Temporal and Spatial Data Management
Fall 2017

Temporal Data Models
SL03

- Modeling temporal data
- Temporal data models
- Temporal query languages
Table of Contents

- Modeling temporal data
  - Dimensions of time
  - Timestamping data
- Temporal data models
  - Snodgrass’ tuple timestamping model
  - Jensen’s backlog model
  - Ben-Zvi’s tuple timestamped model
  - Gadia’s attribute value timestamped model
- Comparative analysis of temporal query languages
  - SQL + abstract data types
  - IXSQL
  - SQL/TP
  - TSQL2
  - ATSQL
Temporal Data Models/1

- **Data model:** \( M = (DS, QL) \)
  - \( DS \) is a set of data structures
  - \( QL \) is a language for querying and updating the data structures
- Example: relational data model is composed of relations and SQL
- Many extensions of the relational data model to support time have been proposed

- Several modeling aspects have to be considered
  - Different time dimensions
  - Different timestamp types
  - Attribute versus tuple timestamping
Temporal Data Models/2

The different modeling aspects lead to **subtle** and **difficult** issues. There are pros and cons in all cases. No single choice seems best and after more than 20 years no consensus has been reached!

- Which time dimensions do we want?
  - Transaction time, valid time, decision time, ...

- Which timestamps shall we use?
  - points, periods, sets of points, ...

- Which information shall be timestamped?
  - objects, facts, part of objects, ...
Dimensions of Time/1

- Time is **multi-dimensional**
  - Valid time
  - Transaction time
  - Publication time
  - Efficacy time
  - Decision time
  - etc.

- The key question: Which time aspects are **sufficiently important** so that they should be supported by the database system?

- There is a broad consensus that **transaction time** and **valid time** are the most important time dimensions.

- Transaction time is about to be incorporated into commercial database systems (law requires accountability, etc).
Dimensions of Time/2

- **Valid time**: The time a fact was/is/will be true in the modeled reality/mini-world
  - A fact is a statement that is either true or false
  - A relation is a collection of facts
  - Example: John has been hired on October 1, 2004
  - Valid time captures the time-varying states of the mini-world
  - All facts have a valid time by definition, however, it might not be recorded in the database
  - Valid time is independent of the recording of the fact in a database
  - Valid time is either bounded (does not extend until infinity) or unbounded (extends until infinity)
Dimensions of Time/3

- **Transaction time**: The time when a fact is current/present in the database as stored data
  - Example: the fact “John was hired on October 1, 2004” was stored in the DB on October 5, 2004, and has been deleted on March 31, 2005
  - Transaction time has a duration: from insertion to deletion, with multiple insertions and deletions being possible for the same fact
  - With transaction time deletions of facts are purely logical
    - the fact remains in the database, but ceases to be part of the database’s current state.
  - Transaction time captures the time-varying states of the database
  - Always bounded on both ends
    - Starts when the database is created (nothing was stored before)
    - Does not extend past now (no facts are known to have been stored in the future)
  - Basis for supporting accountability and “traceability” requirements, e.g., in financial, medical, legal applications.
  - Should be supplied automatically by the DBMS
A data model can support none, one, two, or more of these time dimensions:

- **Snapshot** data model: None of the time dimensions is supported. Represents a single snapshot of the reality and the database.
- **Valid time** data model: Supports only valid time.
- **Transaction time** data model: Supports only transaction time.
- **Bitemporal** data model: Supports valid time and transaction time.

An important issue is what shall be the basic element of a temporal data model:

- objects, relations, classes, tuples, etc
- closure is important
A timestamp is a value that is associated with data in a database
- Captures some temporal aspect, e.g., valid time, transaction time
- Represented as one or more attributes/columns of a relation

Three different types of timestamps are widely used
- Time points
- Time periods
- Temporal elements

Two different ways of timestamping
- Tuple timestamping
- Attribute timestamping
Example: Video store where customers, identified by a CustID, rent video tapes, identified by a TapeNum. Consider the following rentals during May 1997:

- On 3rd of May, customer C101 rents tape T1234 for three days
- On 5th of May, customer C102 rents tape T1245 for 3 days
- From 9th to 12th of May, customer C102 rents tape T1234
- From 19th to 20th of May, and again from 21st to 22nd of May, customer C102 rents tape T1245

These rentals are stored in a relation CheckOut which is graphically illustrated below:

```
(C101,T1234)  (C101,T1245)
(C102,T1245)  (C102,T1234)  (C102,T1245)
```

(TSDM17, SL03 10/66 M. Böhlen, ifi@uzh)
Point-based data model: each tuple is timestamped with a time point/instant.

- Most basic and simple data model.
- Timestamps are atomic values that can be easily compared, using $=, \neq, <, >, \geq, \leq$.
- Multiple tuples are used if a fact is valid at several time points.
- Syntactically different relations store different information.
- Provides an abstract view of a database and is not meant for physical implementation.
- Conceptual simplicity and computational complexity make it popular for theoretical studies.

<table>
<thead>
<tr>
<th>CustID</th>
<th>TapeNum</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C101</td>
<td>T1234</td>
<td>3</td>
</tr>
<tr>
<td>C101</td>
<td>T1234</td>
<td>4</td>
</tr>
<tr>
<td>C101</td>
<td>T1234</td>
<td>5</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>5</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>6</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>7</td>
</tr>
<tr>
<td>C102</td>
<td>T1234</td>
<td>9</td>
</tr>
<tr>
<td>C102</td>
<td>T1234</td>
<td>10</td>
</tr>
<tr>
<td>C102</td>
<td>T1234</td>
<td>11</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>12</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>19</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>20</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>21</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
<td>22</td>
</tr>
</tbody>
</table>
The reconstruction of the original relation is not always possible.

The table on the previous slide makes it impossible to determine if C102 rented T1245 once or twice in the period from 19 to 22.

Additional attributes are required, e.g., SeqNo to represent the 2-day rentals.

It is difficult to predict when an additional attribute is needed.

<table>
<thead>
<tr>
<th>SeqNo</th>
<th>CustID</th>
<th>TapeNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C101</td>
<td>T1234</td>
</tr>
<tr>
<td>1</td>
<td>C101</td>
<td>T1234</td>
</tr>
<tr>
<td>1</td>
<td>C101</td>
<td>T1234</td>
</tr>
<tr>
<td>2</td>
<td>C102</td>
<td>T1245</td>
</tr>
<tr>
<td>2</td>
<td>C102</td>
<td>T1245</td>
</tr>
<tr>
<td>2</td>
<td>C102</td>
<td>T1245</td>
</tr>
<tr>
<td>3</td>
<td>C102</td>
<td>T1234</td>
</tr>
<tr>
<td>3</td>
<td>C102</td>
<td>T1234</td>
</tr>
<tr>
<td>3</td>
<td>C102</td>
<td>T1234</td>
</tr>
<tr>
<td>4</td>
<td>C102</td>
<td>T1245</td>
</tr>
<tr>
<td>4</td>
<td>C102</td>
<td>T1245</td>
</tr>
<tr>
<td>5</td>
<td>C102</td>
<td>T1245</td>
</tr>
<tr>
<td>5</td>
<td>C102</td>
<td>T1245</td>
</tr>
</tbody>
</table>

TSDM17, SL03 12/66 M. Böhlen, ifi@uzh
**Tuple Timestamping with Periods/1**

- **Period-based data model** (usually the term interval-based is used): each tuple is timestamped with a **time period**.

<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>

- Timestamps are atomic values that can be compared using Allen’s 13 basic relationships between periods (before, meets, during, etc.).
  - More convenient than comparing the endpoints of the periods.
  - The benefits of Allen’s predicates are relatively small.
The start and end of an interval are distinguished change points.

The $\text{SeqNo}$ attribute is not needed to distinguish different checkouts.

Multiple tuples are used if a fact is valid over disjoint time periods.

Cannot model a single checkout with a gap.

The most popular model from an implementation perspective.

Time periods are not closed under all set operations.

Example: subtracting $[5, 7]$ from $[1, 9]$ returns a set of periods $\{[1, 4], [8, 9]\}$. 
Data models with temporal elements: each tuple is timestamped with a temporal element, i.e., a finite union of periods.

<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], [19, 22]}</td>
</tr>
<tr>
<td>C102</td>
<td>T1234</td>
<td>{[9, 12]}</td>
</tr>
</tbody>
</table>

The full history of a fact is stored in one tuple.

Usually the periods of a temporal element must be disjoint and non-adjacent. This makes it similar to point timestamps.
Attribute Timestamping/1

- **Attribute value timestamping**: each attribute value is timestamped with a set of time points/periods
- **Goal**: capture all information about a real-world object in a single tuple
  - e.g., all information about a customer in the relation below

<table>
<thead>
<tr>
<th>SeqNo</th>
<th>CustID</th>
<th>TapeNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>{[3, 5]}</td>
<td>{[3, 5]}</td>
<td>{[3, 5]}</td>
</tr>
<tr>
<td>{[5, 7]}</td>
<td>{[5, 7], [9, 12], [19, 22]}</td>
<td>{[5, 7], [19, 22]}</td>
</tr>
<tr>
<td>{[9, 12]}</td>
<td>{[9, 12]}</td>
<td>{[9, 12]}</td>
</tr>
<tr>
<td>{[19, 20]}</td>
<td>{[19, 20]}</td>
<td></td>
</tr>
<tr>
<td>{[21, 22]}</td>
<td>{[21, 22]}</td>
<td></td>
</tr>
</tbody>
</table>

- A single tuple may record multiple facts
  - e.g., the second tuple records the following facts: rental information for customer C102 for the tapes T1245 and T1234, and four different checkouts
- **Non-first-normal-form data model**
Different groupings of the information into tuples are possible for attribute-value timestamping.

Information about other objects is spread across several tuples, e.g., information about tapes.

E.g., grouping on tape number in the example below:

<table>
<thead>
<tr>
<th>SeqNo</th>
<th>CustID</th>
<th>TapeNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>{[3, 5]}</td>
<td>{[3, 5]}</td>
<td>C101, {[3, 5], [9, 12]}</td>
</tr>
<tr>
<td>{[9, 12]}</td>
<td>{[9, 12]}</td>
<td>C102</td>
</tr>
<tr>
<td>{[5, 7]}</td>
<td>{[5, 7], [19, 22]}</td>
<td>C102, {[5, 7], [19, 22]}</td>
</tr>
<tr>
<td>{[19, 20]}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{[21, 22]}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

- Modeling temporal data
  - Dimensions of time
  - Timestamping data

- Temporal data models
  - Snodgrass’ tuple timestamping model
  - Jensen’s backlog model
  - Ben-Zvi’s tuple timestamped model
  - Gadia’s attribute value timestamped model

- Comparative analysis
  - SQL + abstract data types
  - IXSQL
  - SQL/TP
  - TSQL2
  - ATSQL
Snodgrass’ Tuple Timestamped Model

- **Snodgrass’ tuple timestamped data model**
  - Supports valid time and transaction time
  - Adds four *atomic-valued attributes* to each relation
    - Start and end point of the valid time \((V_s, V_e)\)
    - Start and end point of the transaction time \((T_s, T_e)\)
  - Schema: \(R = (A_1, \ldots, A_n, T_s, T_e, V_s, V_e)\)
  - \(T_s, T_e, V_s, V_e\) represent the bitemporal chronons (a bitemporal chronon is a two-dimensional time point) of the corresponding *rectangular* region
  - 1NF relations
A closed region in a two dimensional space (TT x VT) must be represented by a set of rectangles.

- any bitemporal chronon in x. T is contained in at least one rectangle
- each bitemporal chronon in a rectangle is contained in x. T

Various coverings of a 2D area are possible:

- Overlapping versus non-overlapping rectangles
- Partitioning by transaction time versus partitioning by valid time
Example: Department relation in the Snodgrass data model, using partitioning by transaction time.

Partitioning by transaction time yields maximal segments in valid time direction.

Partitioning by valid time yields maximal segments in transaction time direction.
The relation instance (we assume Snodgrass tuple timestamped model) does not change as time passes by.

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
<th>$T_s$</th>
<th>$T_e$</th>
<th>$V_s$</th>
<th>$V_e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>10</td>
<td>14</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>15</td>
<td>19</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Jake</td>
<td>Load</td>
<td>20</td>
<td>now</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Kate</td>
<td>Ship</td>
<td>20</td>
<td>now</td>
<td>25</td>
<td>30</td>
</tr>
</tbody>
</table>
### The Timestamp NOW/2

<table>
<thead>
<tr>
<th>E-ID</th>
<th>EName</th>
<th>Salary</th>
<th>Dept</th>
<th>TT</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jane</td>
<td>40</td>
<td>Advtg</td>
<td>5-now</td>
<td>5-now</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>20</td>
<td>Sales</td>
<td>5-now</td>
<td>3-now</td>
</tr>
<tr>
<td>1</td>
<td>Jane</td>
<td>30</td>
<td>Advtg</td>
<td>1-now</td>
<td>1-4</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>20</td>
<td>Sales</td>
<td>1-4</td>
<td>1-now</td>
</tr>
</tbody>
</table>

A growing stair shape:
- Starting at time 5, the valid time extends from 5 to now.
- At each clock tick, the period end grows by one unit.
A growing stair shape with an initial big step:

- Starting at time 5, the valid time extends from 3 to now.
- At each clock tick, the period end grows by one unit.
### The Timestamp NOW/4

<table>
<thead>
<tr>
<th>E-ID</th>
<th>EName</th>
<th>Salary</th>
<th>Dept</th>
<th>TT</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jane</td>
<td>40</td>
<td>Advtg</td>
<td>5-now</td>
<td>5-now</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>20</td>
<td>Sales</td>
<td>5-now</td>
<td>3-now</td>
</tr>
<tr>
<td>1</td>
<td>Jane</td>
<td>30</td>
<td>Advtg</td>
<td>1-now</td>
<td>1-4</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>20</td>
<td>Sales</td>
<td>1-4</td>
<td>1-now</td>
</tr>
</tbody>
</table>

A growing period:
- Starting at time 1, the valid time extends from 1 to 4.
- The period remains part of the current state.
- The right end of the region tracks the current time.
### The Timestamp NOW/5

<table>
<thead>
<tr>
<th>E-ID</th>
<th>EName</th>
<th>Salary</th>
<th>Dept</th>
<th>TT</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jane</td>
<td>40</td>
<td>Advtg</td>
<td>5-now</td>
<td>5-now</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>20</td>
<td>Sales</td>
<td>5-now</td>
<td>3-now</td>
</tr>
<tr>
<td>1</td>
<td>Jane</td>
<td>30</td>
<td>Advtg</td>
<td>1-now</td>
<td>1-4</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>20</td>
<td>Sales</td>
<td>1-4</td>
<td>1-now</td>
</tr>
</tbody>
</table>

A static stair shape:

- Starting at time 1, the valid time extends from 1 to now until time 4.
- The shape grew until time 4, but is now static.
Implications of Timestamp NOW

- When \textit{now} is stored as a timestamp then the database state changes continuously as time passes by.
- Some facts stop being valid as time passes by.
- Some facts start being valid as time passes by.
- Index structures must also be dynamic (cf. TPR-Tree)
- The database integrity might get violated as time passes by.
- Materialized views (precomputed results) get outdated as time passes by.
- The timestamp \textit{now} cannot be approximated by NULL or infinity (e.g., a query that retrieves all periods that are between 5 and 7 years long).
Relation schema: $R = (A_1, \ldots, A_n, T_s, T_e, V_s, V_e)$
- $V$ is a shorthand for $(V_s, V_e)$, $T$ is a shorthand for $(T_s, T_e)$

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

$$\pi^B_D(r) = \{ z^{(|D|+4)} \mid \exists x \in r (z.D = x.D \land z.T = x.T \land z.V = x.V) \}$$

Note:
- The result of a projection might not be coalesced (even if the argument relation is coalesced).
- $\{(C101, T1234, 7, now, 8, 10), (C101, T1245, 7, now, 11, 15)\}$
Temporal natural join: Two tuples join if they match on the join attributes and have overlapping timestamps.

\[ r \Join^B s = \{ z^{(n+l+m+4)} | (\exists x \in r \exists y \in s (z.A = x.A = y.A \land x.T \cap y.T \neq \emptyset \land x.V \cap y.V \neq \emptyset \land z.B = x.B \land z.C = y.C \land z.T = x.T \cap y.T \land z.V = x.V \cap y.V) \} \]

Note:
- The result of a temporal bitemporal join is again a bitemporal relation.
- The original timestamps are not included in the result relation.
Jensen’s Backlog-Based Data Model

- **Jensen’s backlog-based data model**
  - Supports valid time and transaction time
  - Adds four **atomic-valued attributes** to each relation
    - Start and end point of the valid time \((V_s, V_e)\)
    - Transaction time when the tuple was inserted into the backlog \((T)\)
    - An operation which is either insert or delete \((Op)\)
  - Schema: \(R = (A_1, \ldots, A_n, V_s, V_e, T, Op)\)
  - Tuples in backlogs are never updated, i.e., backlogs are **append-only**
  - 1NF relations
  - The fact in an insertion request is current starting at the transaction’s timestamp and until a matching delete request is recorded.
Example: Department relation in Jensen’s backlog model

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
<th>$V_s$</th>
<th>$V_e$</th>
<th>$T$</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>10</td>
<td>15</td>
<td>5</td>
<td>I</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>5</td>
<td>20</td>
<td>10</td>
<td>I</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>5</td>
<td>20</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>I</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>D</td>
</tr>
<tr>
<td>Kate</td>
<td>Ship</td>
<td>25</td>
<td>30</td>
<td>20</td>
<td>I</td>
</tr>
</tbody>
</table>
Ben-Zvi’s Tuple Timestamped Data Model

- **Ben-Zvi’s Tuple Timestamped Data Model**
  - Adds five *atomic-valued attributes* to each relation
    - $T_{es}$ is the *effective start* of the non-temporal attributes values
    - $T_{ee}$ is the *effective end* of the non-temporal attributes values
    - $T_{rs}$ is the *registration start* and stores when the $T_{es}$ value was recorded
    - $T_{re}$ is the *registration end* and stores when the $T_{ee}$ value was recorded
    - $T_{ee}$ is not necessarily recorded at the same time as $T_{es}$ is recorded
    - $T_d$ indicates the time when the information in the tuple was logically deleted
  - The symbol “−” indicates
    - an unrecorded $T_{ee}(T_{re})$ value
    - in the $T_d$ field that the tuple contains current information
  - Schema: $R = (A_1, \ldots, A_n, T_{es}, T_{rs}, T_{ee}, T_{re}, T_d)$
  - 1NF relations
Example: Department relation in Ben-Zvi’s tuple timestamped data model

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
<th>$T_{es}$</th>
<th>$T_{rs}$</th>
<th>$T_{ee}$</th>
<th>$T_{re}$</th>
<th>$T_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>10</td>
<td>5</td>
<td>15</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Jake</td>
<td>Ship</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>Jake</td>
<td>Load</td>
<td>10</td>
<td>20</td>
<td>15</td>
<td>20</td>
<td>—</td>
</tr>
<tr>
<td>Kate</td>
<td>Ship</td>
<td>25</td>
<td>20</td>
<td>30</td>
<td>20</td>
<td>—</td>
</tr>
</tbody>
</table>
Gadia’s Attribute Value Timestamped Data Model/1

- Gadia’s attribute value timestamped data model
  - Supports valid time and transaction time
  - Schema:
    \[ R = \left\{ \left( [T_s, T_e] \times [V_s, V_e] A_1 \right), \ldots, \left( [T_s, T_e] \times [V_s, V_e] A_n \right) \right\} \]
  - A tuple is composed of \( n \) sets
    - Each set element is a triple of a transaction time period, a valid time period, and an attribute value
  - Non-1NF relations
  - A relation might be restructured (grouped) on different attributes
    - for example, group by department rather than employee yields facts for each department
Example: Department relation in Gadia’s attribute value timestamped data model

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>[20,now] × [10,15]</td>
<td>Jake [20,now] × [10,15]</td>
</tr>
<tr>
<td>[20,now] × [25,30]</td>
<td>Kate [20,now] × [25,30]</td>
</tr>
</tbody>
</table>
Table of Contents

- Modeling temporal data
  - Dimensions of time
  - Timestamping data
- Temporal data models
  - Snodgrass’ tuple timestamping model
  - Jensen’s backlog model
  - Ben-Zvi’s tuple timestamped model
  - Gadia’s attribute value timestamped model
- Comparative analysis
  - SQL + abstract data types
  - IXSQL
  - SQL/TP
  - TSQL2
  - ATSQL
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

- **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 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 $X$
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**: At each time point 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>
**Q4:** At each time point what is the number of tapes that have been checked out?

<table>
<thead>
<tr>
<th>Cnt</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[3,4]</td>
</tr>
<tr>
<td>2</td>
<td>[5,5]</td>
</tr>
<tr>
<td>1</td>
<td>[6,7]</td>
</tr>
<tr>
<td>0</td>
<td>[8,8]</td>
</tr>
<tr>
<td>1</td>
<td>[9,12]</td>
</tr>
<tr>
<td>0</td>
<td>[13,18]</td>
</tr>
</tbody>
</table>

**Q5:** How many checkouts were made in total?

<table>
<thead>
<tr>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>CustID</th>
<th>TapeNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>C101</td>
<td>T1234</td>
</tr>
<tr>
<td>C102</td>
<td>T1245</td>
</tr>
</tbody>
</table>
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.

- New and complex solutions must be invented (= programmed) for queries that refer to “at each time point”.
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 a 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
```
Unfold can be used to formulate queries about each time point.

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 queries about each time point (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)
  - 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 “at each time point” 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

```sql
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.

```sql
SELECT SeqNo, TapeNum, T
FROM CheckOut
GROUP BY SeqNo
HAVING COUNT(T) = 2
```
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.

Q3
```
SELECT b.TapeNum, b.T
FROM CheckOut AS a, CheckOut AS b
WHERE a.TapeNum = T1234
AND b.TapeNum <> T1234
AND a.T = b.T
```

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

The following fails for Q5:

```sql
SELECT COUNT(*) FROM CheckOut
```

Q5

```sql
SELECT COUNT(DISTINCT SeqNo) FROM CheckOut
```

Q6

```sql
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
Assessment of SQL/TP

- **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/1

- ATSQL introduces **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 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:
  - 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 CustID, TapeNum, T
    FROM CheckOut
    WHERE T OVERLAPS [7,9]

Q2  seq vt
    SELECT TapeNum
    FROM CheckOut
    WHERE DURATION(T) = 2
― “At each point in time” is easy to specify and the specification is independent of the query that shall be evaluated at each time point.

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 `nseq vt` ("nonsequenced valid time") modifier indicates that what follows should be treated as regular SQL

- The rest of the query is standard SQL

### Q5
```sql
NSEQ VT
SELECT COUNT(*)
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

### Q6
```sql
SELECT *
FROM CheckOut
```
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.
Temporal data models choose

- time dimensions (valid time, transaction time, ...)
- timestamps (points, intervals, temporal elements)
- scope of timestamps (attribute, tuple, relation, ...)

No single best choice exist. Assessment based on chosen timestamps is difficult; must be considered together with query language.

There is no agreement whether periods should have a meaning beyond a set of points (most likely they should).
There are a number of different temporal data models:
  ▶ Snodgrass’ tuple timestamping model
  ▶ Jensen’s backlog model
  ▶ Ben-Zvi’s tuple timestamped model
  ▶ Gadia’s attribute value timestamped model

Language design criteria: expressive power, orthogonality, closedness, clarity, consistency

Comparison of periods (Allen’s predicates): convenient but not a major issue

Now is a continuously moving constant
  ▶ Understand the meaning of timestamps that include now
  ▶ Often the largest timestamp is used to represent now but this leads to subtle problems (duration, etc)
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