

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.

Some applications need to use line segments as keys, rather than points. For example, hospitalization data may be keyed by datetime-ranges of the form¹ (DT_START, DT_STOP]. Data keyed by ranges require 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 susceptible to the same approach.

Fuzzy Merge

In the SAS® data set HIGHWAY, each observation corresponds 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. The data set is keyed by {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. 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 [LOW, HIGH) of milepost values specified on the HIGHWAYS dataset.

Range Merge

Two data sets are keyed by {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The HIGHWAYS data set has become outdated.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.

Projection

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

We are only interested in the drug information. By combining observations representing changes of dose (with no change of drug), 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 is used

to describe the subsetting of columns (and elimination of any resulting duplicate rows).

Summarization

We have a dataset 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.

FUZZY MERGE

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. Their seminal 1982 SUGI paper, "Techniques for Interrelating and Reducing Data Bases Using SAS", 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 milepost²). They used the example (page 422):

```
data ACCL (keep=ACCID HWYNAME
              MILEPOST HOUR
              SEVERITY LLANES
              rename=(LLANES=LANES));
set HIGHWAYS (in=INHWWY)
  ACCIDENT;
  by HWYNAME MILEPOST;
  retain LLANES;
  if INHWWY then LLANES=LANES;
  else output;
run;
```

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 falls apart 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 cites this technique as an example of SAS coding prior to the availability of PROC SQL, which makes fuzzy merges easy³:

```
proc sql;
  create table ACCL as
  select ACCIDENT.*
         ,HIGHWAY.LANES
  from   ACCIDENT left join HIGHWAY
         on ACCIDENT.HWYNAME
           = HIGHWAY.HWYNAME
         and HIGHWAY.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:

(1, 3] = A	(1, 4] = X
(3, 5] = B	(4, 9] = Y
(5, 9] = C	

I wanted the result

(1, 3] = {A, X}
(3, 4] = {B, X}
(4, 5] = {B, Y}
(5, 9] = {C, Y}

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 at LOREX Pharmaceuticals, managing experimental treatment databases keyed by date ranges. After years of thinking about range-keyed data, I finally realized the solution: so simple that I would call it obvious, except that I have never seen it used or published.

The crux of the problem is to 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 needs three ranges: (1,2], (2,3], 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. The boundaries of the output data set are the union of all boundaries used in the input data sets.

These output boundaries can then be converted into output ranges. I call these "subranges" because they have the property that 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:

<u>Data set A</u>	<u>Data set B</u>	<u>Result</u>	<u>Range</u>
1	1	1	
3	-	3	(1,3]
-	4	4	(3,4]
5	-	5	(4,5]
9	9	9	(5,9]

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:

```

data SUBRANGE;
  merge
    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;
  START = lag(STOP);
  if not first.ID then
    output;
run;

```

After the subranges have been created, each range-keyed data set has to be merged with the subrange data set. For example, subranges (1,3] and (3,4] would both match range (1,4]. I use code such as⁴:

```

proc sql;
  create table SCHOOL2 as
    select SUBRANGE.START
           ,SUBRANGE.STOP
           ,R.*
    from   SUBRANGE
           ,SCHOOL
           (rename=(START=_START_
                     (STOP =_STOP_)
                   ) as R
    where  R._START_ <=SUBRANGE.START
           and SUBRANGE.STOP<=R._STOP_;
quit;

```

This process⁵ needs to be repeated for each input data set. In the current example, we need to also merge SUBRANGE data set with the JOB data set.

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

```

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 dataset UPDATE has the same structure, but only contains update transactions.

The following procedure can be used:

1. Create a SUBRANGE dataset 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:

<u>Low</u>	<u>High</u>	<u>Lanes (Old)</u>	<u>Lanes (New)</u>
0	24	4	4
24	28	2	4 ←
28	90	4	4

These observations do not need to be combined, but the "projection" program described in the next section can be used for this purpose if desired.

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:

<u>Patient</u>	<u>Start</u>	<u>Stop</u>	<u>Drug</u>	<u>Dose</u>
1	01JAN90	20AUG91	A	10mg
1	20AUG91	30MAY92	A	20mg

After:

<u>Patient</u>	<u>Start</u>	<u>Stop</u>	<u>Drug</u>
1	01JAN90	30MAY92	A

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 of the variables on the output data set except for the range keys.

For this application, 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;
  drop START;
  rename _START_ = START;
run;
```

Projection of range keys provides an interesting opportunity for data compression even when there are no range keys. This technique can be used whenever data can be sorted such that adjacent observations tend to have the same data values. For example, different counties often have the same tax rate. We could compress:

<u>County</u>	<u>Rate</u>
Blaine	4%
Keystone	4%
Milton	4%
Smith	5%
Windsor	4%

into

<u>Start</u>	<u>Stop</u>	<u>Rate</u>
Blaine	Milton	4%
Smith	Smith	5%
Windsor	Windsor	4%

and then use a fuzzy merge on the point-keys 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 fact that there exist some values of the variable RATE that occur frequently.

SUMMARIZATION

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

<u>SITE</u>	<u>START</u>	<u>STOP</u>
1	01JAN91	20MAY92
2	01JUL91	31DEC91
3	20SEP79	28APR91

How do we summarize across site in order to produce a data set containing total number of sites on any given date?

This application differs from the others I have discussed because we may encounter multiple observations referring to a single date (once we have dropped SITE).

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

```
proc sql;
  create table SUBRANGE as
    select coalesce(START.START,
                   STOP.STOP )
      as STOP
    from   ( select START
              from   LICENSE
            ) as START
          full join
            ( select STOP
              from   LICENSE
            ) as STOP
          on START.START=STOP.STOP;
quit;

data SUBRANGE;
  set SUBRANGE;
  START = lag (STOP);
  if _N_ > 1 then
    output;
run;
```

Now we merge the range-keyed data with the subranges, and count observations within the subranges. This can be done in a single SQL step:

```
proc sql;
  create table SITE CNT as
  select   SUBRANGE.START
          ,SUBRANGE.STOP
          ,count(*) as COUNT
  from     SUBRANGE, LICENSE
  where    LICENSE.START<=SUBRANGE.START
          and SUBRANGE.STOP<=LICENSE.STOP
  group by START, STOP;
quit;
```

The result looks like:

<u>START</u>	<u>STOP</u>	<u>COUNT</u>
20SEP79	01JAN91	1
01JAN91	28APR91	2
28APR91	01JUL91	1
01JUL91	31DEC91	2
31DEC91	20MAY92	1

ENHANCEMENT: Infinite Range Boundaries

It is sometimes desirable to represent ranges continuing indefinitely into the past and through the future.

Negative infinity can be represented using the SAS missing value .L (meaning "Low").

Using a SAS missing value to represent positive infinity requires extensive programming changes, because SAS missing values sort low. A better alternative may be to use a very large number that could never be a legitimate value.

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:

<u>Start</u>	<u>Stop</u>	<u>Employer</u>
<low>	17MAR80	none
17MAR80	10JUL84	Acme WidgCorp
10JUL84	<high>	Fred's Store

and here is the same employment history stored by boundary:

<u>Start</u>	<u>Employer</u>
<low>	none
17MAR80	Acme WidgCorp
10JUL84	Fred's Store

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.

Continuous versus Discrete values

Here I do have a strong opinion. If two ranges are adjacent, then the STOP value of the first should equal the START value of the second (at least for internal computations). This is contrary to the common practice of expressing date ranges inclusive of the endpoints:

```
[01JAN91, 31JAN91]
[01JAN92, 01JAN92]
```

When ranges are represented inclusively:

- 01JAN92 - 01JAN92 = 0, even though this represents a duration of one day.
- 01JAN92 - 31JAN91 = 1, even though there is no gap between the two ranges.

These mathematical anomalies are the root cause of many difficulties:

- SQL cannot evaluate joins on arithmetic expressions
WHERE START - 1 = STOP;
as efficiently as equijoins
WHERE START = STOP;
- Changes of units, such as from dates to datetimes, may require changes to duration constants in existing programs.
- Programming is unnecessarily complex.

If the STOP point equals the next START point, a convention must be adopted to avoid ambiguities at the boundaries. In this paper, I consider the boundary point to match the STOP value rather than the START value (except in the fuzzy merge example). The comparison operators in the examples may need to be changed if you adopt the opposite convention.

NOTES

1. The notation '[' indicates that the endpoint is not included in the interval. The notation ']' indicates that the endpoint is included.
2. Davis, *et. al.* used only the MILEPOST value as a key. Most programmers (including myself) use both the start and stop points as keys.
3. Data set HIGHWAY is keyed by {HWYNAME, [LOW, HIGH]}, while the HIGHWAYS data set used by Davis, *et. al.* was keyed by {HWYNAME, MILEPOST}. To be compatible with their example, each range includes the LOW point. In the remainder of this paper, ranges include the HIGH point. See the Appendix of this paper for details.
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.

5. The renaming of variables in this SQL step is necessary to avoid conflicts between the variables START and STOP found in both data sets SCHOOL and WORK. The code would be simpler if I explicitly listed the variables in R rather than using the syntax R.*, but the latter avoids future maintenance if there are changes in the contents of data set R.

AUTHOR CONTACT

Andrew A. Norton
Trilogy Consulting Corporation
5822 Lovers Lane, Suite 100C
Kalamazoo, Michigan 49002
(616) 344-2191

internet: 76350.1604@compuserve.com

REFERENCES

Davis, James W., Steven Flint, and Robert U. Anderson (1982), "Techniques for Interrelating and Reducing Data Bases Using SAS", in *SAS Users Group International Proceedings of the Seventh Annual Conference*. Cary, NC: SAS Institute.

Levine, Howard (1991), "Comparing Proc SQL to Standard SAS Coding Techniques", in *Proceedings of the Fourth Annual NorthEast SAS Users Group Conference*.

Shoemaker, Jack (1991), "Fuzzy Merges (How to Make MERGE Really Work)", in *Proceedings of the Fourth Annual NorthEast SAS Users Group Conference*.

ACKNOWLEDGEMENTS

I would like to thank LOREX Pharmaceuticals of Skokie, Illinois for providing the opportunity to explore these issues.

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.