Temporal and Spatial Database
2014/2015 2nd semester

Temporal Extensions of SQL
SL04

- Comparative analysis
  - SQL + abstract data types: extend SQL with abstract data type
  - IXSQL: two operators to convert periods to points and vice versa
  - SQL/TP: only points exist; periods are internal and for display
  - TSQL2: syntactic defaults to easily formulate temporal statements
  - ATSQL: semantic defaults to easily formulate temporal statements

- Implementation of the sequenced semantics
  - RA + normalize, split, extend, absorb, scale

Acknowledgements: I am indebted to Michael Böhlen for providing me the slides.
Table of Contents

- Comparative analysis
  - SQL + abstract data types
  - IXSQL
  - SQL/TP
  - TSQL2
  - ATSQL

- Implementation of the sequenced semantics
  - Requirements
  - Algebraic primitives
  - Reduction rules
  - SQL implementation
  - Scaling
Temporal Query Languages

- A **temporal query language**, $QL$, is a part of a temporal data model: $DM = (DS, QL)$
  - Operates on the data structure $DS$
  - Define/create temporal relations
  - Insert/delete/modify information in a temporal relation
- Many temporal extensions of SQL have been proposed
- The goal is not to learn these languages but to understand and appreciate the concepts represented by these languages.
Language Design Criteria/1

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

- **Clarity**
  - Syntax should reflect the semantics to facilitate the formulation and understanding of queries
  - Consistent naming style

- **Consistency**
  - Upward compatibility with standards, e.g., SQL standard
  - Consistent syntax (e.g., not mix up postfix and prefix notation)
  - Systematic extension (not a new construct for each query)
Language Design Criteria/2

- **Minimality**
  - As few as possible new reserved words and clauses
  - No duplication of functionality

- **Orthogonality**
  - Possibility to freely combine query language constructs that are semantically independent
  - No exceptions
  - 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**:

  - $X \text{ before } Y$
  - $Y \text{ after } X$
  - $X \text{ equals } Y$
  - $X \text{ meets } Y$
  - $Y \text{ met_by } X$
  - $X \text{ overlaps } Y$
  - $Y \text{ overlapped_by } X$
  - $X \text{ during } Y$
  - $Y \text{ contains } X$
  - $X \text{ starts } Y$
  - $Y \text{ started_by } X$
  - $X \text{ finishes } Y$
  - $Y \text{ finished_by } X$
**Example**: Video store where customers \((\text{CustID})\) rent video tapes \((\text{TapeNum})\). The rentals are stored in a valid-time relation `Checkout`.

\[(\text{C101, T1234})\]  
\[(\text{C102, T1245})\]  
\[(\text{C102, T1234})\]  
\[(\text{C102, T1245})\]
Different queries on the **Checkout** table with their intended result:

**Q1:** *All checkouts that overlap time period [7, 9].*

<table>
<thead>
<tr>
<th>CustID</th>
<th>TapeNum</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C102</td>
<td>T1245</td>
<td>[5,7]</td>
</tr>
<tr>
<td>C102</td>
<td>T1234</td>
<td>[9,12]</td>
</tr>
</tbody>
</table>

**Q2:** *All 2-day checkouts.*

<table>
<thead>
<tr>
<th>TapeNum</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1245</td>
<td>[19,20]</td>
</tr>
<tr>
<td>T1245</td>
<td>[21,22]</td>
</tr>
</tbody>
</table>

**Q3:** *Which tapes were checked out together with tape T1234?*

<table>
<thead>
<tr>
<th>TapeNum</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1245</td>
<td>[5,5]</td>
</tr>
</tbody>
</table>
Video Rental Example/3

Q4: What is the number of tapes that have been checked out?

Q5: How many checkouts were made in total?

Q6: List all (current) checkouts (assume current time = 5).
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.

We assume **period timestamped** relations (or separate start and end points, which does not change much)
SQL + Abstract Data Types/2

Q1  SELECT  *  
    FROM  CheckOut  
    WHERE  T  OVERLAPS  [7,9]

Q2  SELECT  TapeNum, T  
    FROM  CheckOut  
    WHERE  DURATION(T) = 2

Q3  SELECT  b.TapeNum, INTERSECT(a.T, b.T) AS T  
    FROM  CheckOut AS a, CheckOut AS b  
    WHERE  a.TapeNum = T1234 AND b.TapeNum <> T1234  
    AND  a.T  OVERLAPS  b.T

Q4  too complicated

Q5  SELECT  COUNT(*)  FROM  CheckOut

Q6  SELECT  CustID, TapeNum  
    FROM  CheckOut  
    WHERE  T  OVERLAPS  [NOW]
Assessment of SQL + Abstract Data Types

- An abstract data type for periods is useful but the gain from having a period instead of start and end point is not big.
- An abstract data type does not offer a **systematic way to generalize** snapshot queries to temporal queries
  - Queries Q3 and Q4 both generalize a simple snapshot query to becoming time-varying.
  - The only difference is in the snapshot query, a join (Q3) versus an aggregation (Q4)
  - However, techniques used to formulate Q3 may not be re-used when formulating Q4 and vice versa.
  - Snapshot reducibility is poorly supported
- New and complex solutions must be invented (= programmed) for snapshot reducible queries.
IXSQL/1

- 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 *unfold*: decompose a period-timestamped tuple into a set of point-timestamped tuples (one for each point in the original period).
  - Function *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 formulation of the first 3 queries are essentially SQL

Unfold/fold mechanism is not required

Q1
SELECT *
FROM CheckOut
WHERE T CP [7, 9]

Q2
SELECT TapeNum, T
FROM CheckOut
WHERE DUR(T) = 2

Q3
SELECT b.TapeNum, INTERSECT(a.T, b.T) AS T
FROM CheckOut AS a, CheckOut AS b
WHERE a.TapeNum = T1234
AND b.TapeNum <> T1234
AND a.T CP b.T

<table>
<thead>
<tr>
<th>CustID</th>
<th>TapeNum</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C101</td>
<td>T1234</td>
<td>[3,5]</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>[5,7]</td>
</tr>
<tr>
<td>C102</td>
<td>T1234</td>
<td>[9,12]</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>[19,20]</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>[21,22]</td>
</tr>
</tbody>
</table>

IXSQL/2
Unfold can be used to formulate snapshot reducible queries.

Idea of fold and unfold:
convert to time points; answer query; convert to intervals

Q4 SELECT COUNT(*), T
  FROM ( SELECT *
          FROM CheckOut
          REFORMAT AS UNFOLD T )
  GROUP BY T
  REFORMAT AS FOLD T

Without unfold we get regular SQL without any special processing of the temporal information.

Q5 SELECT COUNT(*)
  FROM CheckOut

Q6 SELECT CustID, TapeNum
  FROM CheckOut
  WHERE T OVERLAPS [NOW]
Assessment of IXSQL

- Only two functions, fold and unfold, are added to SQL. This is attractive.
- Unfold can be used when needed to formulate snapshot reducible queries (it is optional and not an invasive change).
- **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)
    - Different results for different but snapshot-equivalent relations
  - Fold/unfold only preserve information of a point-based view
    - Normalization step using unfold/fold looses period information
    - Fold is not the inverse of unfold (information about the original periods is lost)
  - The combination of snapshot reducibility and periods is not supported (cf. temporal difference example for SQL/TP)
SQL/TP

- A **point-based** temporal extension of SQL
- Defined on temporal relations that use point timestamps
- Intervals do not exist at the logical level. They are used
  - internally (compact representation, efficient processing)
  - externally (presentation to users)
- Simple, unambiguous, and **well-defined semantics**
**Q1** must compare neighboring DB states

```
SELECT a.*
FROM CheckOut a, CheckOut b
WHERE a.SeqNo = b.SeqNo
AND (b.T = 7 OR b.T = 8 OR b.T = 9)
```

**Q2** can only be answered if `SeqNo` is present. Idea is to use aggregation.

```
SELECT SeqNo, TapeNum, T
FROM CheckOut
GROUP BY SeqNo
HAVING COUNT(T) = 2
```
SQL/TP

- Strength of SQL/TP is to extend snapshot queries to temporal queries if only a single state is involved. Idea is to extend snapshot queries with equality constraints on timestamps.


Q4  SELECT COUNT(*), T FROM CheckOut GROUP BY T
An additional attribute is needed for time-invariant aggregation.

The following fails for Q5:

- `SELECT COUNT(*) FROM CheckOut`

Q5: `SELECT COUNT(DISTINCT SeqNo) FROM CheckOut`

Q6: `SELECT CustID, TapeNum FROM CheckOut WHERE T = NOW`

<table>
<thead>
<tr>
<th>SeqNo</th>
<th>CustID</th>
<th>TapeNum</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C101</td>
<td>T1234</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>C101</td>
<td>T1234</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>C101</td>
<td>T1234</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>C102</td>
<td>T1245</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>C102</td>
<td>T1245</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>C102</td>
<td>T1245</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>C102</td>
<td>T1234</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>C102</td>
<td>T1234</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>C102</td>
<td>T1234</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>C102</td>
<td>T1234</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>C102</td>
<td>T1245</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>C102</td>
<td>T1245</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>C102</td>
<td>T1245</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>C102</td>
<td>T1245</td>
<td>22</td>
</tr>
</tbody>
</table>
Assessment of SQL/TP

- **Additional attribute needed** to preserve information about periods of the argument relations.

- Additional attribute is **not a systematic approach** to obtain point-based semantics **and** associate information with periods.

- Example: *When was tape T1245, but not tape T1234, checked out?*
  - Expected answer: \{[6, 7], [19, 20], [21, 22]\}
  - Plausible solution:

    ```sql
    SELECT T FROM CheckOut WHERE TapeNum = T1245
    EXCEPT
    SELECT T FROM Checkout WHERE TapeNum = T1234
    ```

    - Does not guarantee that (fragments of) individual checkout periods are returned because the sequence numbers are not included

    - Including the sequence number “disables” the difference
Period representation is still needed for user interaction and internal representation:

- What the user sees is periods (points are coalesced for presentations to the user).
- For formulating queries the user needs to (conceptually) map periods to points.

- Some timestamps, e.g., \((C101, T1234, [22, \infty])\) lead to problems (infinite relations).
Comprehensive extension of SQL2 to support time periods and time points.

Adopts syntactic defaults that make the formulation of common temporal queries more convenient.

A syntactic default is an abbreviation for a more complex expression. Instead of the abbreviation it is always possible to use the complex expression.

A default valid clause is implicitly placed after the select clause:

- Computes the intersection of the valid times of the argument relation(s) in the from clause
- Intersection is returned as timestamp in the result relation
Q1  SELECT  *
FROM  CheckOut
WHERE  VALID(T)  OVERLAPS  period  '7-9'

Q2  SELECT  SeqNo,  TapeNum,  T
FROM  CheckOut
WHERE  CAST(VALID(T)  AS  INTERVAL)  =  2

Result  in  Q2  must  include  SeqNo,  otherwise  value-equivalent  tuples  are  coalesced.

Q3  Implicit  default  valid  clause  in  Q3

-  valid  intersect(VALID(a),VALID(b))
-  i.e.,  non  intersecting  tuples  do  not  contribute  to  the  result

SELECT  b.TapeNum
FROM  CheckOut  AS  a,  CheckOut  AS  b
WHERE  a.TapeNum  =  T1234  AND  b.TapeNum  <>  T1234
Q4 not well defined (what should default valid clause be?!)

Q5 `SELECT SNAPSHOT COUNT(*)
FROM CheckOut`

The snapshot keyword is used to retrieve non-temporal relations (the default result is a temporal relation)

Q6 `SELECT SNAPSHOT *
FROM CheckOut
WHERE VALID(CheckOut) OVERLAPS [NOW]`
Assessment of TSQL2

- TSQL2 is a **large language** with many parts and an **informally defined semantics**.
- It is unclear what the syntactic defaults for Q4 should be (no formal definition).
- Syntactic defaults are **not scalable**:
  - A default is required for each language construct
  - Difficult to be comprehensive and systematic, and to ensure that defaults do not interact in unanticipated ways
- The language is **complex** and therefore difficult to understand and use
ATSQL introduces **temporal statement modifiers** to add temporal support to SQL.

Statement modifiers are **semantic defaults** that indicate snapshot reducible semantics without specifying how to compute it.

Provides a **systematic way** of constructing temporal queries from nontemporal queries:

1. Formulate the corresponding nontemporal query
2. Apply a statement modifier

**Example: Temporal join**

- Formulate the nontemporal 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:**

- Value-equivalent tuples with adjacent or overlapping periods are permitted
- Periods have a meaning beyond a set of points
The `seq vt` ("sequenced valid time") modifier ensures that timestamps are returned as if the enclosed statement was evaluated on each state of the database. In addition interval boundaries (change points) are preserved.

The rest of the query is standard SQL.

**Q1**
```
SEQ VT
SELECT *
FROM CheckOut
WHERE T OVERLAPS [7, 9]
```

**Q2**
```
SEQ VT
SELECT TapeNum
FROM CheckOut
WHERE DURATION(T) = 2
```
Snapshot reducibility is easy to specify and the specification is independent of the query that shall be evaluated with snapshot reducible semantics.

Q3  seq vt
    select b.TapeNum
    from CheckOut as a, CheckOut as b
    where a.TapeNum = T1234
    and b.TapeNum <> T1234

Q4  seq vt
    select count(*) from CheckOut
The \texttt{nseq vt} ("nonsequenced valid time") modifier indicates that what follows should be treated as regular SQL.

The rest of the query is standard SQL.

\textbf{Q5} \texttt{NSEQ VT}
\texttt{SELECT COUNT(*)}
\texttt{FROM CheckOut}

A query without a modifier considers only the current state of the argument relations (i.e., NOW)

Ensures that legacy queries on nontemporal relations are unaffected if the nontemporal relations are made temporal.

\textbf{Q6} \texttt{SELECT *}
\texttt{FROM CheckOut}
Assessment of ATSQL

- Language mechanism is **independent of the syntactic complexity** of the queries.
- The temporal parts are to a large degree separated from the nontemporal parts of the query.
- The semantics of SQL extended with statement modifiers has been defined.
- Represents a **more fundamental change** to the language design than other approaches, e.g., abstract data types.
Table of Contents

- Comparative analysis
  - SQL + abstract data types
  - IXSQL
  - SQL/TP
  - TSQL2
  - ATSQL

- Integration of temporal support in DBMS kernel
  - Requirements
  - Algebraic primitives
  - Reduction rules
  - SQL implementation
  - Scaling
Integration of Temporal Support into DBMS Kernel

Goal:
- Determine and implement in the DBMS kernel the functionality that is required to offer comprehensive temporal support.

Solution:
- Three requirements that define “comprehensive temporal support”.
- Two algebraic primitives that split periods into pieces, such that snapshot reducible queries only need equality for comparison.
- The algebraic primitives satisfy the three requirements.
- The combination of the three requirements is called sequenced semantics.
R1. Snapshot Reducibility (SR)

\( \varphi^T \) is snapshot reducible iff

\[
\forall t(\tau_t(\psi^T(R_1, \ldots, R_n)) \equiv \psi(\tau_t(R_1), \ldots, \tau_t(R_n)))
\]

For each non-temporal operator there is a snapshot reducible operator:

- count at each point in time
- join at each point in time
- primary key at each point in time
- ranking at each point in time

Snapshot reducibility (SR) ensures a systematic generalization of all non-temporal operators to temporal operators.

- \( D^\theta \sum \ldots \Rightarrow D^{\theta^T} \sum \ldots \)
- \( \bowtie \theta \ldots \Rightarrow \bowtie^{\theta^T} \ldots \)
R2. Extended Snapshot Reducibility (ESR)

\( \psi^T \) is extended snapshot reducible iff

\[
\forall t(\tau_t(\psi^T(R_1, \ldots, R_n)) \equiv \pi_E(\psi(\tau_t(\epsilon U_1(R_1)), \ldots, \tau_t(\epsilon U_n(R_n)))))
\]

where \( \psi \in \{\vartheta, \sigma, \pi, \times, \sqcap, \sqcap, \sqcup, \sqcap, \triangleright \} \), \( E = \) schema of \( \psi^T(R_1, \ldots, R_n) \)

We can access time intervals in snapshot reducible operators.

- at each point in time do a join of long and short projects

Extended snapshot reducibility (ESR) makes it possible to get snapshot reducibility and to access timestamps (by propagating timestamps with \( \epsilon \)).

- at each point in time the average duration of externally funded projects: \( D^T_{AVG}(DUR(T)) \)
R3. Change Preservation (CP)

$\psi^T$ is change preserving iff $\forall z, z' \in \psi^T(R_1, \ldots, R_n)($

$$\forall t, t' \in z. T(L(z, t) = L(z, t')) \land$$

$$(z.T_S - 1 \in z'. T \land z.A = z'. A \Rightarrow L(z', z.T_S - 1) \neq L(z, z.T_S)) \land$$

$$(z.T_E \in z'. T \land z.A = z'. A \Rightarrow L(z', z.T_E) \neq L(z, z.T_S)))$$

We use the lineage to preserve changes (start and end of intervals).

- $L(z, t)$ is the lineage of result tuple $z$ at time point $t$
- $\{\text{(DB, 8k, [Feb, Jul])}\} \neq \{\text{(DB, 8k, [Feb, Apr]), (DB, 8k, [Apr, Jul])}\}$
- Scaling of values (based on old and new timestamps) gets possible

Change preservation (CP) allows facts that hold for an entire interval but not a subinterval.

- $D^\psi_T \text{SUM}(\text{scale}(B))$
To implement the sequenced semantics **two new algebra operators** (primitives) for the **adjustment** of periods are needed:
- Temporal normalization $\mathcal{N}$
- Temporal alignment $\phi$

- Adjustment respects the lineage.

- **Reduction rules** from temporal RA to nontemporal RA.

- **Period propagation** $\epsilon$ is possible and makes it possible to access the original periods.
Temporal Primitives/2

- The purpose of a temporal primitive is to break periods into pieces.

- Two temporal primitives are required since there are two classes of operators in relational DBMSs:
  - One input tuple contributes to at most one result tuple per time point.  
    ⇒ temporal normalization  
    Example: Aggregation
  - One input tuple contributes to more than one result tuple per time point.  
    ⇒ temporal alignment  
    Example: Joins
Temporal Normalization

- Average duration of contracts per department: \( D^{\mathcal{T}}_{\text{AVG}}(DUR(T))(R) \)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( D )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>DB</td>
<td>[Feb, Jul]</td>
</tr>
<tr>
<td>Ann</td>
<td>DB</td>
<td>[Feb, Jul]</td>
</tr>
<tr>
<td>Sam</td>
<td>AI</td>
<td>[May, Oct]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( N )</th>
<th>( D )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>6</td>
<td>DB</td>
</tr>
<tr>
<td>AVG</td>
<td>7</td>
<td>DB</td>
</tr>
<tr>
<td>AVG</td>
<td>5</td>
<td>AI</td>
</tr>
</tbody>
</table>

- One input tuple contributes to at most one result tuple per month.
Temporal Alignment

- Employees managed by manager: $M \triangleright^T_{M.D=R.D} R$

![Diagram of temporal alignment]

- One input tuple contributes to more than one result tuple per month. E.g., $m_1$ contributes twice to month Feb.
## Reduction Rules

<table>
<thead>
<tr>
<th>Operator</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$\sigma^T_\theta(R) = \sigma_\theta(R)$</td>
</tr>
<tr>
<td>Projection</td>
<td>$\pi^T_B(R) = \pi_{B,T}(\mathcal{N}_B(R, R))$</td>
</tr>
<tr>
<td>Aggregation</td>
<td>$\mathcal{B}^T_\vartheta(R) = \mathcal{B}_{T \vartheta}(\mathcal{N}_B(R, R))$</td>
</tr>
<tr>
<td>Difference</td>
<td>$R \mathcal{A}^T_S = \mathcal{N}_A(R, S) \mathcal{A}^T_S(S, R)$</td>
</tr>
<tr>
<td>Union</td>
<td>$R \cup^T_S = \mathcal{N}_A(R, S) \cup \mathcal{N}_A(S, R)$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$R \cap^T_S = \mathcal{N}_A(R, S) \cap \mathcal{N}_A(S, R)$</td>
</tr>
<tr>
<td>Cartesian Prod.</td>
<td>$R \times^T_S = \alpha(\phi^T(R, S) \times_{R.T=S.T} \Phi^T(S, R))$</td>
</tr>
<tr>
<td>Inner Join</td>
<td>$R \times^T_\theta S = \alpha(\phi_\theta(R, S) \times_{\theta \land R.T=S.T} \Phi_\theta(S, R))$</td>
</tr>
<tr>
<td>Left O. Join</td>
<td>$R \times^T_\theta S = \alpha(\phi_\theta(R, S) \times_{\theta \land R.T=S.T} \Phi_\theta(S, R))$</td>
</tr>
<tr>
<td>Right O. Join</td>
<td>$R \times^T_\theta S = \alpha(\phi_\theta(R, S) \times_{\theta \land R.T=S.T} \Phi_\theta(S, R))$</td>
</tr>
<tr>
<td>Full O. Join</td>
<td>$R \times^T_\theta S = \alpha(\phi_\theta(R, S) \times_{\theta \land R.T=S.T} \Phi_\theta(S, R))$</td>
</tr>
<tr>
<td>Anti Join</td>
<td>$R \triangleright^T_\theta S = \phi_\theta(R, S) \triangleright_{\theta \land R.T=S.T} \Phi_\theta(S, R)$</td>
</tr>
</tbody>
</table>

\(\alpha \ldots\) temporal duplicate elimination.
Constructing Sequenced Algebra Expressions

Query: $D^{\vartheta}_{AVG}(DUR(T))(R)$

1. Timestamp propagation:
   $D^{\vartheta}_{AVG}(DUR(T))(\epsilon_U(R))$

2. Timestamp substitution:
   $D^{\vartheta}_{AVG}(DUR(U))(\epsilon_U(R))$

3. Temporal adjustment:
   $\mathcal{N}_D(\epsilon_U(R), \epsilon_U(R))$

4. Nontemporal aggregation:
   $D,T^{\vartheta}_{AVG}(DUR(U))(\mathcal{N}_D(\epsilon_U(R), \epsilon_U(R)))$
DBMS kernel integration of temporal primitives.
SQL extension that provides direct access to primitive operators:

- $\epsilon_U(R)$: \texttt{SELECT Ts Us, Te Ue, * FROM R}
- $\mathcal{N}_B(R, S)$: \texttt{FROM (R \text{ NORMALIZE} S \text{ USING} (B)) R}
- $\phi_\theta(R, S)$: \texttt{FROM (R \text{ ALIGN} S \text{ ON} \theta) R}
- $\alpha(R)$: \texttt{SELECT ABSORB * FROM R}

Temporal SQL languages can be implemented in Parser/Analyzer.

Source Code:
http://www.ifi.uzh.ch/dbtg/research/align.html
Algebraic Basis for Sequenced Semantics

- Reduction is at algebra level.
  ⇒ Any existing language supporting sequenced semantics can be implemented.

Diagram:

- RA + $\mathcal{N}$ + $\phi$
- ATSQL
- SQL/Temporal
- IXSQL
- TSQL2
- SQL/TP
- DBMS
Timestamp Propagation in SQL

- **Extend** adds to the schema of \( R \) attribute \( U \) that is a duplicate of the period of \( R \).

\[
R = \begin{array}{cc}
N & T \\
Ann & [1, 8) \\
Joe & [2, 6) \\
Ann & [8, 12) \\
\end{array}
\]

\[
\epsilon_U(r) = \begin{array}{ccc}
N & U & T \\
Ann & [1, 8) & [1, 8) \\
Joe & [2, 6) & [2, 6) \\
Ann & [8, 12) & [8, 12) \\
\end{array}
\]

- Algebra: \( \epsilon_U(R) \)

- SQL:

```
WITH
  R AS (SELECT N, T AS U, T FROM R)

... 
```
Absorb in SQL

- **Absorb** eliminates from \( r \) temporal duplicates, i.e., tuples with a period that is contained in the period of a value-equivalent tuple.

\[
R
\begin{array}{ccc}
A & B & T \\
\hline
a & c & [1, 9) \\
a & c & [3, 7) \\
a & d & [3, 7) \\
b & c & [3, 7) \\
b & d & [3, 7)
\end{array}
\]

\[
\alpha(R)
\begin{array}{ccc}
A & B & T \\
\hline
a & c & [1, 9) \\
a & d & [3, 7) \\
b & c & [3, 7) \\
b & d & [3, 7)
\end{array}
\]

- **Algebra:** \( \alpha(R) \)
- **SQL:**

```
SELECT ABSORB ...
```
Temporal Normalization in SQL

- **Normalize** splits each tuple in $R$ with respect to the group of tuples in $S$ that match on the grouping attributes $B$.

  ![Diagram](image)

- Algebra: $\mathcal{N}_B(R, S)$
- SQL:

  ```sql
  ... FROM (R NORMALIZE S USING(B)) AS Rnorm ...
  ```
Temporal Alignment in SQL

- **Align** splits each tuple in $R$ with respect to each tuple in the group of tuples in $S$ that satisfy $\theta$.

- Algebra: $\phi_{\theta}(R, S)$
- SQL:

  ```sql
  FROM ( R ALIGN S ON theta ) AS R
  ```
Answering Sequenced Queries

Approach for formulating sequenced queries:

- Formulate query without thinking about time/periods; make all operators temporal (e.g., $\vartheta^T$).
  
  $\vartheta^T_{\text{AVG}(\text{DUR}(T))}(R)$

- Add copies of periods for periods that are used in conditions.
  
  $\vartheta^T_{\text{AVG}(\text{DUR}(T))}(\epsilon U(R))$

- Replace all references to periods with references to these copies.
  
  $\vartheta^T_{\text{AVG}(\text{DUR}(U))}(\epsilon U(R))$

- Apply reduction rules to get nontemporal relational algebra expression.
  
  $\mathcal{T} \vartheta_{\text{AVG}(\text{DUR}(U))}(\mathcal{N}(\epsilon U(R), \epsilon U(R)))$
Example/1

\[ R \]

\[ r_1 = (\text{Ann}) \]
\[ r_2 = (\text{Joe}) \]
\[ r_3 = (\text{Ann}) \]

\[ \epsilon_U(R) \]

\[ r_1 = (\text{Ann}, [1, 8]) \]
\[ r_2 = (\text{Joe}, [2, 6]) \]
\[ r_3 = (\text{Ann}, [8, 12]) \]

\[ \mathcal{N}(\epsilon_U(R), \epsilon_U(R)) \]

\[ (\text{Ann}, [1, 8]) \]
\[ (\text{Ann}, [1, 8]) \]
\[ (\text{Joe}, [2, 6]) \]
\[ (\text{Ann}, [8, 12]) \]

\[ T^{\vartheta_{AVG(DUR(U))}}(\mathcal{N}(\epsilon_U(R), \epsilon_U(R))) \]

\[ (7) \]
\[ (5.5) \]
\[ (7) \]
\[ (4) \]
Translate resulting algebra expression to SQL:

\[ T^n_{AVG(DUR(U))}(N(\epsilon_U(R), \epsilon_U(R))) \]

- Schema: R(N, Ts, Te)

```
WITH x AS (SELECT Ts Us, Te Ue, * FROM R )
SELECT AVG(Ue - Us), Ts, Te
FROM (x NORMALIZE x USING()) Radj
GROUP BY Ts, Te;
```
Scaling/1

Consider a project relation $p$ that keeps track of project funding (funding amount $B$)

\[
p
\]

<table>
<thead>
<tr>
<th>$D$</th>
<th>$N$</th>
<th>$B$</th>
<th>$T_S$</th>
<th>$T_E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>1</td>
<td>181K</td>
<td>2013/2/1</td>
<td>2013/8/1</td>
</tr>
<tr>
<td>DB</td>
<td>2</td>
<td>196K</td>
<td>2013/5/1</td>
<td>2014/1/1</td>
</tr>
<tr>
<td>AI</td>
<td>1</td>
<td>153K</td>
<td>2013/4/1</td>
<td>2013/9/1</td>
</tr>
<tr>
<td>AI</td>
<td>2</td>
<td>120K</td>
<td>2013/4/1</td>
<td>2013/9/1</td>
</tr>
</tbody>
</table>

When aggregating such data according to snapshot reducibility we usually want to scale the funding to the adjusted time periods.

The result should be as follows:

\[
r
\]

<table>
<thead>
<tr>
<th>$D$</th>
<th>$B$</th>
<th>$T_S$</th>
<th>$T_E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>89K</td>
<td>2013/2/1</td>
<td>2013/5/1</td>
</tr>
<tr>
<td>DB</td>
<td>165.6K</td>
<td>2013/5/1</td>
<td>2013/8/1</td>
</tr>
<tr>
<td>DB</td>
<td>122.4K</td>
<td>2013/8/1</td>
<td>2014/1/1</td>
</tr>
<tr>
<td>AI</td>
<td>273K</td>
<td>2013/4/1</td>
<td>2013/9/1</td>
</tr>
</tbody>
</table>
Scaled values are functions of the original value, the old period, and the new period.

The following function is a simple user-defined function that uniformly scales values.

```
CREATE OR REPLACE FUNCTION scale(x FLOAT, ts_new DATE, te_new DATE,
                                    ts_old DATE, te_old DATE)
RETURNS FLOAT AS
BEGIN
    RETURN x * (te_new - ts_new) / (te_old - ts_old);
END;
LANGUAGE PLPGSQL;
```
Scaling/3

Procedure for scaling with normalize (old periods cannot be propagated through nontemporal operators)

1. propagate periods, $\epsilon$;
2. normalize, $N_{\theta}$ (possibly scale values);
3. possibly scale values and remove propagated attributes;
4. apply the corresponding nontemporal operator, $\psi$.

Procedure for scaling with adjust (old periods can be propagated through nontemporal operators)

1. propagate periods, $\epsilon$;
2. align, $\phi_{\theta}$ (possibly scale values);
3. apply nontemporal operator, $\psi$ (possibly scale values);
4. possibly scale values and remove propagated attributes.
Query: Determine amount of external funding per department.

\[ D \vartheta^T \text{SUM}(\text{scale}(B))(p) \]

\[ \Rightarrow D \vartheta^T \text{SUM}(\text{scale}(B))(\epsilon U(p)) \]

\[ \Rightarrow D, T \vartheta^T \text{SUM}(\text{scale}(B))(\mathcal{N}_D(\epsilon U(p), \epsilon U(p))) \]

\[
\begin{align*}
\text{WITH} \quad & \\
& P1 \text{ AS } (\text{SELECT } Ts \text{ Us, } Te \text{ Ue, } *, \text{ FROM } P), \\
& P2 \text{ AS } (\text{SELECT } *, \text{ FROM } (P1 \times \text{NORMALIZE } P1 \text{ y ON x.D=y.D}) \text{ P}), \\
& P3 \text{ AS } (\text{SELECT } D, N, \text{ scaleU}(B, Ts, Te, Us, Ue) B, Ts, Te \text{ FROM } P2)
\end{align*}
\]

\[
\begin{align*}
& \text{SELECT } D, \text{ SUM}(B), Ts, Te \\
& \text{FROM } \quad P3 \\
& \text{GROUP BY } D, Ts, Te;
\end{align*}
\]
Language design criteria: expressive power, minimality, orthogonality, closedness, clarity, consistency

Comparison of periods: convenient but not a major issue

Concrete SQL-based query languages

- **SQL + abstract data types**: extend SQL with abstract data type; gain is limited
- **IXSQL**: two operators to convert periods to points and vice versa; simple and attractive; issues with performance and semantics
- **SQL/TP**: only points exist; periods are used internally and for display; based on points, which gives simple semantics; issue with infinity, and presentation
- **TSQL2**: use syntactic defaults to easily formulate temporal statements; comprehensive language; no precise semantics
- **ATSQL**: use semantic defaults to easily formulate temporal statements; adds modifiers to SQL, which is a significant change
New algebraic primitives in the DBMS kernel that provide support for the sequenced semantics.

- normalize
- align
- extend
- absorb

Direct mapping of the primitives to SQL for illustration purposes. Other SQL extensions are possible.

Change point preservation makes it possible to scale attribute values to fit adjusted periods.