ANSI X3H2-96-264r1
ISO/IEC JTC1/SC21/WG3 DBL MCI-144

I S O
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

May 10, 1996

Subject:	SQL/Temporal
Title:	Road Map Queries in the US and UK Proposals
Source:	ANSI Expert's Contribution
Authors:	Richard T. Snodgrass
Abstract:	This document compares queries from the Road Map using constructs proposed by the US and the UK.

References

[1] Melton, J. (ed.) *SQL/Temporal.* March, 1996. (ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-009.)

[2] Snodgrass, R. T. *A Road Map of Additions to SQL/Temporal.* 1996. (ISO/IEC JTC1/SC21/WG3 DBL MCI-99, ANSI X3H2-96-013r1.)

[3] Snodgrass, R. T., M. H. Böhlen, C. S. Jensen and A. Steiner. *Adding Valid Time to SQL/Temporal.* 1996. (ISO/IEC JTC1/SC21/WG3 DBL MCI-142, ANSI X3H2-96-151r1.)

[4] Snodgrass, R. T., M. H. Böhlen, C. S. Jensen and A. Steiner. *Adding Transaction Time to SQL/Temporal.* 1996. (ISO/IEC JTC1/SC21/WG3 DBL MCI-143, ANSI X3H2-96-152r1.)

[5] UK change proposal, *Expanded Table Operations.* 1996. (ISO/IEC JTC1/SC21/WG3 DBL MCI-67)

This document compares the constructs proposed by the US [3, 4] and the UK [5], by applying them to the seven statements in the Road Map [2]. We assume that readers are somewhat familiar with both proposals, as well as the constructs currently in SQL/Temporal [1].

1. An `Employee` table has three columns: `Name`, `Manager` and `Dept`. In this table, `Manager` is a foreign key referencing the column `Employee.Name`. If `Employee` records time-varying information, this means that at each point in time where some specific value exists in the `Manager` column of `Employee`, that value must also exist in the `Name` column (possibly in another row) for that point in time.

   SQL without time:

   ```
   CREATE TABLE Employee(
      Name    VARCHAR(30),
      Manager VARCHAR(30) REFERENCES Employee (Name),
      Dept    VARCHAR(20))
   ```

   US proposal: (based on TSQL2):

   ```
   CREATE TABLE Employee(
      Name    VARCHAR(30),
      Manager VARCHAR(30) VALID REFERENCES Employee (Name),
      Dept    VARCHAR(20)) AS VALID DAY
   ```

   `VALID` specifies that the integrity constraint is to apply at each instant (in this case, each day). `AS VALID DAY` specifies that an unnamed column, maintained by the DBMS, will contain the row's timestamp.

   UK proposal (based on IXSQL):

   ```
   CREATE TABLE Employee(
      Name    VARCHAR(30),
      Manager VARCHAR(30),
      Dept    VARCHAR(20),
      When    PERIOD(DATE))
   ```

   The UK proposal does not have support for referential integrity for such tables. Additional syntax is needed. Currently the only way to do this is with a complex `ASSERTION`, left as an exercise for the reader.

2. "List the history of those employees who are *or were* not managers."

   SQL without time:

   ```
   SELECT Name FROM Employee WHERE Name NOT IN (SELECT Manager FROM Employee)
   ```

   US proposal:

   ```
   VALID SELECT Name FROM Employee WHERE Name NOT IN (SELECT Manager FROM Employee)
   ```

   To get the history of *any* query using the US proposal, simply prepend `VALID`. The change proposal and public-domain prototype demonstrate that the semantics may be implemented via a period-based algebra. The large body of performance-related research in temporal databases is applicable to implementing this semantics.

   UK proposal:

   ```
   WITH E1 AS (SELECT Name, EXPAND(When) AS EW FROM Employee)
   SELECT Name, PERIOD [ When, When ] AS When
   FROM E1, TABLE(E1.EW) AS E2(When)
   WHERE Name NOT IN (SELECT Manager FROM Employee AS E3 WHERE E3.When = E2.When)
   NORMALIZE ON When
   ```

   The semantics of `EXPANDING` is to duplicate each row of the left and right argument tables for each granule (day) in the `When` period, perform the `NOT IN`, then `NORMALIZE` the `When` column back to a period.

   As an alternative, the user can take the original SQL query, above, and map it into the algebra, with `NOT IN` being mapped to relation difference.

   $$\pi_{\texttt{Name}}(\texttt{Employee}) - \pi_{\texttt{Manager}}(\texttt{Employee})$$

   Then the user can map this back into SQL.

   ```
   SELECT Name FROM Employee EXCEPT SELECT Manager FROM Employee
   ```

   As a third step, the user can map this into a temporal query using `EXPAND` and `NORMALIZE`.

   ```
   WITH E1 AS (SELECT Name, EXPAND(When) AS EW FROM Employee),
        E2 AS (SELECT Manager, EXPAND(When) AS EW FROM Employee),
        E3 AS (SELECT Name, When
               FROM E1, TABLE(E1.EW) AS E4(When)
               EXCEPT
               SELECT Manager AS Name, When
               FROM E2, TABLE(E2.EW) AS E4(When))
   SELECT Name, PERIOD [ When, When ] AS When
   FROM E3
   NORMALIZE ON When
   ```

   Finally, the user can recognize that this can be simplified using `EXPANDING`.

   ```
   SELECT Name FROM Employee EXCEPT EXPANDING(When) SELECT Manager FROM Employee
   ```

   This can also be done with the US proposal, omitting the complex third step.

   ```
   VALID SELECT Name FROM Employee EXCEPT SELECT Manager FROM Employee
   ```

   All of the UK alternatives have the problem (not shared by the US alternatives) that if the left-hand table has duplicates, then `NORMALIZE` will automatically remove them, yielding an incorrect result (as the original SQL query did not specify `DISTINCT`). It is an exercise to the reader to show how this English query can be correctly expressed using an explicit `When` column. It *is* possible to do so, but it is exceedingly difficult.

   There have been essentially no results published on how to optimize queries with expansion or normalize operations. Also, no general procedure has been provided for converting an arbitrary, non-temporal query into its temporal analogue using the UK constructs.

3. "Give the history of the number of employees in each department."

   SQL without time:

   ```
   SELECT Dept, COUNT(*) FROM Employee GROUP BY Dept
   ```

   US proposal:

   ```
   VALID SELECT Dept, COUNT(*) FROM Employee GROUP BY Dept
   ```

   The `VALID` specifies that we are interested in the time-varying count. The syntax is declarative. The semantics is specified on a row-by-row basis; changing the granularity from day to second will not impact its performance.

   UK proposal:

   ```
   WITH E1 AS (SELECT Name, Dept, EXPAND(When) AS EW FROM Employee)
   SELECT Dept, COUNT(*), PERIOD [ When, When ] AS When
   FROM E1, TABLE(E1.EW) AS E2(When)
   GROUP BY Dept, When
   NORMALIZE ON When
   ```

   The syntax is procedural: first expand, then execute the select, then normalize. The `EXPAND` operator generates a `SET` of `DAY`s, which is then used to duplicate the rows of `Employee`, one for each day each row is valid (the join on the third line). The `GROUP BY` ensures that the `COUNT` is performed separately for each day. The `NORMALIZE` converts the many rows, one for each day, into periods. If each row is valid on average for one year, then `E2` will have 360 *times* the number of rows of `Employee`, with a dramatic decrease in performance. Changing the granularity to second generates additional tuples on the order of a factor 10^5, which could seriously affect performance. The approach of using `EXPANDING` doesn't work here, because the aggregate should be evaluated between the `EXPAND` and the `NORMALIZE`.

4. "Change the manager of the tools department for 1994 to Bob."

   SQL without time:

   ```
   UPDATE Employee
   SET Manager = 'Bob' WHERE Dept = 'Tools'
   ```

   US proposal:

   ```
   VALID PERIOD '[1994-01-01 - 1994-12-31]' UPDATE Employee
   SET Manager = 'Bob' WHERE Dept = 'Tools'
   ```

   UK proposal:

   The UK proposal has no support for this operation. Instead, each row must be examined to determine the overlap with 1994, and adjusted with an `UPDATE` and two `INSERT` statements. This is left as an exercise for the reader.

5. To know when rows are inserted and (logically) deleted, we add transaction-time support.

   US proposal:

   ```
   ALTER TABLE Employee ADD TRANSACTION
   ```

   Since transaction time is automatically managed by the DBMS, system integrity is ensured. Due to temporal upward compatibility, the referential integrity works as before, as do updates, such as the one above.

   UK proposal:

   ```
   ALTER TABLE Employee ADD COLUMN InsertTime TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP
   ALTER TABLE Employee ADD COLUMN DeleteTime TIMESTAMP(3) DEFAULT NULL
   ```

There is no support for transaction time in the UK proposal. There is no way to ensure that the application correctly manages the information in these two columns. System integrity can easily be compromised. Adding these two columns also breaks the referential integrity constraint between `Manager` and `Name`. The referential integrity must be formulated as a complex assertion that takes the three time columns into account. Updates are more complicated when these additional columns are present.

6. "How many employees are in each department?"

   SQL without time:

```
SELECT Dept, COUNT(*) FROM Employee GROUP BY Dept
```

   US proposal:

```
SELECT Dept, COUNT(*) FROM Employee GROUP BY Dept
```

   This still works, because the default is to take the currently valid data that has not been deleted or updated (temporally upward compatible in both valid and transaction time).

   UK proposal:

```
WITH E1 AS (SELECT Name, Dept FROM Employee
      WHERE DeleteTime IS NULL AND CURRENT_DATE OVERLAPS When)
SELECT Dept, COUNT(*)
FROM E1
GROUP BY Dept
```

   Since temporal upward compatibility is not satisfied by the UK proposal, the user must explicit select the current information.

   To get the *history* of the number of employees, some changes are required.

   US proposal:

```
VALID SELECT Dept, COUNT(*) FROM Employee GROUP BY Dept
```

   We retain temporal upward compatibility in transaction time (i.e., the data that has not been deleted or updated), but specify sequenced valid semantics to get the history, via `VALID`.

   UK proposal:

```
WITH E1 AS (SELECT Name, Dept, EXPAND(When) AS EW FROM Employee
      WHERE DeleteTime IS NULL)
SELECT Dept, COUNT(*), PERIOD [ When, When ] AS When
FROM E1, TABLE(E1.EW) AS E2(When)
GROUP BY Dept, When
NORMALIZE ON When
```

   The user must explicit select the currently stored information in transaction time (`WHERE DeleteTime IS NULL`) and must `EXPAND` and `NORMALIZE` to compute the aggregate.

7. "When did we think that departments are overly large (over 25 employees)?"

   US proposal:

```
TRANSACTION SELECT Dept, COUNT(*)
FROM Employee
GROUP BY Dept
HAVING COUNT(*) > 25
```

   `TRANSACTION` specifies that we wish to look over past states of the table. `VALID` is not specified, as we want to know only about the information about current departments. The execution is on a row-by-row basis, and is independent of both the valid time and transaction time granularities.

   UK proposal:

```
WITH E1 AS (SELECT Name, Dept, EXPAND(WhenP) AS EW
            FROM (SELECT Name, Dept, PERIOD(InsertTime, DeleteTime) AS WhenP
                  FROM Employee
                  WHERE CURRENT_TIMESTAMP OVERLAPS When) AS ET)
SELECT Dept, COUNT(*), PERIOD [ When, When ] AS When
FROM E1, TABLE(E1.EW) AS E2(When)
GROUP BY Dept, When
HAVING COUNT(*) > 25
NORMALIZE ON When
```

   The transaction time granularity is generally no coarser than a millisecond. Compared with the US proposal, this query will expand into $3 \cdot 10^{10}$ times the number of rows in the `Employee` table. It is not clear how to optimize this query, as the result could change at any millisecond: the aggregate must be computed for each millisecond. It is doubtful that the UK query can even be computed with currently known query optimization/evaluation technology.