J9.10 STATION HISTORY DATABASE ARCHITECTURAL TECHNIQUES

Jeff Arnfield*
National Climatic Data Center, Asheville, NC

Gary Shears
Sherian Corporation, Morgantown, WV

1. INTRODUCTION

As the world's preeminent climatic data archive, the National Climatic Data Center (NCDC) manages both observational data and the metadata necessary to interpret them. Chief amongst these metadata is station history – the changes to a station's identity, location, configuration and observing practices over time.

Working with contractors under the auspices of the Climate Database Modernization Program (CDMP), NCDC has developed an Oracle-based station history database to accommodate stations from a variety of climatological observing systems. The project presented a variety of challenges, some logistical, some data-related, some conceptual and some architectural (Arnfield, 2000).

The database has more than 130 tables, each representing a fact about a station, like its instrumentation or the phenomena observed, or a relationship between two facts, such as which instrument is used to observe which phenomenon. To remove dependency on external identifiers beyond our control, like Coop and WMO IDs, these tables are linked using a numerical, non-information bearing Station Key.

A system of this scope almost always presents challenges. Two key architectural hurdles were managing the sometimes disparate periods of validity found in various tables and logging the changes made to rows in each table. We will examine these two issues and their respective solutions here.

2. RESOLVING VALID DATE RANGE OVERLAPS

Each component of a station – its location, equipment, observers, reporting methods, even its identity – can change independently of other components; thus, each component may have its own unique period of validity. This temporal aspect of a station's configuration is critical to interpreting its reported observations, since observations may be affected by changes in location, instrumentation or observing protocol. In the database, the period of validity for each row in each data table is represented using a begin date column and an end date column.

There are many possible views of a station's configuration data, each drawing details from columns in different tables to produce a composite. The date ranges in one table often span or are eclipsed by those in other tables, making queries and reporting complex. Each composite row in a view will have its own period of validity, derived from the period of validity for each component in the view.

Short of a programmatic solution for each query, view and report, how does one determine the begin date and end date for each row? How does one combine the date ranges from various tables to produce a coherent row of data?

2.1 An Example

Consider a simple report showing a station's name, Coop ID, primary temperature instrument and time of temperature. The report requires a join of three tables, with each row in each table containing its own begin date and end date. However, because the name, ID, instrumentation and time of observation may change independently, none of the tables contains the correct dates for each row in the entire view. A simplified version of the three tables and the desired output, extracted for a single station, is shown in Figure 1.

FIGURE 1

Station Name

Station Key	Name	Begin Date	End Date
2000023	ANNISTON MUNICIPAL ARPT	06/01/1948	07/01/1967
2000023	ANNISTON FAA AIRPORT	07/01/1967	08/14/1995
2000023	ANNISTON FAA AP	08/14/1995	06/17/1998
2000023	ANNISTON METRO AP	06/17/1998	12/31/9999

Station ID (showing Coop IDs only)

Station Key	ID Type	ID	Begin Date	End Date
2000023	Coop number	010272	06/01/1948	08/01/1949
2000023	Coop number	014734	08/01/1949	03/01/1950
2000023	Coop number	010272	03/01/1950	12/31/9999

Station Phenomenon Observing Protocol

Station Key	Phenomenon	Instrument	Obs Time	Begin Date	End Date
2000023	Temp	MXMN	2400	01/01/0000	08/14/1998
2000023	Temp	HYGR	2400	06/17/1998	12/31/9999

Resulting Composite View

Station Key	Begin Date	End Date	Coop ID	Name	Inst.	Obs Time
2000023	06/01/1948	08/01/1949	010272	ANNISTON MUNICIPAL ARPT	MXMN	2400
2000023	08/01/1949	03/01/1950	014734	ANNISTON MUNICIPAL ARPT	MXMN	2400
2000023	03/01/1950	07/01/1967	010272	ANNISTON MUNICIPAL ARPT	MXMN	2400
2000023	07/01/1967	08/14/1995	010272	ANNISTON FAA AIRPORT	MXMN	2400
2000023	08/14/1995	06/17/1998	010272	ANNISTON FAA AP	MXMN	2400
2000023	06/17/1998	12/31/9999	010272	ANNISTON METRO AP	HYGR	2400

* *Corresponding author address:* Jeffrey D. Arnfield, National Climatic Data Center, Active Archive Branch, 151 Patton Ave, Asheville, NC 28801;
e-mail: jeff.Arnfield@noaa.gov

When two tables are joined so that values may be extracted from both, the database determines which rows to combine by comparing values in one or more columns in each table. Where the values match, data from both rows are included in the query result. If one does not carefully specify columns for comparison, though, spurious matches may occur.

In the example in Figure 1, joining on only the Station Key would produce a result set of 24 rows: all three rows for the station in the Station ID table would match all the four of station's rows in the Station Name, and each resulting row would match both of the station's rows in the Station Observing Protocol table. In order to get the correct result set of six rows, one must include a data comparison so that only contemporary records in various tables are matched. But because the dates in one table may not precisely match those in another, this is a difficult proposition, involving correlated subqueries.

Although not illustrated in this example, matters are further complicated if one table may contains no data at all for a station during a period when another table does have data. A structured query language (SQL) technique, the outer join, is useful in resolving this problem if only one table may be missing data, but not if all three may be missing data. For the outer join to work, one table would need to contain all the necessary begin/end date pairs for the resulting view. As we've already seen, though, none of the tables in the join can be counted on for the master list of begin/end dates for the final view.

2.2 Possible Solutions

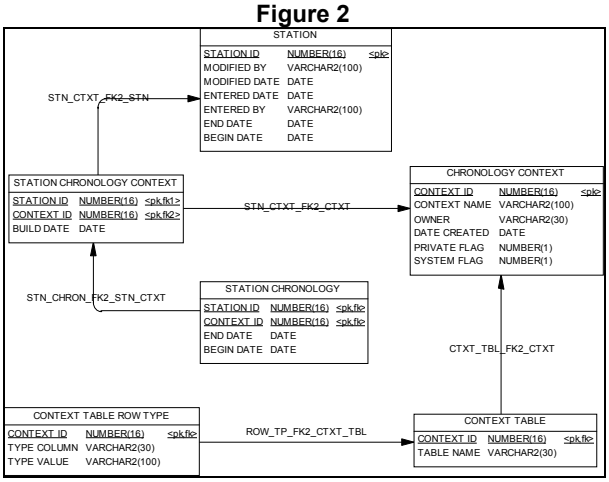
Several solutions were considered and dismissed. One possibility was to make an entry in all tables whenever data for a station changed in any table. This would result in a matching begin/end date pair in all tables. This simplifies queries, but involves massive data duplication and greatly complicates updates. It also makes it more difficult to determine when a value has changed, because a table may contain be many rows for a station, all with different begin/end dates, but all containing the same information about the station. Because historical data comes from many sources, attribution of data would become more complex as well.

A programmatic approach to combining data from various tables is possible. However, this is cumbersome, and presupposes significant programming skill and tools on the part of the user. Even skilled programmers are unlikely to be familiar with the intricacies of a database schema containing more than 130 tables. More importantly, though, it fails to solve the problem in an ad hoc SQL query environment.

2.3 The Approach Taken

What was needed was a table containing the date pairs for each station for a given view. Such a table would permit all other tables in the view to be outer joined to it. However, creating and maintaining a table for each view would be prohibitive. Instead, we designed a subsystem to address the problem for all tables in the database and all possible views of the data. Figure 2 shows the Context Chronology subsystem's schema. This schema, along with supporting programs, permits a user to define a view, specify the tables from which data will be drawn, and then automatically establish and maintain the begin/end date pairs needed to produce the view, derived from the included tables.

Figure 2



A brief discussion of the tables and their roles is in order:

- Station – this is the central table in the database; all station-related tables have an explicit foreign key relationship to it.
- Chronology Context – the table in which the context, or view, is defined. It may be used by the system, in which case the database protects it from deletion.
- Context Table – this table lists the tables contained in the context view. Our example from Figure 1 would require 3 rows (Station Name, Station ID, Station Phenomenon Observing Protocol).
- Context Table Row Type – Referring to Figure 1, we note that the Station ID table has an ID Type column. The table may contain many different IDs for a station, but if we are only interested in the Coop ID we don't want date pairs from other IDs for the station to be included in our view. This table permits us to define the column containing the "type" descriptor, and the type descriptor value to include in the view. If we also wanted to include the FAA call sign in this report, we'd have two "Row Type" rows for the Station ID table.
- Station Chronology Context – An intermediate table that links a station to a context. The build date is used by stored procedures to determine whether a context must be rebuilt for a given station.
- Station Chronology – Finally, the dates! This table contains the logical begin/end date pairs extracted from the tables composing the view. It is used to drive the query that extracts data

from the tables. Each table is outer joined to this one, so that any available data is pulled from all tables, but absence of data in any table does not cause entire composite rows to be omitted from the view.

Data that requires user intervention to remain synchronized generally succumbs to entropy, becoming inaccurate. In order to ensure data integrity and currency the update process is handled automatically by the database itself. Oracle has a database-level construct called a trigger that causes a particular piece of code to execute each time a specific event, such as inserting, deleting or updating a row, occurs. Once a context view is defined, Oracle triggers automatically invoke the stored procedures necessary to maintain these date management tables whenever rows in any station-related tables in the database are inserted, updated or deleted. This feature may be deferred in order to speed data loading, then invoked once the load is completed.

3. LOGGING CHANGES

The best-intended corrections sometimes overwrite valid data. While mistakes must be corrected, discarding original data values may be perilous. There is no audit trail unless the previous value and information source are retained when erroneous data values are corrected. To be effective, changes must always be logged regardless of how the change is made, including by direct database update outside control of the user interface.

FIGURE 3

CHANGE JOURNAL TABLE STRUCTURE

TableName – the name of the table in which the change occurred
TableKeyValue – the numeric ID for the row in which the change occurred
ColumnName – the name of column whose value was changed
OldValue – the column's original value
NewValue – the column's new value
NewValueSource – Foreign key referencing the source of the column's new value
ModifiedBy -- ID of the user making the change
ModifiedDate – date/time stamp of when the change was made.

A single Change Journal table is used to document all value changes for all tables. This approach requires a common key structure for all tables for ease of management. A numeric column provides a unique, alternate key for each table, and is populated using an Oracle sequence number specific to that table. A separate sequence for each table keeps key values, and thereby space overhead, small.

In a system that experienced frequent updates to values, the overhead in journaling each column's updates in a separate row might be prohibitive. However, in this system rows are updated only to correct errors, not to document changes in configuration over time. Changes over time are noted by inserting new rows, not by altering existing ones.

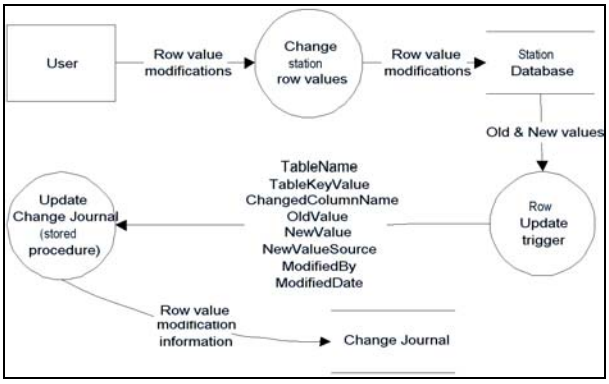
A simple SQL query will return the modification update history for a column:

```
SELECT Old_Value, New_Value, New_Value_Source,
        Modified_By, Modified_Date
FROM Change_Journal
WHERE TableName = <name of the table>
        AND TableKeyVal = <key value for the row>
        AND ColumnName = <name of the column>
ORDER BY Modified_Date
```

To ensure consistent journaling of changes, the log table is automatically maintained by stored procedures in the database rather than relying on the application developer to properly implement the code.

An "on update" trigger is created for every logged table in the database. This trigger passes certain critical values to a stored procedure, which in turn writes them to a single, common Change Journal table. The process is illustrated in Figure 4.

FIGURE 4



If we permit a row in a station detail table to be deleted and there are entries in the Change Journal table for the row, a trigger on the station detail table must handle deleting the change journal entries.

4. CONCLUSION

Both the date range handling and change journaling, while seemingly complex, have greatly simplified the development and increased both usability and data integrity of the comprehensive station history system. Performance is always a concern when implementing triggered procedures in a database. In a more transaction-intensive environment, either of these techniques might require modification, or prove unsuitable. However, the flexibility in querying station history data, combined with the ability to audit changes made by users, outweighs any performance penalties.

5. REFERENCES

Arnfield, J. D., 2000: A Flexible System To Manage And Query NOAA Station History Information, American Meteorological Society, 17th Conference on Interactive Information Processing Systems, Albuquerque, NM, Jan 14-18, 2001; p468-47