Temporal Query Languages

Fabio Grandi
fabio.grandi@unibo.it
DISI, Università di Bologna

A short course on Temporal Databases for DISI PhD students, 2016
Credits: most of the materials used is taken from slides prepared by Prof. M. Böhlen (Univ. of Zurich, Switzerland)
Temporal Query Languages

- Data model: $DM = (DS, QL)$
  - $DS$ is a set of data structures
  - $QL$ is a language for querying and updating the data structures

- Example: the relational data model is composed of relations and SQL (or relational algebra)

- Many extensions of the relational data model and SQL to support time have been proposed
Relational Algebra for the BCDM

- An algebra provides a procedural/operational language for a data structure that is suitable for implementation

- Algebra for the standard relational algebra operators in BCDM
  - Schema: \( R = (A_1, \ldots, A_n, T) \)
  - Domains: \( A_i \) has domain \( D_i \) and \( T \) has domain \( \mathcal{P}(D_{TT} \times D_{VT}) \)
    - \( D_{TT} \) is the transaction-time domain and \( D_{VT} \) is the valid-time domain
  - \( r \) is an instance relation of schema \( R \)
  - The operators then have the following signature

\[
\begin{align*}
\pi^B_D &: r \rightarrow r \\
\sigma^B_P &: r \rightarrow r \\
\cup^B &: r \times r \rightarrow r \\
\bowtie^B &: r \times r \rightarrow r \\
\end{align*}
\]

\[
\begin{align*}
\pi_D &: r \rightarrow r \\
\sigma_P &: r \rightarrow r \\
\cup &: r \times r \rightarrow r \\
\bowtie &: r \times r \rightarrow r \\
\end{align*}
\]
Projection in BCDM

- Temporal projection: Project a relation $r$ with non-timestamp attributes $A_1, \ldots, A_n$ to a subset $D$ of attributes

\[
\pi_D^B(r) = \{ z^{(|D|+1)} \exists x \in r(z[D] = x[D]) \land \\
\forall y \in r(y[D] = z[D] \implies y[T] \subseteq z[T]) \land \\
\forall t \in z[T] \exists y \in r(y[D] = z[D] \land t \in y[T]) \}
\]

- Calculation of timestamps of result tuples
  - All chronons in any value-equivalent tuple of $r$ must be included and no spurious chronons can be introduced
  - (automatic coalescence is performed)

- Ex. Projection on the Emp attribute:

<table>
<thead>
<tr>
<th>dept</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emp</td>
<td>T</td>
</tr>
<tr>
<td>Jake</td>
<td>{(5,10), \ldots, (5,15), \ldots, (9,10), \ldots, (9,15), (10,5), \ldots, (10,20), (14,5), \ldots, (14,20), (15,10), \ldots, (15,15), \ldots, (19,10), \ldots, (19,15)}</td>
</tr>
<tr>
<td>Jake</td>
<td>{(now,10), \ldots, (now,15)}</td>
</tr>
<tr>
<td>Kate</td>
<td>{(now,25), \ldots, (now,30)}</td>
</tr>
<tr>
<td>Ship</td>
<td>{now,10), \ldots, (now,15}</td>
</tr>
<tr>
<td>Load</td>
<td>{(now,10), \ldots, (now,15)</td>
</tr>
<tr>
<td>Ship</td>
<td>{(now,25), \ldots, (now,30)}</td>
</tr>
</tbody>
</table>
Selection in BCDM

- Temporal selection: Select from relation r with non-timestamp attributes $A_1, \ldots, A_n$ all tuples that satisfy a predicate $P$ defined on the non-timestamp attributes

$$\sigma_P^B(r) = \{z | z \in r \land P(z[A])\}$$

- Ex. Select all tuples of employee Kate:

<table>
<thead>
<tr>
<th>dept</th>
<th>Emp</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jake</td>
<td>Ship</td>
<td></td>
</tr>
<tr>
<td>Jake</td>
<td>Load</td>
<td></td>
</tr>
<tr>
<td>Kate</td>
<td>Ship</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>${(5,10),\ldots,(5,15),\ldots,(9,10),\ldots,(9,15),\ldots,(10,5),\ldots,(10,20),\ldots,(14,5),\ldots,(14,20),\ldots,(15,10),\ldots,(15,15),\ldots,(19,10),\ldots,(19,15)}$</td>
</tr>
</tbody>
</table>

- Week 1, Day 1:  
  - Table:
    | Emp | Dept | T |
    |-----|------|---|
    | Jake| Ship | $\{(5,10),\ldots,(5,15),\ldots,(9,10),\ldots,(9,15),\ldots,(10,5),\ldots,(10,20),\ldots,(14,5),\ldots,(14,20),\ldots,(15,10),\ldots,(15,15),\ldots,(19,10),\ldots,(19,15)\}$ |

<table>
<thead>
<tr>
<th>week 2, day 1:</th>
<th>Emp</th>
<th>Dept</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>week 2, day 1:</td>
<td>Kate</td>
<td>Ship</td>
<td>${(now,25),\ldots,(now,30)}$</td>
</tr>
</tbody>
</table>
Union in BCDM

- Temporal union: Compute the union of tuples from two relations $r_1$ and $r_2$ that are instances of the same schema (or union-compatible schemas)

\[
r_1 \cup^B r_2 = \{z^{(n+1)} | (\exists x \in r_1 \exists y \in r_2 (z[A] = x[A] = y[A] \land z[T] = x[T] \cup y[T])) \lor \\
(\exists x \in r_1 (z[A] = x[A] \land (\neg \exists y \in r_2 (y[A] = x[A])) \land z[T] = x[T])) \lor \\
(\exists y \in r_2 (z[A] = y[A] \land (\neg \exists x \in r_1 (x[A] = y[A])) \land z[T] = y[T]))\}\]

- The first clause handles value-equivalent tuples found in $r_1$ and $r_2$
- The second (third) clause handles those tuples that are found only in $r_1$ ($r_2$)
Ex. Compute the union of relations dept and emp:

```latex
dep \cup^B emp
```
Difference in BCDM

- Temporal difference: Compute those tuples that are in \( r_1 \) and not in \( r_2 \) where the two relations are instances of the same schema (or union-compatible schemas)

\[
r_1 \ominus_B r_2 = \{ z^{(n+1)} | \exists x \in r_1 ((z[A] = x[A]) \wedge ((\exists y \in r_2 (z[A] = y[A]) \wedge z[T] = x[T] - y[T]) \vee (\neg \exists y \in r_2 (z[A] = y[A]) \wedge z[T] = x[T]))) \}
\]

- The last two lines compute the bitemporal element, depending on whether a value-equivalent tuple may be found in \( r_2 \) or not
Difference in BCDM

- Ex. Compute the difference of relations dept and emp:

\[ dept - B emp \]
Join in BCDM

- Temporal join: Two tuples join if they match on the join attributes $A_1, \ldots, A_n$ and have overlapping bitemporal-element timestamps
  - $r$ and $s$ are instances over the following schemas:
    \[
    R(A_1, \ldots, A_n, B_1, \ldots, B_l, T) = R(A, B, T)
    \]
    \[
    S(A_1, \ldots, A_n, C_1, \ldots, C_m, T) = S(A, C, T)
    \]
  - The timestamp of a result tuple is the intersection of the timestamps of the corresponding argument tuples

\[
\begin{align*}
    r \Join^B s &= \{ z^{(n+l+m+1)} | (\exists x \in r \exists y \in s (x[A] = y[A] \land x[T] \cap y[T] \neq \emptyset \land \forall z. z[A] = x[A] \land z[B] = x[B] \land z[C] = y[C] \land z[T] = x[T] \cap y[T]) \}\}
\end{align*}
\]
Join in BCDM

- Ex. Temporal join to compute “Who managed whom”?:

The timestamp is the overlap of timestamp regions of tuples with matching join attribute.
Timeslice Operators in BCDM

- **Transaction-timeslice operator:** selects the relation at transaction time \( t_1 \) (not exceeding the current time)
  - Takes a bitemporal relation \( r \) as input and returns a valid-time relation

\[
\rho_{t_1}^B(r) = \{ z^{(n+1)} \mid \exists x \in r(z[A] = x[A] \land z[T_v] = \{ t_2 | (t_1, t_2) \in x[T] \} \land z[T_v] \neq \emptyset) \}
\]

- **Valid-timeslice operator:** selects the relation at valid time \( t_2 \)
  - Takes a bitemporal relation \( r \) as input and returns a transaction-time relation

\[
\tau_{t_2}^B(r) = \{ z^{(n+1)} \mid \exists x \in r(z[A] = x[A] \land z[T_t] = \{ t_1 | (t_1, t_2) \in x[T] \} \land z[T_t] \neq \emptyset) \}
\]
Timeslice Operators in BCDM

- Timeslice operators can be extended for transaction-time and valid-time relations
  - $\rho^T$ gets as input a transaction-time relation and returns a snapshot relation
  - $\tau^V$ gets as input a valid-time relation and returns a snapshot relation

- Ex.

<table>
<thead>
<tr>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emp</td>
</tr>
<tr>
<td>Jake</td>
</tr>
<tr>
<td>Jake</td>
</tr>
<tr>
<td>Kate</td>
</tr>
</tbody>
</table>

- $\tau^B_{12} (\text{dept}) = \{ (\text{Jake}, \text{Ship}, \{5, \ldots, 19\}), (\text{Jake}, \text{Load}, \{\text{now}\}) \}$
- $\rho^V_7 (\tau^B_{12} (\text{dept})) = \{ (\text{Jake}, \text{Ship}) \}$
Sequenced Semantics

- There is a close relationship between a temporal and a non-temporal database:
  - the snapshot of a temporal relation at a time t is a non-temporal relation
  - a temporal relation is a collection of timestamped snapshots
- All non-temporal statements can be evaluated at each snapshot of a temporal database ("at each time point")
- There should be a close relationship between a temporal and a non-temporal statement:
  - e.g. a temporal aggregation should resemble a non-temporal aggregation
- With SQL this is not the case (remember temporal join versus join)
Sequenced Semantics

Notations (we assume R is a valid-time relation):

- Relation schema: $R(A_1, ..., A_n, T_S, T_E)$
- $r$ is a relation with schema $R$ (instance of $R$)
- $A_1, ..., A_n$ are the explicit (non-temporal) attributes
- $T_S, T_E$ are temporal attributes
  - $T_S$ is the valid time start
  - $T_E$ is the valid time end
- $z^{(n+2)}$ denotes a tuple of arity $n+2$
- We assume periods are half open intervals $[T_S, T_E)$
- We write $T$ to refer to the period $[T_S, T_E)$
  - $t \in T \equiv T_S \leq t < T_E$
Sequenced Semantics

Notations:

- The timeslice operator $\tau$ maps a temporal to a non-temporal relation
- Definition of the timeslice operator:
  \[
  \tau_t(r) = \{ z^{(n)} | \exists x \in r \ (z.A = x.A \land x.T_S \leq t < x.T_E) \}
  \]
- Two temporal relations, $r$ and $s$, are snapshot equivalent iff for all times $t$ their snapshots are identical
- Definition of snapshot equivalence:
  \[ r \equiv s \iff \forall t(\tau_t(r) = \tau_t(s)) \]
Sequenced Semantics

- Snapshot reducibility reduces the semantics of temporal operators to the semantics of the corresponding non-temporal operators.

- Temporal operator $\psi^T$ is snapshot-reducible to the non-temporal operator $\psi$ iff for all $t$:

\[
\tau_t(\psi^T(R_1, \ldots, R_n)) \equiv \psi(\tau_t(R_1), \ldots, \tau_t(R_n))
\]
Sequenced Semantics

Illustration of snapshot

Reducibility:

- \( D^T \) = temporal DB
- \( \psi^T = \{\sigma^T, \pi^T, \theta^T, x^T, U^T, -^T\} \)
- \( R^T \) = temporal result relation
- \( \tau_t \) = snapshot at time point \( t \)
- \( D_t \) = snapshot of \( D^T \) at time \( t \)
- \( \psi = \{\sigma, \pi, \theta, x, U, -\} \)
- \( R_t \) = result relation at time \( t \)

\[ \forall t : \tau_t(\psi^T(D^T)) = \psi(\tau_t(D^T)) \]
Sequenced Semantics

- A temporal relation can be viewed as made up of a sequence of timestamped snapshot relations.
- Mutual consistency of the two viewpoints along the time axis gives rise to the notion of snapshot reducibility.
- If period/element timestamping is adopted, timestamps of the argument tuples are taken into account when forming the timestamp associated to the result tuples (e.g. intersection is used when executing a join).
- Enforcement of snapshot reducibility gives rise to a sequenced semantics (i.e. “at each time point”) in query execution [Böhlen, Jensen, Snodgrass]
Non-SEQUENCED SEMANTICS

- Snapshot reducibility does not apply to queries involving predicates and functions over the timestamps of argument relations.

- In such queries, snapshots valid at different times have to be mixed in order to find the answer.

- Hence, their evaluation requires a non-sequenced semantics.

- Such queries give the full temporal expressivity to a temporal query language (and fully exploits the power of a temporal database).
  - Ex. Find the employees who were programmer before becoming DBA.
  - The information about being programmer and about being DBA must be found by combining (with a join) different snapshots.
Beyond Sequenced Semantics [Böhlen]

- Period-based semantics (even in a weak sense) requires the preservation of the individual timestamp periods through the application of operators.

- *Extended snapshot reducibility* allows non-sequenced queries to be executed with a sequenced semantics.

- It can be enforced via *timestamp propagation* (making copies of timestamp columns to be treated as explicit attributes).

- Enforcement of *change propagation* corresponds to a correct application of a sequenced semantics with true period-based timestamping (coalescence not automatic).

- It can be implemented via manipulation of lineage sets (sets of witness lists of argument tuples).
Upward Compatibility [Snodgrass et al.]

- Let $M_1=(DS_1,QL_1)$ and $M_2=(DS_2,QL_2)$ two data models, then $M_1$ is **syntactically upward compatible** with $M_2$ if
  - $\forall \text{db}_2 \in DS_2 \Rightarrow \text{db}_2 \in DS_1$
  - $\forall q_2 \in QL_2 \Rightarrow q_2 \in QL_1$
  (a database/query in $M_2$ is also a database/query in $M_1$)

- Let $M_1=(DS_1,QL_1)$ and $M_2=(DS_2,QL_2)$ two data models, then $M_1$ is **upward compatible** with $M_2$ if
  - $M_1$ is syntactically upward compatible with $M_2$
  - $\forall \text{db}_2 \in DS_2, \forall q_2 \in QL_2 \Rightarrow [[ q_2(\text{db}_2) ]]_{M_2} = [[ q_2(\text{db}_2) ]]_{M_1}$
  (evaluating a query on a database instance in $M_2$ gives identical results if evaluated in $M_1$)

- We will use this notion with $M_1 = TDB$ and $M_2 = \text{Rel. DB}$...
Upward Compatibility

- A Temporal Query Language (TQL) is upward compatible with SQL if
  - Traditional tables are also legal instances of tables in the underlying temporal data model
  - Traditional SQL queries are also queries in the TQL and give the same results when evaluated according to the TQL semantics (TQL and SQL queries give the same results on a non-temporal table)
Language Design Criteria

- **Expressive power**
  - Suitable for intended applications
  - Economy of encoding is relevant

- **Clarity**
  - Syntax should reflect the semantics
  - Consistent naming style

- **Consistency**
  - Upward compatibility with standards, e.g. SQL standard
  - Systematic (not a new construct per query, no exceptions)

- **Orthogonality**
  - Possibility to freely combine query language constructs
  - Zero-One-Infinity principle (the only reasonable numbers in a programming language design are zero, one, and infinity)

- **Closed-form evaluation**
  - The result of a query is a proper object of the data model
Comparison of Timestamps

- Comparison of Timestamps is part of every temporal query language
- Many query languages adopt (a variant of) Allen’s 13 period relations:

```
X before Y
Y after X

X equals Y

X meets Y
Y met_by X

X overlaps Y
Y overlapped_by X

X during Y
Y contains X

X starts Y
Y started_by X

X finishes Y
Y finished_by X
```
SQL + Abstract Data Types

- Extend existing language (e.g. SQL) with time data types and associated predicates and functions
  - e.g. predicates for timestamp comparison
- Earliest and (from a language design perspective) simplest approach
- Has limited impact on existing language and is well understood technically

- An abstract data type does not offer a systematic way to generalize snapshot queries to temporal queries
- New and very complex solutions must be invented (i.e. programmed) to implement common temporal operations:
  - Temporal join, temporal aggregates, coalescence…
  - Enforcement of key constraints, sequenced semantics…
The IXSQL Approach [Lorentzos et al.]

- IXSQL extends SQL-92 with (time) period data type
- Periods are convenient for representing temporal aspects, but create difficulties when formulating temporal queries
- IXSQL addresses this problem by normalizing timestamps so that they are aligned (identical or disjoint):
  - Function \textit{UNFOLD}: decompose a period-timestamped tuple into a set of point-timestamped tuples (one for each point in the original period)
  - Function \textit{FOLD}: collapse a set of point timestamped tuples into value-equivalent tuples timestamped with maximum periods
- General pattern for query processing using fold/unfold:
  1. Construct the point-based representation by unfolding the argument relation(s)
  2. Compute the query on point-based representation
  3. Fold the result to end up with an period-based representation
The IXSQL Approach

- Example of a temporal join (sequenced semantics):

```
SELECT DeptName, Location, DeptManager, Salary,
     intersect(Department.T, Employee.T) as T
FROM Employee, Department
WHERE EmpName = DeptManager
     AND Department.T overlaps Employee.T
     AND Location = 'Miami'
REFORMAT AS FOLD T
```

- (the REFORMAT AS FOLD instruction, i.e. UNFOLD to
time instants followed by FOLD to time periods, is
necessary for coalescence of tuples in the result)
The IXSQL Approach

- Only two functions, fold and unfold, are added to SQL
- Unfold can be used when needed to formulate queries about each time point (it is optional and not an invasive change at query language level)
- Efficient evaluation of queries formulated using fold/unfold has yet to be resolved
- Neither a purely point-based nor period-based view:
  - Sensitive to specific period representation of data (e.g. queries that do not use fold/unfold)
  - Fold/unfold only preserve information of a point-based view
    - Normalization step using unfold/fold loses period information
    - Fold is not the inverse of unfold (information about the original periods is lost)
  - The combination of “at each time point” and periods is not supported (sequenced semantics with periods cannot be supported)
The TSQL2 Language (Temporal SQL-92 Extension)
The TSQL2 Language

Desired features of the underlying data model that inspired the TSQL2 design:

- TSQL2 should not distinguish between value-equivalent instances (to provide conceptual simplicity)
- TSQL2 should support only one valid-time dimension
- TSQL2 should support transaction time
- For simplicity, tuple timestamping should be employed
- Event and state tables should be supported
- Valid-time support should include support for both the past and the future
- Timestamp values should not be limited in range or precision
The TSQL2 Language

Proper desired features of the query language that inspired the TSQL2 design:

- TSQL2 should be a consistent, fully upward compatible extension of SQL-92
- TSQL2 should allow the restructuring of tables on any set of attributes
- TSQL2 should allow for flexible temporal projection
- Operations in TSQL2 should not accord any explicit attributes special semantics (e.g. op. relying on keys)
- Temporal support should be optional, on a per-table basis
The TSQL2 Language

Proper desired features of the query language that inspired the TSQL2 design:

- User-defined time support should include instants, periods and intervals
- Existing aggregates should have temporal analogues in TSQL2
- Multiple calendars and multiple language support should be present in timestamp I/O and operations
- It should be possible to derive temporal and non-temporal tables from underlying temporal and non-temporal tables
The TSQL2 Language

Ease of implementation was made a priority in the design:

- TSQL2 tables should be implemented in terms of tables in some 1NF representational model
- TSQL2 should have an efficiently implementable algebra that allows for optimization and that is an extension of the snapshot algebra
- The TSQL2 data model should allow multiple representational data models
The TSQL2 Language

- Timestamping columns are “hidden columns” with an implied special semantics and syntactic defaults have been embedded in order to make the formulation of common temporal queries easier.

- For example, intersection of the valid time of all the relations involved in a query to be assigned as timestamps to the results is automatically done, yielding:
  - Snapshot reducibility is ensured
  - Sequenced semantics is enforced by default

- The implied sequenced semantics can be overridden via a custom temporal projection or explicit manipulation of timestamps for temporal selection.
Time Representation in TSQL2

- Time representation conforms to the BDCM
- Time is discrete with chronons as base unit
- Available base temporal datatypes:
  - Datetime (instant)
  - Period
  - Interval
- Such datatypes are inherited from the SQL-92 specification but with several flaws fixed
The Datetime Datatype

- Conforms to predefined SQL-92 types: DATE, TIME, TIMESTAMP (compliant to ISO 8601 standard formats)

- Examples:
  
  DATE '2016-02-29'
  DATE 'February 29, 2016'
  TIME '21:30:10'
  TIME '9:30:10 PM'
  TIMESTAMP '2015-12-31 12:00:00.00'
  TIMESTAMP 'Noon December 31, 2015'
The Period Datatype

- Represents open/closed time periods
- Examples:

  PERIOD '[March 2014]'
  PERIOD '(2010]'
  PERIOD '[1994-01-01 - 1994-01-31]'
  PERIOD '(12:15:00.0 - 12:16:00.0]'
  PERIOD '[Midnight July 1, 2013 - September 10, 2014 10:20 AM]'
The Interval Datatype

- Represents unanchored pure durations

- Examples:
  
  \[
  \begin{align*}
  \text{INTERVAL '10' YEAR} \\
  \text{INTERVAL 'November' DAY} \\
  \text{INTERVAL '3' WEEK} \\
  \text{INTERVAL '02:30' HOUR TO MINUTE} \\
  \text{INTERVAL '-20' SECOND} \quad \text{(cf. negative duration)}
  \end{align*}
  \]
Mixed Expressions

- A set of any datetime (period) data is an instant set (temporal element): in any case it is a set of chronons

- Examples:

```
PERIOD '[2014-01-01 - 2014-06-01]'
+ INTERVAL '10' MONTH
= PERIOD '[2014-11-01 - 2015-04-01]'

TIMESTAMP '2000-01-01 12:30'
+ INTERVAL 'February 2016' DAY
= TIMESTAMP '2000-01-30 12:30'
```
Mixed Expressions

- Further examples:
  
  PERIOD 'March 2014' + INTERVAL '10' DAY = PERIOD '2014-03-11 - 2014-04-10'

  TIMESTAMP '13:30 April 1, 2000'
  + INTERVAL '1' YEAR - INTERVAL '15' MINUTE = TIMESTAMP '2001-04-01 13:15'

- Special predefined constants:
  BEGINNING, FOREVER, INITIATION, UNTIL_CHANGED, CURRENT_TIMESTAMP, NOW (possibly with nobind option)
Schema Declaration and Modification

- Temporal definition clause AS... (6 temporal table types)
- Examples:
  
  CREATE TABLE Employee ( ... )
  AS VALID STATE

  CREATE TABLE Department ( ... )
  AS VALID AND TRANSACTION

  CREATE TABLE Transfer ( ... )
  AS VALID EVENT DAY

  ALTER TABLE Employee
  ADD TRANSACTION
Temporal Selection

- Selection based on temporal conditions in the WHERE clause
- Temporal comparison operators (for datetime, period, instant set and element): PRECEDES, =, OVERLAPS, MEETS, CONTAINS
- Comparison (<, =, >) and arithmetic (+, -, *) operators for intervals
- Various functions: BEGIN(.), END(.), FIRST(.), LAST(.), INTERSECT(..,..), +, -
- Constructors: PERIOD(..,..)
- Timestamp extractors: VALID(.), TRANSACTION(.)
Temporal Comparison Operators

- The semantics of TSQL2 temporal comparison operators corresponds to their meaning in natural language (whereas Allen’s operators have artificial and innatural names), following the SQL (SEQUEL) philosophy.

- $X \text{ PRECEDES } Y$ iff $\text{END}(X) < \text{BEGIN}(Y)$

- $X = Y$ iff $X$ and $Y$ are identical

- $X \text{ OVERLAPS } Y$ iff $X \cap Y \neq \emptyset$

- $X \text{ MEETS } Y$ iff $X \text{ PRECEDES } Y$ without any instants in between

- $X \text{ CONTAINS } Y$ iff $X \supseteq Y$
Temporal Comparison Operators

- The TSQL2 temporal comparison operators can be used with instants, periods and elements, and also for mixed comparisons (e.g. elements with instants)
- As to periods, TSQL2 is anyway Allen-complete
- \( X = Y \) has been preferred to \( X \) EQUALS \( Y \) not to introduce a new keyword
- For the same reason, inverse operators have not been considered necessary
  - \( X \) MET\_BY \( Y \) can be expressed as \( Y \) MEETS \( X \)
  - \( X \) FOLLOWS \( Y \) can be expressed as \( Y \) PRECEDES \( X \)
  - \( X \) DURING \( Y \) can be expressed as \( Y \) CONTAINS \( X \)
Temporal Selection - Examples

- SELECT * FROM Employee
  WHERE EmpName = 'Ted'

- SELECT Salary FROM Employee
  WHERE VALID(Employee) CONTAINS DATE 'NOW'

- SELECT * FROM Employee
  WHERE EmpName = 'Ted'
  AND VALID(Employee) OVERLAPS
    PERIOD '[2013]' + PERIOD '[2015]'
Temporal Selection - Examples

- SELECT EmpName, Salary
  FROM Employee
  WHERE FIRST(VALID(Employee)) CONTAINS

- SELECT EmpName, Salary
  FROM Employee
  WHERE Job = 'Programmer'
    AND LAST(VALID(Employee))
      PRECEDES DATE '2014-03-01'
Temporal Projection

- Assignment of a timestamp to the results of a query done with the \( \text{VALID (VALID INTERSECT)} \) clause

- Examples:

  - `SELECT SNAPSHOT EmpName, DateOfBirth FROM Employee WHERE Job='Engineer'`

  - `SELECT DISTINCT EmpName FROM Employee VALID PERIOD(DateOfBirth, DATE 'FOREVER') WHERE Job = 'Manager'`
Temporal Projection - Examples

- SELECT Department.*, Employee.Salary
  FROM Employee, Department
  VALID INTERSECT (Employee, Department)
  WHERE EmpName = DeptManager
  AND VALID(Employee)
  OVERLAPS VALID(Department)
  AND Location = 'Miami'

- SELECT Department.*, Employee.Salary
  FROM Employee, Department
  WHERE EmpName = DeptManager
  AND Location = 'Miami'
  
  (the same VALID clause as above is understood and, thus, the overlap is implied; cf. temporal join)
TSQL2 Range Variables

- The TSQL2 range variables generalize the concept of history variables [Grandi] and allow for temporal restructuring [Gadia] of a relation. Automatic coalescing of timestamps is implied.

- In the `FROM` clause:
  ```sql
  FROM Employee(EmpName) AS Emp
  ```
The variable `Emp` ranges over groups of tuples of the relations with the same `EmpName` attribute value. Grouping can also be based on periods.

- Notice that the clause `FROM Employee` is equivalent to `FROM Employee AS Employee` that is to `FROM Employee(*) AS Employee`
TSQL2 Range Variables

- Declaration of range variables (and, thus, grouping) can be nested:

  FROM Employee(EmpName) AS Emp, 
  Emp(Job) AS E1, E2

is equivalent to:

  FROM Employee(EmpName) AS Emp, 
  Employee(EmpName,Job) AS E1, E2
  WHERE E1.EmpName=Emp.EmpName
  AND E2.EmpName=Emp.EmpName

(groups are synchronized on the common attributes; nested declarations are “syntactic sugar”)

TSQL2 Range Variables

- Examples:

```sql
SELECT *
FROM Employee(EmpName,Salary) AS Emp
WHERE Salary = 2500
AND CAST(Emp AS INTERVAL YEAR) >= INTERVAL '2' YEAR
```

```sql
SELECT SNAPSHOT E1.EmpName, BEGIN(VALID(E2))
FROM Employee(EmpName) AS Emp,
    Emp(Job,Salary) AS E1, E2
WHERE E1 MEETS E2
    AND E1.Job <> E2.Job
    AND E1.Salary = E2.Salary
```
TSQL2 Range Variables

Examples:

```
SELECT E1.EmpName, E1.Job
FROM Employee(EmpName) AS Emp,
    Emp(Job)(PERIOD) AS E1, E2, E3
WHERE E1 MEETS E2 AND E2 MEETS E3
    AND E2.Job = 'Manager'
```

```
SELECT Emp.*
FROM Employee(EmpName) AS Emp,
    Emp(Job) AS E1, Emp(Salary) AS E2
WHERE E1.Salary = 2300 AND E2.Job = 'DeptHead'
    AND BEGIN(VALID(E2)) - END(VALID(E1))
        > INTERVAL '18' MONTH
```
The **VALID** clause allows for the specification of the *applicability period* of the modification.

**Examples:**

```sql
INSERT INTO Employee
VALUES ('Kim', '1982-05-15', 'Engineer', 2500)
VALID PERIOD( DATE '2016-01-01', NOBIND(DATE 'NOW'))
```

<table>
<thead>
<tr>
<th>EmpName</th>
<th>DateOfBirth</th>
<th>Job</th>
<th>Salary</th>
<th>VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kim</td>
<td>15/5/1982</td>
<td>Engineer</td>
<td>2500</td>
<td>[1/1/2016, Now)</td>
</tr>
</tbody>
</table>
TSQL2 Modification Operations

- Examples:

```sql
UPDATE Employee
SET Salary = Salary + 200
WHERE EmpName = 'Kim'
AND VALID(Employee)
    CONTAINS DATE 'CURRENT_TIMESTAMP'
VALID PERIOD 'February 2016'
```

<table>
<thead>
<tr>
<th>EmpName</th>
<th>DateOfBirth</th>
<th>Job</th>
<th>Salary</th>
<th>VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kim</td>
<td>15/5/1982</td>
<td>Engineer</td>
<td>2500</td>
<td>[1/1/2016, 1/2/2016)</td>
</tr>
<tr>
<td>Kim</td>
<td>15/5/1982</td>
<td>Engineer</td>
<td>2700</td>
<td>[1/2/2016, 1/3/2016]</td>
</tr>
<tr>
<td>Kim</td>
<td>15/5/1982</td>
<td>Engineer</td>
<td>2500</td>
<td>[1/3/2016, Now]</td>
</tr>
</tbody>
</table>
TSQL2 Modification Operations

- **Examples:**

  ```sql
  DELETE FROM Employee
  WHERE EmpName = 'Kim'
  VALID PERIOD '[2016-06-01 - FOREVER]'
  ```

<table>
<thead>
<tr>
<th>Employee</th>
<th>DateOfBirth</th>
<th>Job</th>
<th>Salary</th>
<th>VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kim</td>
<td>15/5/1982</td>
<td>Engineer</td>
<td>2500</td>
<td>[1/1/2016, 1/2/2016)</td>
</tr>
</tbody>
</table>
TSQL2 Modifications and Surrogates

- Surrogates are transparent time-invariant identifiers.
- Example:
  
  ```
  CREATE TABLE Supplier(ID SURROGATE, Name CHAR PRIMARY KEY, Address CHAR)
  AS VALID;
  INSERT INTO Supplier
  VALUES (NEW, 'Acme Inc.', 'New York')
  VALID PERIOD '[2014-01-01 - FOREVER]'
  ```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
<th>VALID</th>
</tr>
</thead>
</table>
TSQL2 Modifications and Surrogates

INSERT INTO Supplier
SELECT ID, 'New Acme Ltd.', Address
FROM Supplier
WHERE Name = 'Acme Inc.'
VALID PERIOD '[2016-01-01 - FOREVER]' 

or:  UPDATE Supplier
SET Name = 'New Acme Ltd.'
WHERE ID = ( SELECT ID FROM Supplier
WHERE Name = 'Acme Inc.' )
VALID PERIOD '[2016-01-01 - FOREVER]' 

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
<th>VALID</th>
</tr>
</thead>
</table>
TSQl2 Aggregate Functions

- Temporal grouping criteria:
  - Partition domain (valid or user-defined, instant or period)
  - Partition granularity
  - Associated time window (LEADING and TRAILING options)
  - Group belonging

- Example:

  SELECT Salary  
  FROM Employee AS Emp1  
  WHERE Emp1.EmpName = 'Tony'  
  AND VALID(Emp1) OVERLAPS  
  ( SELECT MIN(VALID(Emp2))  
    FROM Emp AS Emp2  
    WHERE Emp2.EmpName = 'Eve' )
TSQL2 Aggregate Functions

- Examples:

```sql
SELECT EmpName, SUM(WEIGHTED Salary)
FROM Employee(EmpName) AS Emp
GROUP BY VALID(Emp) USING '1' YEAR
HAVING MIN(Salary) > 2500

SELECT AVG(WEIGHTED Salary)
FROM Employee
WHERE EmpName = 'Tony'
GROUP BY VALID(Employee)
    USING '1' MONTH LEADING '11' MONTH
```
Calendars and Calendric Systems

- Calendars and calendric systems composed of multiple calendars are supported in TSQL2
- Ex. of calendars: Gregorian, Julian, Astronomic, Traditional_Chinese, US_Fiscal, UniBO_Academic
- Ex. of a calendric system: Russian (Roman till 100 B.C. then Julian till 1917, then Gregorian till 1929, then Communist till 1931 and then Gregorian again)

- Selection of a calendric system (Gregorian) in TSQL2:
  
  DECLARE CALENDRIC SYSTEM
  AS SQL92CALENDRICSYSTEM
Calendars and Calendric Systems

- Calendars are necessary for correct I/O and formatting of time data, that can be specified via the DATETIME_FORMAT property, ex.

```
SET PROPERTY FOR Italian_Calendar WITH VALUES ('DATETIME_FORMAT', '
  <DAY>/<MONTH>/<YEAR>  <HOUR>:<MINUTE>:<SECOND> ' )
```

then '19/02/2016 ' is a correct date literal for the Italian_Calendar

- Time zones and daylight saving are also supported, e.g. the following expressions are equivalent:

```
TIME '10:30:25' AT TIME ZONE INTERVAL '1' HOUR
TIME '10:30:25' AT TIME ZONE 'CET'
TIME '10:30:25+01:00'
```
Calendars and Calendric Systems

- Like in SQL-92, an EXTRACT() operator is also available to extract components from a temporal expression.
- Examples:

  ```sql
  EXTRACT (HOUR FROM TIME '01:27.30 PM')
  returns 13
  
  EXTRACT (MONTH FROM DATE 'June 7, 2010')
  returns 6
  
  EXTRACT (TIMEZONE_HOUR FROM TIMESTAMP '2015-05-13 13:27.30-4:00')
  returns -4
  ```
Temporal Indeterminacy

- Based on a probabilistic approach [Dyreson & Snodgrass]

- An indeterminate instant \( t = (t^- \sim t^+, P) \) is represented through:
  - Its lower \((t^-)\) and upper \((t^+)\) support
  - Its probability distribution \( P \) (null outside the support)

- Evaluation of selection predicates involving indeterminate instants (at a given plausibility level \( p \)) is based on the \textit{Before()} primitive:

\[
\text{Before}(p, t_1, t_2) := \neg (t_1 \equiv t_2) \land \Pr[t_1 < t_2] \geq p/100
\]

where the precedence probability is evaluated as:

\[
\Pr[t_1 < t_2] = \sum_{i<j} P_1(i)P_2(j)
\]
Temporal Indeterminacy

- The probability distribution can be STANDARD (i.e. UNIFORM or MISSING) or NONSTANDARD.

- Non standard distributions are user-defined point by point such that:
  \[ P(i) = 0 \text{ if } i < t^- \text{ or } i > t^+ \]
  \[ \sum_{t^- \leq i \leq t^+} P(i) = 1 \]

- Non standard distributions samples with predefined shapes could be provided by the system or made available by a DBA (e.g. PROBABLY_EARLY, PROBABLY_VERY_LATE, AROUND etc.)
Temporal Indeterminacy

Example:

CREATE TABLE Shipment( ParcelNo CHAR PRIMARY KEY, Destination CHAR, Arrival NONSTANDARD INDETERMINATE DATE )

INSERT INTO Shipment VALUES ('P102', 'Rome', '2016-02-20 ~ 2016-02-24' WITH DISTRIBUTION PROBABLY_EARLY)

SELECT * FROM Shipment WHERE Destination='Paris' AND VALID(Shipment) OVERLAPS DATE '2016-03-01' WITH PLAUSIBILITY '95'
Granularities in TSQL2

- Granularities are based on the lattice associated to a calendar
- TSQL2 extends the mechanism available in SQL-92 for the INTERVAL datatype, e.g.
  \texttt{INTERVAL\ DAY\ TO\ SECOND}
  (duration at a granularity between day and second)
- The upper granularity may be expressed as a range, e.g.
  \texttt{INTERVAL\ '1000'\ DAY\ TO\ SECOND}
- TSQL2 allows granularity definitions also for instant and period datatypes
- A precision specification can also be used, e.g.
  \texttt{TIME\ MINUTE(2)\ TO\ SECOND(3)}

The first is a range spec. ($10^2$ minutes) the second spec. is the maximum number of decimal digits ($10^{-3}$ seconds)
Granularities in TSQL2

- Comparison on operands with different granularities are effected at the granularity of the left operand
- Explicit granularity conversions are possible by means of the SCALE and CAST operators, e.g.
  - `SCALE(DATE '2010-01-01' AS MONTH)`
  - `CAST(DATE '2010-01-01' AS MONTH)`
    - both return 'January 2010'
  - `SCALE(DATE '2010-01-01' AS MINUTE)`
    - returns '2010-01-01 00:00 ~ 2010-01-01 23:59' (indeterm.)
  - `CAST(DATE '2010-01-01' AS MINUTE)`
    - returns '2010-01-01 00:00' (the first value at the finer gran.)
  - `SCALE(DATE 'March 2014 ~ April 2014' AS DAY)`
    - returns '2014-03-01 ~ 2014-04-30' (maximizes indet.)
  - `CAST(DATE 'March 2014 ~ April 2014' AS DAY)`
    - returns '2014-03-01 ~ 2014-04-01' (converts the supports)
The ATSQL Approach

- ATSQL [Böhlen, Jensen & Snodgrass] uses temporal statement modifiers to add temporal support to SQL
- Statement modifiers are semantic defaults that indicate “at each time point” without specifying how to compute it
- Provides a systematic way to construct temporal queries from non-temporal queries:
  1. Formulate the corresponding non-temporal query
  2. Apply a statement modifier
- Example: Temporal join
  - Formulate the non-temporal join
  - Modifier ensures that the argument timestamps overlap and that the result timestamp is the intersection of the argument periods
- ATSQL assumes period-timestamped tuples:
  - Periods have a meaning beyond a set of points
The ATSQL Approach

Example (temporal join):

SEQ VT
SELECT Department.*, Employee.Salary
FROM Employee, Department
WHERE EmpName = DeptManager
    AND Location = 'Miami'

The NSEQ VT ("nonsequenced valid time") modifier indicates that what follows should be treated as regular SQL, for example (tuple count):

NSEQ VT
SELECT COUNT(*) FROM Employee
The ATSQL Approach

- A query without a modifier considers only the present state of the argument relations (i.e. valid at NOW)

- Ensures that legacy queries on non-temporal relations are unaffected if the non-temporal relations are made temporal, e.g.

  ```sql
  SELECT * FROM Employee
  ```

- The modifiers mechanism is independent of the syntactic complexity of the queries

- The temporal parts are to a large degree separated from the non-temporal parts of the query

- The semantics of SQL extended with statement modifiers has been defined
TDB Support in SQL:2011

- The SQL/Temporal chapter was cancelled from the SQL3 definition in 2001 due to controversy within the ISO SQL committee (cf. ATSQL vs IXSQL approach)

- New temporal language extensions were recently submitted to and accepted by the ISO SQL committee as part of the SQL/Foundation Chapter of the new SQL:2011 standard

- The ability to create and manipulate temporal tables is the most important new feature in SQL:2011
TDB Support in SQL:2011

- Valid-time tables, dubbed as “Application-time period tables”, are supported
- Transaction-time tables, dubbed as “System-versioned tables”, are supported
- Bitemporal tables, dubbed as “System-versioned application-time period tables” (!), are supported
- Period timestamping is supported via 2 columns
- Temporal primary key and referential integrity constraints are supported
- Predicates are defined for querying along valid and transaction time
Application-time Period Tables

- Application-time period tables are tables that contain a PERIOD clause (newly-introduced) with a user-defined period name.
- Application-time period tables must contain two (user-defined) additional columns to store the start and end time of a period associated with the row.
- Values of both start and end columns are set by the users.
- Additional syntax is provided for users to specify primary key/unique constraints that ensure no two rows with the same key value have overlapping periods.
Creating an Application-time Period Table

CREATE TABLE Employee
(emp_name VARCHAR(50) NOT NULL PRIMARY KEY,
department_id VARCHAR(10),
start_date DATE NOT NULL, end_date DATE NOT NULL,
PERIOD FOR emp_period (start_date, end_date),
PRIMARY KEY (emp_name, emp_period WITHOUT OVERLAPS),
FOREIGN KEY (department_id, PERIOD emp_period) REFERENCES Department (department_id, PERIOD department_period))

- PERIOD clause automatically enforces the constraint end_date > start_date
- The name of the period can be any user-defined name
- The timestamping period is considered open to the right, i.e. [start_date, end_date)
Querying an Application-time Period Table

- Application-time period tables can be queried using the regular SQL syntax (temporal selection predicates can be expressed using comparison conditions over the timestamping columns)
- More user-friendly and Allen-complete period comparators (reminiscent of the TSQL2 ones) are also available:
  - CONTAINS, OVERLAPS, EQUALS, PRECEDES, SUCCEEDS, IMMEDIATELY PRECEDES, IMMEDIATELY SUCCEEDS

- Ex. SELECT * FROM Employee
  WHERE emp_period CONTAINS PERIOD '2015'

SELECT DISTINCT E1.emp_name, E2.emp_name
FROM Employee E1, E2
WHERE E1.emp_name < E2.emp_name
    AND E1.dept_id = E2.dept_id
    AND E1.emp_period OVERLAPS E2.emp_period
Modifying an Application-time Period Table

- Regular INSERT, UPDATE, DELETE statements can be used by explicitly managing values of conventional columns but also of the timestamping columns.
- A more user-friendly new FOR PORTION clause can be used to specify the applicability period of modifications.

Ex. UPDATE Employee
   FOR PORTION OF emp_period
   FROM DATE '2015-05-01' TO DATE '2015-06-01'
   SET dept_id = 'D5' WHERE emp_name = 'Tom'

DELETE Employee
   FOR PORTION OF emp_period
   FROM DATE '2016-03-01' TO DATE '9999-12-31'
   WHERE emp_name = 'Annabel'
System-versioned Tables

- System-versioned tables are tables that contain a PERIOD clause with a pre-defined period name (SYSTEM_TIME) and specify WITH SYSTEM VERSIONING.

- System-versioned tables must contain two additional (user-defined) columns to store the start and end time of the SYSTEM_TIME period.

- Values of both start and end columns are set by the system (users are not allowed to supply values).
System-versioned Tables

- Unlike regular tables, system-versioned tables preserve the old versions of rows as the table is updated.
- Rows whose periods intersect the current time are called current system rows. All others are called historical system rows.
- Only current system rows can be updated or deleted. System time applicability of modifications cannot be managed by the user.
- All constraints are enforced on current system rows only.
Creating a System-versioned Table

CREATE TABLE Employee
(emp_name VARCHAR(50) NOT NULL, dept_id VARCHAR(10),
system_start TIMESTAMP(6) GENERATED ALWAYS AS ROW START,
system_end TIMESTAMP(6) GENERATED ALWAYS AS ROW END,
PERIOD FOR SYSTEM_TIME (system_start, system_end),
PRIMARY KEY (emp_name),
FOREIGN KEY (dept_id) REFERENCES Department (dept_id);
) WITH SYSTEM VERSIONING

- Unlike regular tables, system-versioned tables preserve the old versions of rows as the table is updated.
- PERIOD clause automatically enforces the constraint system_end > system_start.
- The name of the period must be SYSTEM_TIME.
- The timestamping period is considered open to the right.
Querying a System-versioned Table

- The clause FOR SYSTEM_TIME can be used after the FROM clause to access past states of a table along transaction time (rollback queries)

- It comes with three variants:
  - FOR SYSTEM_TIME AS OF T (current at T)
  - FOR SYSTEM_TIME FROM T1 TO T2 (current in [T1,T2) )
  - FOR SYSTEM_TIME BETWEEN T1 AND T2 (current in [T1,T2] )

- Ex. SELECT * FROM Employee
  FOR SYSTEM_TIME
  FROM TIME '2011-01-01' TO TIME '2011-12-31'

SELECT * FROM Employee
FOR SYSTEM_TIME
AS OF TIMESTAMP '2014-04-01 12:30:00'
Creating a System-versioned Application-time Table

CREATE TABLE Employee
(emp_name VARCHAR(50) NOT NULL PRIMARY KEY,
department_id VARCHAR(10),
start_date DATE NOT NULL, end_date DATE NOT NULL,
system_start TIMESTAMP(6) GENERATED ALWAYS AS ROW START,
system_end TIMESTAMP(6) GENERATED ALWAYS AS ROW END,
PERIOD FOR emp_period (start_date, end_date),
PERIOD FOR SYSTEM_TIME (system_start, system_end),
PRIMARY KEY (emp_name, emp_period WITHOUT OVERLAPS),
FOREIGN KEY (department_id, PERIOD emp_period) REFERENCES Department (department_id, PERIOD department_period)
) WITH SYSTEM VERSIONING
CREATE TABLE Employee
(emp_name VARCHAR(50) NOT NULL PRIMARY KEY,
department_id VARCHAR(10),
FOREIGN KEY department_id REFERENCES Department
) AS VALID AND TRANSACTION

In practice, it is the same declaration done with regular SQL of a snapshot table Employee, simply augmented with the “AS VALID AND TRANSACTION” bitemporal specification (that implies the so deprecated syntactic and semantic defaults)