inserting rows.

In the application, the Make History function (hstTakeHistory) inserts a new record into the log table (tsysHistoryLog) in PoolHist.mdb. This causes the autonumber HistoryID field in the log file to generate the next HistoryID number. The function then copies every row from the tables in the "current" database into its "historic" twin, adding the HistoricID value to make the records unique. If the inserts all work, I set the IsActive field in the log table to True, indicating that users can rely on the data's integrity.

Collectively, the data across the tables is the "historic time slice" I described earlier. Inserting to all tables sequentially keeps the data "tight" on the hard disk—or perhaps even clustered in an Oracle or SQL Server scenario—which helps performance. The final table (tfltHistory) is a local table (PoolForm.mdb) that holds the HistoryID that the user is viewing.

Figure 2 shows how I use the user's HistoryID

choice to select a specific historic time slice from among the many within a "thst" table. The join on the HistoryID restricts the data, which is the concept of "horizontal segmentation" that I first introduced in my "Temporary Tables with No Bloat" article in the January 2001 issue of *Smart Access*. The other table (DUAL) is an idea I stole from Oracle. DUAL is a handy utility table that always and only contains a single record. The table has two fields:

- A field called ID, set to 1
- A field called Dummy, set to "X"

I included DUAL in my "thst" queries without a join, technically creating a Cartesian product against the DUAL table. Since there's only one row in DUAL, the Cartesian product of joining a table to DUAL is a duplicate of the table... but in read-only mode, which is why I bothered. This technique protects the historic data from inadvertent or malicious tampering. Leave DUAL out uf your queries if you need to rewrite history in your applications.

Having explained the data structures, I can now introduce the Hockey Pool sample application. Figure 3 shows the main form (fnavMain). The combo box at the top (cboHistoryID) is bound to tfltHistory.HistoryID, which governs the "qhst" queries. It also uses the DUAL table in its row source:

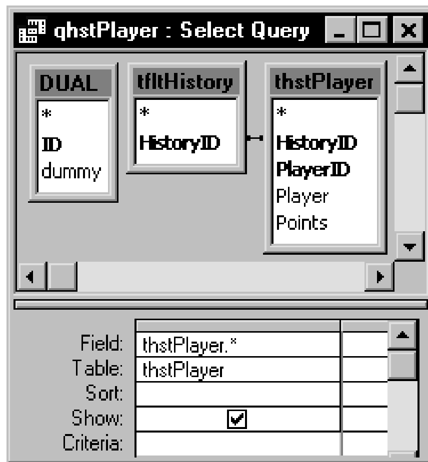

Figure 2. Sample "qhst" query.

```
SELECT 999999999,"<Current Information>"
  FROM DUAL
 UNION SELECT tsysHistoryLog.HistoryID,
  [HistoryDate] & " -- " & [Comment]
    AS LogicalHistoryDate
  FROM tsysHistoryLog
 WHERE tsysHistoryLog.IsActive=True
 ORDER BY 1 DESC;
```

This query creates a fake row in cboHistoryID, a row that doesn't actually exist in tsysHistoryLog. This default rule is the default value for the control. The Order By clause places the default at the top of the list, followed by every valid historic time slice from most recent to oldest (I'll explain later what happens when you pick a real HistoryID).

The Player tab in Figure 3 is bound to tblPlayer and is editable when viewing the Current Information, as you can see from the new row at the bottom of the datasheet.

The Team tab shown in Figure 4 maintains the tblTeam on the left, and tblTeamPlayerXref on the right. There are two chunks of code worth mentioning behind this form. The first set of code keeps the Players datasheet synchronized with the selected Team. In the OnCurrent event of the Team subform, I call a routine of mine called cbfTeamRefresh, which is a public method in fnavMain. This routine pulls the current TeamID into the hidden txtTeamID field on the main form, independent of any tab, and then refreshes the dependant tab, which refers to txtTeamID in its Link Master Field:

```
Me.txtTeamID = Form.Controls("sfrmTeam") _
  .Form.Controls("txtTeamID")
Form.Controls("sfrmTeamPlayerXref").Requery
```

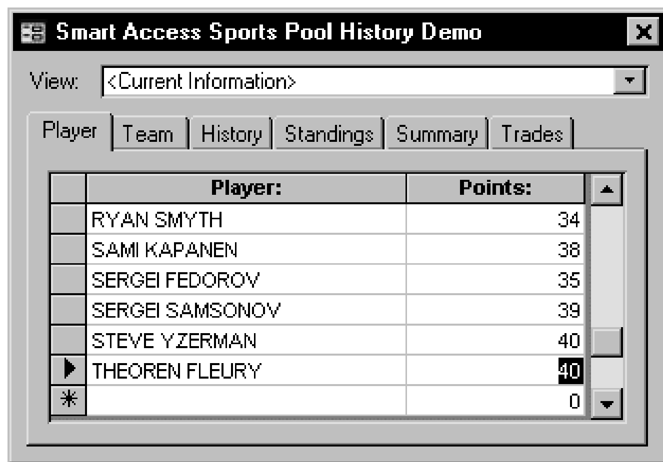

Figure 3. Hockey pool sample application.

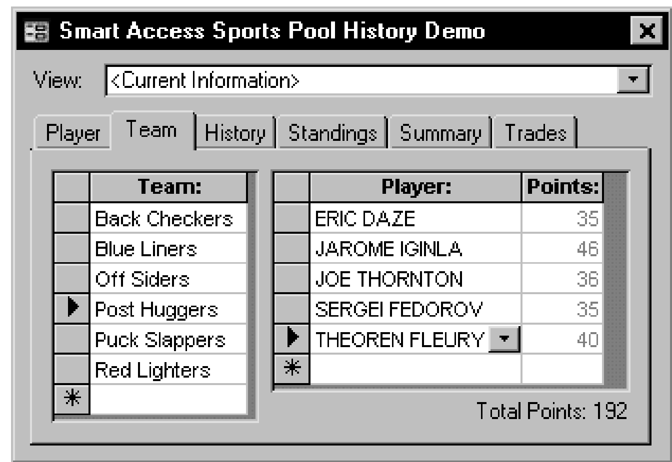

Figure 4. Team tab.

The other trick is in the BeforeInsert of the Player subform, where I limit the number of players on each team to five. It's trivial code, but important if your rules differ from mine.

```
If Me.RecordsetClone.RecordCount >= 5 Then
    MsgBox "Sorry: Maximum of 5 Players." _
      , vbInformation + vbOKOnly, "Denied"
    Cancel = True
End If
```

Show time

Figure 5 illustrates the all-important History tab. The Make History button (cmdHistory) first warns the users that they're about to take a copy of the entire database then gathers a reason/description from the users, and, finally, calls the hstTakeHistory function. After the hstTakeHistory code runs, the cmdHistory_Click routine continues by requerying the datasheet at the bottom of the form. That datasheet is bound to the table tblHistoryLog to display the information already entered and allow the users to enter additional comments.

The Delete History button (cmdDeleteHistory) is similar, but deletes the historic time slice selected in the datasheet and sets the IsValid in the log table to False. With IsValid set to False, that historic time slice disappears from the cboHistoryID combo box at the top of the form. Just to be sure that it's missing, I requery that control in its OnEnter event.

Rather than review the hstLibrary functions in detail, I've summarized them in Table 1.

So many things are possible once you start tracking historic time slices! Figure 6 shows the bar chart I created to illustrate the team points over time. The query behind the chart (qselHistoryTeamPoints) joins

Table 1. The hstLibrary functions.

hstLibrary function	Comments
hstBuildHistoryQueries A developer utility to build or refresh all of the "qhst" queries.	After you link your Current tbl* and Historic "thst" tables, run this function in the immediate window. It will walk each tbl* and create (and replace) a "qhst" select query similar to Figure 2. Whenever you make a data model change to a tbl*, make the same change to its "thst" counterpart (using a reasonable default value for the existing historic data), and then RERUN this function. If you forget, the other functions may fail.
hstTakeHistory Takes the contents of all linked tbl* tables and inserts the data into their linked "thst" table partners, creating a "History" copy as of that time.	This assumes all the tables are linked with passwords, and that the source tables are prefixed with "tbl" and the target tables are prefixed with "thst." It assumes that Referential Integrity between the target tables has been dropped, since the tables are simply loaded in alphabetical order, and that all indices (especially alternate unique keys) are dropped from the "thst" tables. It also assumes that every "thst" table has the special field "HistoryID" added to its primary key (at the *front* for best performance), but that otherwise, the table definitions are compatible (same column names, compatible datatypes). Note the feedback in the status bar as it runs.
hstEraseHistory(lngHistoryID) Deletes the entries in all "thst" tables for a given HistoryID.	This calls hstEraseHistory to undo mistakes (for example, missing a Player's point), to recover space (for example, if you compact PoolHistory.mdb), and to simplify the cboHistoryID options (for example, keeping only the official Friday stats).
hstSwapHistory(booShowCurrent, strTestTable) Swaps Current and History data.	Depending on booShowCurrent, this swaps the "tbl*" (current data) tables with the "qhst" (historical time slice) queries, effectively changing the source of the app's data. It's definitely worth stepping through.

Figure 5. History tab.

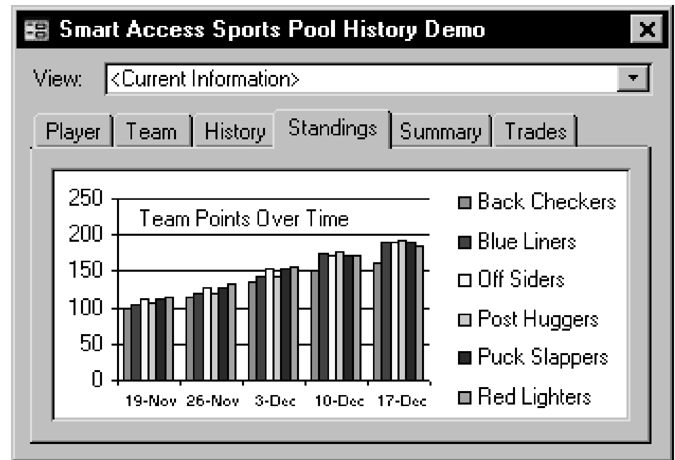

Figure 6. Standings tab.

the "thst" tables as usual, but also ties each table's HistoryID back to tsysHistoryLog to group by the HistoryDate and, within dates, by Team. For each group, the query sums the total points by team.

There's a subtle point worth mentioning here. Remember that I decided that each team could trade one player every two weeks to improve their point total? Those trades are also captured in the history. So the bar graph is actually driving the point totals off a *changing* list of players. Before I invented the hstLibrary, I would have used a traditional "effective date" data model for this application. The typical table in that kind of design is a tblPlayerPoint child table with a primary key of PlayerID and an EffectiveDate as the primary key. The data element would have been the Points I was tracking over time. This would have captured the Player Points as they change, but it would miss the data involved in trading players. Certainly, I could further modify the application and also base the Team/Player relationship on an effective date. But my point is this: By capturing *all* the data in *every* table at *regular* intervals, you leave the door open to do things you never dreamed about until later in the project.

The quantum leap

The most powerful and important function in the whole application is hstSwapHistory, which I call after a user updates cboHistoryID at the top of fnavMain. The code is surprisingly simple, and it runs almost instantly.

When the users chooses a HistoryID, the hstSwapHistory function "masks" the tbl* tables by appending "~Current" at the end of each table's name. The code then replaces the "qhst" prefix in the "qhst" queries with "tbl." The cboHistoryID_AfterUpdate then requeries all the subforms and... it's magic! Each subform runs blindly against "tbl*whatever," not caring that it's no longer the "current data table" but, in fact, a "historical query" (like Figure 2) that pulls a historic time slice. Because the "qhst" query has the same columns and datatypes as the real table, all forms, reports, and even other queries just work—total code reuse. That was a huge win on my oil exploration project, considering the complexity of some of the budgeting report queries. I confess that I did tune some of the queries for performance, since the "qhst" queries can't be indexed the way a real table can. But in a way, that was code reuse too: Speeding the queries for historic ("qhst"-based) information from four minutes to one minute also sped the current (tbl*-based) information from one minute to 15 seconds.

When I change from a historic time slice back to current data, the hstSwapHistory routine reverses the renaming process. In the source code, I've included a function to generate the "qhst" queries for you automatically.

Even when the tbl* tables are masked, you can still query them. Figure 7 shows an example. The datasheet compares each Team's Current Points against its Points from the historic time slice and shows a variance column.

To pull the data, I first switched to a historic time slice, which masked the tbl* tables. I then wrote a query (qselTeamPointsCurrent) that drew the Current Points from the tbl*~Current masked tables, grouped by the Team, summed the Points, and included an extra placeholder column called LogicalPointsPrevious with a value of zero. Next, I wrote the query qselTeamPointsHistoric, drawing from the tbl*-masked qhst queries and joining to the table tfltHistory's

Too Much of a Good Thing

Recently, I was called back to investigate some serious performance issues on the first application on which I had implemented my history techniques. A calculation-intensive variance report comparing two historic time slices had slipped from running in 10 minutes to taking more than an hour and a half. After investigating the usual suspects (database bloat, the network, and "bad data"), I realized that it was my historic database that was to blame. As the users dutifully captured their time slices, the queries that ran against them became exponentially slower since there was more data to sift through. The historic mdb had no supporting indices other than primary keys. I worked around the issue by taking a backup of the historic mdb for archival purposes, and then deleting all but the last few time slices using the hstEraseHistory function that you'll find in the source code. After compacting, the report was back to its original performance time. So, to paraphrase Steven Wright, "You can't have everything. Where would you keep it?"

Smart Access Sports Pool History Demo

View: 01/12/03 19:22:54 -- Week 2

Player | Team | History | Standings | Summary | Trades

Team:	Current:	Old:	Change:
Post Huggers	192	143	49
Puck Slappers	191	154	37
Off Siders	190	154	36
Blue Liners	189	144	45
Red Lighters	186	155	31
Back Checkers	162	136	26

Figure 7. Summary tab.

HistoryID field. This query also grouped by Team, LogicalPointsCurrent with a value of zero, and summed Points. Keeping the number of columns, their order, and the datatypes the same was intentional because I could union these two queries together in the query qselTeamPointsUnion. Finally, my last query (qselTeamPointsSummary) did a group by Team, summing LogicalPointsCurrent and LogicalPointsPrevious to calculate the variance between the two sets of logical points in a fourth column in the query. This is the query that's actually bound to the datasheet (frmTeamPoints).

There's another dimension to this tab. The function fmtDatasheetColumnSort allows the user to double-click any column header to sort the datasheet descending by that column. I can watch the results in the Team column as I click and monitor the Current Rank, Former Rank, and Largest Change, respectively. It's a very interactive way of examining data.

I think that the Trades tab shown in Figure 8 is my favorite example of the power of capturing history with my technique. The datasheet shows the *changes* over time as a team trades one player off its roster for a new player with higher points. As usual, the tricks are in the queries.

First, the query qselTeamPlayerCurrent selects the current Team, Players. The query qselTeamPlayerHistoric does the same for the historic time slice. Next, the query qselTeamPlayerAdded left-joins qselTeamPlayerHistoric to qselTeamPlayerCurrent and selects the players that have been added (that is, exist now, but didn't in the history). Similarly, qselTeamPlayerDeleted does the reverse (that is, finds players that don't exist now, but did in the history). Then qselTeamPlayerTrades simply unions the two queries, which the form (frmTeamPlayerTrades) displays, sorting by Team and Player. With hundreds of oil wells to drill, tracking changes like this adds real value to my client's project, and saves them the bother of keeping separate error-prone lists.

The concept of capturing *all* data from *every* table at *regular* intervals is changing the way I design my applications. Imagine offering your users the ability to instantly flip between current and historic information within the same application. Imagine reusing every query, form, and report to view both sets of data. Imagine not having to retrofit objects into archived environments. Imagine saying "yes" when someone asks whether your solution will still work against Oracle or SQL Server. I'm still imagining the possibilities myself.

In the meantime, I suspect that you can implement my techniques in almost any application in less than four hours by using the objects from my sample application. So bolt my sample code into your application, and start capturing historic data now so you can use it later! Who knows what you'll think of? ▲

HISTORY.ZIP at www.smartaccessnewsletter.com

Doug Den Hoed is a founder of Lumina Systems Delivery in Calgary, Canada, which specializes in customized software solutions using Access, Visual Basic, InterDev, SQL Server, and Oracle. Doug gets a lot of his material for his articles from The KB™ (www.thekb.com), the commercial Access application he invented to manage multiple software development projects. busi_ops@anadarko.com.

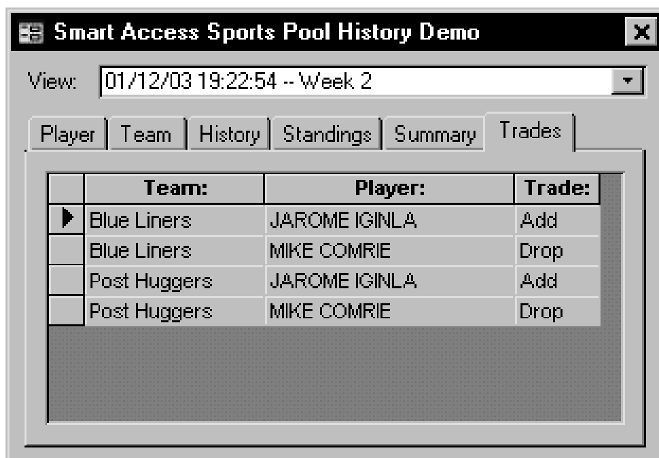

Figure 8. Trades tab.