Management of Time-range Data

Andrew A. Norton
Trilogy Consulting Corporation

Conventional database management techniques identify data set observations using the values of a set of key variables. Data sets are joined by matching key values, treating these values as discrete points. For example, event data can be identified by a single datetime value because events occur at a distinct point in time.

In some data sets, key values correspond to line segments rather than points. For example, hospitalization data may be keyed by datetime-ranges of the form \([\text{DT\_START}, \text{DT\_STOP})\). Managing data keyed by ranges requires special techniques.

This paper extends point-key techniques to range-key applications. Although most of the examples in this paper use time-range keys, the techniques are equally applicable to other types of range keys, such as highway data identified by milepost ranges.

**SAMPLE APPLICATIONS**

This paper addresses all of the applications below using a single simple technique which reduces each problem to conventional point-key processing. There are undoubtedly other problems of practical interest amenable to the same approach.

**Fuzzy Merge**

Suppose there is a SAS® data set HIGHWAY, with each observation corresponding to a segment (or range) of highway. The ranges do not necessarily exist in any physical or administrative sense - they are defined by the data such that each range is homogeneous with respect to the attributes of interest. The data set is keyed by \(\{\text{HWYNAME, [LOW, HIGH]}\}\). LOW and HIGH identify the highway range by specifying the mileposts at either end. The data set includes a variable LANEs (the number of traffic lanes in that range).

A second data set, ACCIDENT, describes traffic accidents, including the name of the highway and milepost at which the accident occurred. The ACCIDENT and HIGHWAYS data sets can be linked using the variables HWYNAME and MILEPOST. To have a match, the MILEPOST value of the ACCIDENT must fall into the range \([\text{LOW, HIGH})\) of milepost values specified on the HIGHWAYS data set.

**Range Merge**

Two data sets are keyed by \(\{\text{PERSON, [START, STOP]}\}\). JOB describes the average hours worked per week during the period; SCHOOL describes the average hours attending class during the period. We would like to join the two data sets together for an analysis, even though the reporting periods do not correspond.

**Range Update**

The HIGHWAYS data set has become outdated. We obtain another data set (with the same structure) which contains update transactions, but the milepost ranges on the update data set do not correspond with those on the original data set.

**Restriction**

We wish to subset the HIGHWAYS data set and keep only that data pertaining to a specified jurisdiction, defined by a minimum and maximum milepost.

**Projection**

The SAS data set TIMEDOSE describes
experimental treatment (drug and dose) prescribed for each patient during the specified date ranges.

We are only interested in the drug information. By combining adjacent observations with the same DRUG value (ignoring DOSE differences), we can reduce the volume of data and thereby save on storage and processing costs.

The term “projection” comes from relational database management theory, where it refers to an operator which subsets columns and eliminates any resulting duplicate rows.

**Summarization**

We have a data set describing software licenses. For each license, we have a START and STOP date. We wish to determine how many licenses were in effect at any given time.

This is analogous to summarizing groups defined by point-keys, except that the ranges may overlap in various ways.

**FUZZY MERGE**

A fuzzy merge matches a point key to a range key rather than another point key. The term “fuzzy merge” appears to have been first coined by James W. Davis, Steven Flint, and Robert U. Anderson of the Institute for Applied Research Services at the University of New Mexico, in the 1982 SUGI paper, "Techniques for Interrelating and Reducing Data Bases Using SAS". This seminal paper featured many pioneering techniques and has not received the recognition it deserves.

Davis, et al. wanted to merge accident data (by milepost) with highway segment data (keyed by starting milepost). They used the example (page 422):

```sas
if INHWY then LLANES=LANES;
else output;
run;
```

Observations from the data set HIGHWAYS are read whenever the milepost range changes. The LANES value is retained in the LLANES variable during the processing of each ACCIDENT observation in the range.

**Other Approaches**

The term "fuzzy merge" lives on, although this elegant technique has largely been forgotten.

Some implementations of fuzzy merges simply generate individual observations for each date in the range, then use a conventional point-key MERGE. This works fine when there are only a few dates, but collapses when the range contains many dates or when datetime variables are required.

Jack Shoemaker (1991) implemented a fuzzy merge by screening the results of a many:many merge.

Howard Levine (1991) describes the use of format table-lookups to achieve a fuzzy merge. Levine cited this technique as an example of SAS coding prior to the availability of PROC SQL, which makes fuzzy merges easy:

```sas
proc sql;
create table ACCL as
select ACCIDENT.*,
HIGHWAY.LANES
from ACCIDENT left join HIGHWAY
on ACCIDENT.HWYNAME = HIGHWAY.HWYNAME
and HIGHWAY.LOW <= ACCIDENT.MILEPOST < HIGHWAY.HIGH;
quit;
```

**RANGE MERGE**

In 1985, I became interested in the problem of how these range-keyed tables could be maintained. In particular, I wanted to find an analog to the MERGE statement, so I could combine two range-keyed tables (not
necessarily with matching ranges) and get a range-keyed result. Given:

\[
\begin{align*}
[1, 3) &= A \\
[3, 5) &= B \\
[5, 9) &= C \\
[1, 4) &= X \\
[4, 9) &= Y
\end{align*}
\]

I wanted the result

\[
\begin{align*}
[1, 3) &= \{A, X\} \\
[3, 4) &= \{B, X\} \\
[4, 5) &= \{B, Y\} \\
[5, 9) &= \{C, Y\}
\end{align*}
\]

I was able to modify Davis, et. al.'s program to accomplish this by explicitly considering each possible range overlap condition. It took three days of hard work, and the result was too complex to inspire complete confidence.

In 1989 I began an assignment managing experimental treatment databases keyed by date ranges. After years of thinking about range-keyed data, I finally realized a solution that is so simple that I would call it obvious, except that I have never seen it used or published.

The key to the solution is to first generate observations with the required output range-key values, then attach the data. Neither the SQL JOIN operator nor the SAS DATA-step MERGE statement are suitable. When merging two one-observation data sets, JOIN and MERGE will yield either one observation (a match) or two observations (a nonmatch). But when we merge the two ranges \([1,3)\) and \([2,4)\) together, the result should have three ranges: \([1,2), \[2,3), \text{ and } [3,4)\).

The solution requires a shift in perspective, considering ranges as a set of boundaries.

Different output observations are needed only when data changes. If a point is not used as a boundary in any input data set, it is not needed as a boundary in the output data set. Therefore, the set of boundaries of the output data is the union of all boundaries used in the input data.

These output boundaries can then be converted into output ranges. I call these "subranges" because one or more can always be combined to match any range found in the input data.

For example, the boundaries of the range merge in the example above are:

\[
\begin{array}{cccc}
\text{Data set L} & \text{Data set R} & \text{Result} & \text{Range} \\
1 & 1 & 1 & \[1, 3) \\
3 & - & 3 & \[3, 4) \\
- & 4 & 4 & \[3, 4) \\
5 & - & 5 & \[4, 5) \\
9 & 9 & 9 & \[5, 9)
\end{array}
\]

The following diagram illustrates how the subranges are constructed and used:

![Diagram](https://via.placeholder.com/150)

**Example**

At the beginning of the paper, I described a range merge application involving SCHOOL hours and JOB hours. The SAS code to create the subranges could be:

```sas
* find the union of all boundaries; data SUBRANGE; set SCHOOL (keep=ID START rename=(START=STOP)) SCHOOL (keep=ID STOP ) JOB (keep=ID START rename=(START=STOP)) JOB (keep=ID STOP ); by ID STOP; if first.STOP then output; run;
```
* convert boundaries back to ranges;
data SUBRANGE;
  set SUBRANGE;
  by ID;
  START = lag(STOP);
  if not first.ID then
    output;
run;

After the subranges have been created, each range-keyed data set is merged with the subrange data set. Only the START value of the subrange needs to be compared to the range, because the subranges were constructed such that each fits inside one of the ranges. For example, subranges [1,3) and [3,4) match range [1,4). This property makes it possible to use fuzzy merge (point-to-range) techniques such as:

```sql
proc sql;
create table SCHOOL2 as
  select SUBRANGE.START,
         SUBRANGE.STOP,
         SCHOOL.HOURS
  from SUBRANGE,
       SCHOOL
  where SUBRANGE.ID = SCHOOL.ID
     and SCHOOL.START <= SUBRANGE.START
     < SCHOOL.STOP;
quit;
```

This process needs to be repeated for each input data set. In this example, we need to join the JOB data set with the SUBRANGE data set. SQL can perform all of these joins with a single SELECT statement.

Now we have two data sets (SCHOOL2 and JOB2) with corresponding key values on (ID, [START, STOP]), so conventional merge techniques can be used:

```sas
data MERGED;
  merge JOB2 SCHOOL2;
  by ID START STOP;
run;
```

### RANGE UPDATE

Exactly the same method can be used to prepare data sets for a range update operation.

In the application described at the beginning of the paper, we have a data set HIGHWAY of highway maintenance data, keyed by HWYNAME and the milepost range-key (LOW, HIGH). A second data set UPDATE has the same structure, but contains only update transactions.

The following procedure can be used:

1. Create a SUBRANGE data set based upon the range-key values of both input data sets.
2. Merge data set SUBRANGE with each input data set individually.
3. Use existing point-key facilities such as the UPDATE or MODIFY statement to perform the update.

After the updates have been applied, adjacent observations may have identical data. For example:

```
<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
<th>Lanes (Old)</th>
<th>Lanes (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>28</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>90</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
```

These observations do not need to be combined, but the "projection" program (described below) can be used for this purpose if desired.

### RESTRICTION

In relational database terms, "restriction" means to subset the rows of a table. Data sets keyed by point keys can easily be restricted by itemizing the desired values. Restricting data sets keyed by range keys is more problematic because part of the range may be desired while the rest is not.

Suppose we are only interested in mileposts (20, 50) of the HIGHWAY data set. We could use a program such as:
data RESTRICT;
set HIGHWAY;
where HIGH > 20 and LOW < 50;
LOW = max (LOW, 20);
HIGH = min (HIGH, 50);
run;

PROJECTION

Suppose we have a data set describing experimental treatment (drug and dose), keyed by (PATIENT, [START, STOP]). We wish to reduce this to a smaller data set describing drug alone:

Before:

<table>
<thead>
<tr>
<th>Patient</th>
<th>Start</th>
<th>Stop</th>
<th>Drug</th>
<th>Dose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01JAN90</td>
<td>20AUG91</td>
<td>A</td>
<td>10mg</td>
</tr>
<tr>
<td>1</td>
<td>20AUG91</td>
<td>30MAY92</td>
<td>A</td>
<td>20mg</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>Patient</th>
<th>Start</th>
<th>Stop</th>
<th>Drug</th>
<th>Dose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01JAN90</td>
<td>30MAY92</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

This is an ideal application for the NOTSORTED option of the BY statement, which allows us to detect whenever any variables in the BY statement change. The BY statement must include all input data set variables except for the range keys.

The following code would do the trick:

```
proc sort data=TIMEDOSE
(keep=PATIENT START STOP DRUG)
out=DRUG;
   by PATIENT START STOP;
run;
data DRUG;
   set DRUG;
   by PATIENT DRUG notsorted;
   if first.DRUG then
      START = START;
   if last.DRUG then
      output;
   retain _START_
      format _START_ date.;
   drop START;
   rename _START_ = START;
run;
```

Range-key projection can also be used as a means of data compression, even when the data set does not contain any range keys. This technique can be used whenever data can be sorted such that adjacent observations often have the same data values. For example, different counties often have the same tax rate. We could compress:

<table>
<thead>
<tr>
<th>County</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blaine</td>
<td>4%</td>
</tr>
<tr>
<td>Keystone</td>
<td>4%</td>
</tr>
<tr>
<td>Milton</td>
<td>4%</td>
</tr>
<tr>
<td>Smith</td>
<td>5%</td>
</tr>
<tr>
<td>Windsor</td>
<td>4%</td>
</tr>
</tbody>
</table>

into

<table>
<thead>
<tr>
<th>Start</th>
<th>Stop</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blaine</td>
<td>Milton</td>
<td>4%</td>
</tr>
<tr>
<td>Smith</td>
<td>Smith</td>
<td>5%</td>
</tr>
<tr>
<td>Windsor</td>
<td>Windsor</td>
<td>4%</td>
</tr>
</tbody>
</table>

and then use a fuzzy merge on the point-key COUNTY to retrieve the tax rates. Note that this technique is effective even though the alphabetic sequence of county names has no substantive meaning. It depends upon the frequent occurrence of certain values of RATE.

SUMMARIZATION

Projection drops non-key variables; summarization drops point-key variables. For example, consider a data set of software licenses:

<table>
<thead>
<tr>
<th>SITE</th>
<th>START</th>
<th>STOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01JAN91</td>
<td>20MAY92</td>
</tr>
<tr>
<td>2</td>
<td>01JUL91</td>
<td>31DEC91</td>
</tr>
<tr>
<td>3</td>
<td>20SEP79</td>
<td>28APR91</td>
</tr>
</tbody>
</table>

How do we summarize across site in order to find the total number of sites for any given date?

The first step is to produce subranges applicable across all sites:

```
data SUBRANGE;
   set LICENSE (keep = START
         rename=(START=STOP))
   LICENSE (keep = STOP);
run;
```
Now merge the range-keyed data with the subranges, and count observations within the subranges:

```
proc sql;
create table EXPANDED as
select SUBRANGE.START, 
     SUBRANGE.STOP, 
     LICENSE.SITE
from SUBRANGE, LICENSE
where LICENSE.START<= 
     SUBRANGE.START < LICENSE.STOP;
quit;
```

The result looks like:

<table>
<thead>
<tr>
<th>START</th>
<th>STOP</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>20SEP79</td>
<td>01JAN91</td>
<td>1</td>
</tr>
<tr>
<td>01JAN91</td>
<td>28APR91</td>
<td>2</td>
</tr>
<tr>
<td>28APR91</td>
<td>01JUL91</td>
<td>1</td>
</tr>
<tr>
<td>01JUL91</td>
<td>31DEC91</td>
<td>2</td>
</tr>
<tr>
<td>31DEC91</td>
<td>20MAY92</td>
<td>1</td>
</tr>
</tbody>
</table>

APPENDIX: RANGE DATA STRUCTURES

Boundaries versus Ranges

Ranges can be represented either as ordered pairs of keys (describing line segments), or as boundaries (describing change points). Here is an employment history stored by ordered pairs of dates:

<table>
<thead>
<tr>
<th>ID</th>
<th>Start</th>
<th>Stop</th>
<th>Employer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;low&gt;</td>
<td>17MAR80</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>17MAR80</td>
<td>10JUL84</td>
<td>Acme WidgCorp</td>
</tr>
<tr>
<td>1</td>
<td>10JUL84</td>
<td>&lt;high&gt;</td>
<td>Fred's Store</td>
</tr>
</tbody>
</table>

and here is the same employment history stored by boundary:

<table>
<thead>
<tr>
<th>ID</th>
<th>Start</th>
<th>Employer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;low&gt;</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>17MAR80</td>
<td>Acme WidgCorp</td>
</tr>
<tr>
<td>1</td>
<td>10JUL84</td>
<td>Fred's Store</td>
</tr>
</tbody>
</table>

I have not established a clear preference either way. I lean toward the ordered pairs approach, which is more efficient for SQL and provides a convenient representation of update transactions. On the other hand, data sets keyed by boundaries store each date only once, which saves storage space and insures that ranges cannot illegally overlap.

This paper uses the ordered pairs approach throughout. Davis, et. al. used the boundary data structure for their fuzzy merge application.

To convert from boundaries to ordered pairs, use a program such as:

```
data PAIRS;
set BOUNDS 
    (rename=(BOUNDARY=START)); 
    by ID ;
if last.ID then 
    STOP = 99999; 
    * max stop; 
else do; 
    * STOP=lead(BOUNDARY) ; 
NPLUS = N + 1; 
    set BOUNoS-(keep=BOUNDARY 
        rename=(BOUNDARY=STOP)) 
        point=_NPLUS_; 
    end; 
run;
```

To convert from ordered pairs to boundaries, use a program such as:

```
data BOUNDS;
set PAIRS 
    (rename=(START=BOUNDARY)) 
    PAIRS (keep=ID STOP 
        rename=(STOP=BOUNDARY)); 
    by ID ;
where BOUNDARY ne 99999; 
if first.BOUNDARY; 
r
```
Infinite Range Boundaries

It is sometimes desirable to represent ranges continuing indefinitely into the past and future. The latter represents the "current" state.

The correct results can be obtained by using values that lie outside the applicable range, such as .L for negative infinity, and 99999 for positive infinity.

Another alternative is to use .H for positive infinity and modify the programs to treat this value as a special case.

Continuous versus Discrete values

Ranges are sometimes expressed inclusive of the endpoints:

['01 JAN91'd, '31 DEC91'd]
['01 JAN92'd, '31 DEC92'd]

Although this is cosmetically attractive, this convention makes programming more difficult and less intelligible.

When ranges are represented inclusively:

- Given the range
  ['01 JAN92'd, '01 JAN92'd)
  then STOP - START = 0 even though this represents a duration of one day.

- Given ranges
  ['01 JAN92'd, '01 JAN92'd)
  ['02 JAN92'd, '02 JAN92'd)
  then START - lag(STOP) = 1, even though there is no gap between the two ranges.

As a result, programs require extensive adjustments using constants, such as:

DURATION = STOP - START + 1;

These adjustments are the root cause of many difficulties:

- Programming is unnecessarily complex.

- SQL cannot evaluate joins of the form
  where START - 1 = STOP;
  as efficiently as equijoins such as
  where START = STOP;

- Constants are unit-dependent. Changes of units (such as from dates to datetimes) may require extensive revisions to existing programs.

For these reasons, I follow the rule that if two ranges are adjacent, then the STOP value of the first equals the START value of the second. For example,

['01 JAN91'd, '01 JAN92'd)
['01 JAN92'd, '01 JAN93'd)

This eliminates the need for adjustment constants:

DURATION = STOP - START;

A convention must be adopted to avoid ambiguities at the boundaries. In this paper, I consider the boundary point to match the START value rather than the STOP value. If you adopt the opposite convention, the comparison operators may need to be changed.

The programs in this paper assume that all range keys in all data sets follow the same conventions. One consequence is that ranges cannot represent a single point.

Dates versus Datetimes

The date '25 DEC92'd corresponds to the datetime range

['25 DEC92:00:00'dt, '26 DEC92:00:00'dt)

Midnight is considered to be part of the new day, in accord with the SAS datetime formats and DATEPART function.

Date ranges can be converted to datetime ranges simply by using the time '00:00', so the two ranges below both represent the year 1991:

['01 JAN91'd, '01 JAN92'd)
['01 JAN91:00:00'dt, '01 JAN92:00:00'dt)
NOTES

1. The notation ‘[‘ indicates that the endpoint is included in the interval. The notation ‘)’ indicates that the endpoint is not included.

2. Data set HIGHWAY is keyed by \{HWYNAME, [LOW, HIGH]\}; while the HIGHWAYS data set used by Davis, et. al. was keyed by \{HWYNAME, MILEPOST\}. The Appendix of this paper discusses range data structures.

3. The programs in this paper are written so as to clarify the points under discussion, not for efficiency. In particular, I have restricted the use of SQL to fuzzy merges, although in practice I prefer to use it for most applications.

4. If you do not have SQL available, the method used by Davis, et. al. can be used to perform this and other fuzzy merges throughout this paper.

AUTHOR CONTACT

Andrew A. Norton
Trilogy Consulting Corporation
5148 Lovers Lane
Kalamazoo, Michigan 49002
(616) 344-4100
Internet: 76350.1604@compuserve.com

REFERENCES


ACKNOWLEDGEMENTS

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Thanks to Bruce Bandstra for helping to format this document.

An earlier version of this paper appeared in Norton (1992).