Temporal Databases

Esteban ZIMÁNYI
Department of Computer & Decision Engineering (CoDE)
Université Libre de Bruxelles
ezimanyi@ulb.ac.be

Info-H-415 Advanced Databases
Academic Year 2014-2015

Temporal Databases: Topics

- Introduction
  - Time Ontology
  - Temporal Conceptual Modeling
  - Manipulating Temporal Databases with SQL-92
  - Temporal Support in SQL 2011
  - Summary
Introduction

Applications with temporal aspects abound
- Academic
- Accounting
- Data warehousing
- Financial
- Geographical Information Systems
- Insurance
- Inventory
- Law
- Medical records
- Reservation systems
- Scientific databases
- ... 

Need for a Temporal DBMS

- It is difficult to identify applications not needing management of temporal data
- These applications would benefit from built-in temporal support in the DBMS
  - More efficient application development
  - Potential increase of performance
- Temporal DBMS: Provide mechanisms to store and manipulate time-varying information
Temporal Databases: Case Study

- Personnel management in a database
  \[ \text{Employee}(\text{Name}, \text{Salary}, \text{Title}, \text{BirthDate} \ \text{DATE}) \]

- It is easy to know the salary of an employee
  
  SELECT \text{Salary} \\
  \text{FROM Employee} \\
  \text{WHERE Name = 'John'}

- It is also easy to know the date of birth of an employee
  
  SELECT \text{BirthDate} \\
  \text{FROM Employee} \\
  \text{WHERE Name = 'John'}

Converting to a Temporal Database

- We want to keep the employment history
  \[ \text{Employee}(\text{Name}, \text{Salary}, \text{Title}, \text{BirthDate}, \text{FromDate} \ \text{DATE}, \text{ToDate} \ \text{DATE}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Title</th>
<th>BirthDate</th>
<th>FromDate</th>
<th>ToDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>60.000</td>
<td>Assistant</td>
<td>9/9/60</td>
<td>1/1/95</td>
<td>1/6/95</td>
</tr>
<tr>
<td>John</td>
<td>70.000</td>
<td>Assistant</td>
<td>9/9/60</td>
<td>1/6/95</td>
<td>1/10/95</td>
</tr>
<tr>
<td>John</td>
<td>70.000</td>
<td>Lecturer</td>
<td>9/9/60</td>
<td>1/10/95</td>
<td>1/2/96</td>
</tr>
<tr>
<td>John</td>
<td>70.000</td>
<td>Professor</td>
<td>9/9/60</td>
<td>1/2/96</td>
<td>1/1/97</td>
</tr>
</tbody>
</table>

- For the data model, new columns are identical to attribute \text{BirthDate}
Determine the Salary

- To know the employee’s current salary, things are more difficult
  
  ```sql
  SELECT Salary
  FROM Employee
  WHERE Name = 'John' AND FromDate <= CURRENT_TIMESTAMP
  AND CURRENT_TIMESTAMP <= ToDate
  ```

- Determine the salary history
  
  - Result: for each person, the maximal intervals of each salary
    
    | Name | Salary | FromDate | ToDate |
    |------|--------|----------|--------|
    | John | 60.000 | 1/1/95   | 1/6/95 |
    | John | 70.000 | 1/6/95   | 1/1/97 |
  
  - An employee could have arbitrarily many title changes between salary changes

Determine the Salary, cont.

- Alternative 1
  
  - Give the user a printout of Salary and Title information, and have the user determine when his/her salary changed

- Alternative 2
  
  - Use SQL as much as possible
  
  - Find those intervals that overlap or are adjacent and that should be merged
CREATE TABLE Temp(Salary, FromDate, ToDate) AS
SELECT Salary, FromDate, ToDate
FROM Employee
WHERE Name = 'John'

repeat
    UPDATE Temp T1
    SET (T1.ToDate) = (SELECT MAX(T2.ToDate)
                        FROM Temp AS T2
                        WHERE T1.Salary = T2.Salary
                        AND T1.FromDate < T2.FromDate
                        AND T1.ToDate >= T2.FromDate
                        AND T1.ToDate < T2.ToDate)
    WHERE EXISTS ( SELECT *
                   FROM Temp as T2
                   WHERE T1.Salary = T2.Salary
                   AND T1.FromDate < T2.FromDate
                   AND T1.ToDate >= T2.FromDate
                   AND T1.ToDate < T2.ToDate)

until no tuples updated

---

**SQL Code, cont.**

- Initial table

- After one pass

- After two passes
SQL Code, cont.

- Loop is executed logN times in the worst case, where N is the number of tuples in a chain of overlapping or adjacent value-equivalent tuples
- Then delete extraneous, non-maximal intervals

```sql
DELETE FROM Temp T1
WHERE EXISTS (  
    SELECT *
    FROM Temp AS T2
    WHERE T1.Salary = T2.Salary
    AND ((T1.FromDate > T2.FromDate AND T1.ToDate <= T2.ToDate)
    OR (T1.FromDate >= T2.FromDate AND T1.ToDate < T2.ToDate) )
```

---

**Same Functionality Entirely in SQL**

```sql
CREATE VIEW Temp(Salary, FromDate, ToDate) AS
SELECT Salary, FromDate, ToDate
FROM Employee
WHERE Name = 'John'

SELECT DISTINCT F.Salary, F.FromDate, L.ToDate
FROM Temp AS F, Temp AS L
WHERE F.FromDate < L.ToDate AND F.Salary = L.Salary
AND NOT EXISTS (  
    SELECT *
    FROM Temp AS T
    WHERE T.Salary = F.Salary
    AND F.FromDate < T.FromDate AND T.FromDate < L.ToDate
    AND NOT EXISTS (  
        SELECT *
        FROM Temp AS T1
        WHERE T1.Salary = F.Salary
        AND T1.FromDate < T.FromDate AND T1.ToDate <= T1.ToDate )
    )
AND NOT EXISTS (  
    SELECT *
    FROM Temp AS T2
    WHERE T2.Salary = F.Salary
    AND ( (T2.FromDate < F.FromDate AND F.FromDate <= T2.ToDate)
    OR (T2.FromDate <= L.ToDate AND L.ToDate < T2.ToDate))
```
Same Query in Tuple Relational Calculus

\[
\{ f.\text{FromDate}, l.\text{ToDate} \mid \\
\quad \text{Temp}(f) \land \text{Temp}(l) \land f.\text{FromDate} < l.\text{ToDate} \land f.\text{Salary} = l.\text{Salary} \land \\
\quad (\forall t)(\text{Temp}(t) \land t.\text{Salary} = f.\text{Salary} \land f.\text{FromDate} < t.\text{FromDate} \land \\
\quad \quad t.\text{FromDate} < l.\text{ToDate} \rightarrow \\
\quad \quad (\exists t_1)(\text{Temp}(t_1) \land t_1.\text{Salary} = f.\text{Salary} \land t_1.\text{FromDate} < l.\text{FromDate} \land \\
\quad \quad \quad t.\text{FromDate} \leq t_1.\text{ToDate}) \land \\
\quad \quad \neg(\exists t_2)(\text{Temp}(t_2) \land t_2.\text{Salary} = f.\text{Salary} \land \\
\quad \quad \quad (\quad (t_2.\text{FromDate} < f.\text{FromDate} \land f.\text{FromDate} \leq t_2.\text{ToDate}) \lor \\
\quad \quad \quad \quad (t_2.\text{FromDate} \leq l.\text{ToDate} \land l.\text{ToDate} < t_2.\text{ToDate}))) \} \}
\]

Other Possibilities

- Use the transitive closure or triggers in SQL3
- TSQL2

```sql
SELECT Salary
FROM Employee
WHERE Name = 'Bob'
```
Alternative: Reorganize the Schema

- Split the information on Salary, Title, and BirthDate

  Employee(Name, BirthDate DATE)
  EmployeeSal(Name, Salary, FromDate DATE, ToDate DATE)
  EmployeeTitle(Name, Title, FromDate DATE, ToDate DATE)

- Determine the information about the salary is easy now

  SELECT Salary, FromDate, ToDate
  FROM EmployeeSal
  WHERE Name = 'John'

- However, how to obtain a table of salary, title intervals?

Example of Temporal Join

<table>
<thead>
<tr>
<th>EmployeeSal</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Salary</td>
<td>FromDate</td>
<td>ToDate</td>
</tr>
<tr>
<td>John</td>
<td>60.000</td>
<td>1/1/95</td>
<td>1/6/95</td>
</tr>
<tr>
<td>John</td>
<td>70.000</td>
<td>1/6/95</td>
<td>1/1/97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EmployeeTitle</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Title</td>
<td>FromDate</td>
<td>ToDate</td>
</tr>
<tr>
<td>John</td>
<td>Assistant</td>
<td>1/1/95</td>
<td>1/10/95</td>
</tr>
<tr>
<td>John</td>
<td>Lecturer</td>
<td>1/10/95</td>
<td>1/2/96</td>
</tr>
<tr>
<td>John</td>
<td>Professor</td>
<td>1/2/96</td>
<td>1/1/97</td>
</tr>
</tbody>
</table>

EmployeeSal ≈ EmployeeTitle

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Title</th>
<th>FromDate</th>
<th>ToDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>60.000</td>
<td>Assistant</td>
<td>1/1/95</td>
<td>1/6/95</td>
</tr>
<tr>
<td>John</td>
<td>70.000</td>
<td>Assistant</td>
<td>1/6/95</td>
<td>1/10/95</td>
</tr>
<tr>
<td>John</td>
<td>70.000</td>
<td>Lecturer</td>
<td>1/10/95</td>
<td>1/2/96</td>
</tr>
<tr>
<td>John</td>
<td>70.000</td>
<td>Professor</td>
<td>1/2/96</td>
<td>1/1/97</td>
</tr>
</tbody>
</table>
Evaluation of Temporal Join

* Alternative 1: Print the two tables and leave the user make the combinations
* Alternative 2: Use SQL entirely

```
SELECT S.Name, Salary, Title, S.FromDate, S.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND T.FromDate <= S.FromDate
AND S.ToDate < T.ToDate
UNION ALL
SELECT S.Name, Salary, Title, S.FromDate, T.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND S.FromDate > T.FromDate
AND T.ToDate < S.ToDate
AND S.ToDate < T.ToDate
UNION ALL
SELECT S.Name, Salary, Title, T.FromDate, S.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND T.FromDate >= S.FromDate
AND T.ToDate <= S.ToDate
AND T.FromDate < S.ToDate
UNION ALL
SELECT S.Name, Salary, Title, T.FromDate, T.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND T.ToDate >= S.ToDate
AND T.ToDate <= S.ToDate
AND T.ToDate <= S.ToDate
```

Temporal Join in SQL

```
SELECT S.Name, Salary, Title, S.FromDate, S.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND T.FromDate <= S.FromDate
AND S.ToDate <= T.ToDate
UNION ALL
SELECT S.Name, Salary, Title, S.FromDate, T.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND S.FromDate > T.FromDate
AND T.ToDate < S.ToDate
AND T.ToDate <= T.ToDate
UNION ALL
SELECT S.Name, Salary, Title, T.FromDate, S.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND T.FromDate <= S.FromDate
AND S.ToDate < T.ToDate
AND T.ToDate < S.ToDate
UNION ALL
SELECT S.Name, Salary, Title, T.FromDate, T.ToDate
FROM EmployeeSal S, EmployeeTitle T
WHERE S.Name = T.Name
AND T.FromDate >= S.FromDate
AND T.ToDate <= S.ToDate
AND T.ToDate <= S.ToDate
```

17

18
Temporal Join, cont.

- Alternative 3: Use embedded SQL

  TSQL2: Give the salary and title history of employees

  ```sql
  SELECT EmployeeSal.Name, Salary, Title
  FROM EmployeeSal, EmployeeTitle
  WHERE EmployeeSal.Name = EmployeeTitle.Name
  ```

Introduction: Summary

- Applications managing temporal data abound
- Classical DBMS are not adequate
- If a temporal DBMS is used
  - Schemas are simpler
  - SQL queries are much simpler
  - Much less procedural code is necessary
- Benefits
  - Application code is less complex
    - Easier to understand, to produce, to ensure correctness, to maintain
  - Performance may be increased by relegating functionality to DBMS
Temporal Databases: Topics

- Introduction
- **Time Ontology**
  - Temporal Conceptual Modeling
  - Manipulating Temporal Databases with SQL-92
  - Temporal Support in SQL 2011
  - Summary

Time Ontology

- Notions of time
  - Structure
  - Density
  - Boundedness
- TSQL2 time ontology
- Time data types
- Times and facts
**Time Structure**

- Linear: total order on instants
  
- Hypothetical (possible futures): tree rooted on now
  
- Directed Acyclic Graph (DAG): possible futures may merge
- Periodic/cyclic time: weeks, months, . . ., for recurrent processes

**Boundedness of Time**

- Assume a linear time structure
- Boundedness
  - Unbounded
  - Time origin exists (bounded from the left)
  - Bounded time (bounds on two ends)
- Nature of bound
  - Unspecified
  - Specified
- Physicists believe that the universe is bounded by the “Big Bang” (12-18 billions years ago) and by the “Big Crunch” (? billion years in the future)
### Time Density

- **Discrete**
  - Time line is isomorphic to the integers
  - Time line is composed of a sequence of non-decomposable time periods, of some fixed minimal duration, termed *chronons*
  - Between each pair of chronons is a finite number of other chronons

- **Dense**
  - Time line is isomorphic to the rational numbers
  - Infinite number of instants between each pair of chronons

- **Continuous**
  - Time line is isomorphic to the real numbers
  - Infinite number of instants between each pair of chronons

- **Distance** may optionally be defined

---

### TSQL2: Time Ontology

- **Structure**
  - TSQL2 uses a linear time structure

- **Boundedness**
  - TSQL2 time line is bounded on both ends, from the start of time to a point far in the future

- **Density**
  - TSQL2 do not differentiate between discrete, dense, and continuous time ontologies
  - No questions can be asked that give different answers
    - E.g., instant $a$ precedes instant $b$ at some specified granularity. Different granularities give different answers
  - Distance is defined in terms of numbers of chronons
Ontological Temporal Types

- **Instant**: chronon in the timeline
  - **Event**: instantaneous fact, something occurring at an instant
  - **Event occurrence time**: valid-time instant at which the event occurs in the real world
- **Instant Set**: set of instants
- **Time period**: time between two instants
  - Also called interval, but conflicts with SQL data type `INTERVAL`
- **Time interval**: a directed duration of time
- **Duration**: amount of time with a known length, but no specific starting or ending instants
  - **positive interval**: forward motion time
  - **negative interval**: backward motion time
- **Temporal element**: finite union of periods

Representing Time in TSQL2

- TSQL2 supports a bounded discrete representation of the time line
- Time line composed of chronons, which is the smallest granularity
- Consecutive chronons may be grouped together into granules, yielding multiple granularities
- Different granularities are available, and it is possible to convert from one granularity to another (via scaling)
Temporal Data Types in SQL-92 and TSQL2

- SQL92
  - DATE (YYYY-MM-DD)
  - TIME (HH:MM:SS)
  - DATETIME (YYYY-MM-DD HH:MM:SS)
  - INTERVAL (no default granularity)

- TSQL2
  - PERIOD: DATETIME - DATETIME

Time and Facts

- Valid time of a fact: when the fact is true in the modeled reality
  - Independently of its recording in the database
  - Past, present, future

- Transaction time of a fact: when the fact is current in the database and may be retrieved
  - Identify the transactions that inserted and deleted the fact

- Two dimensions are orthogonal

- Four kinds of tables
  - Snapshot
  - Valid time
  - Transaction time
  - Bitemporal
**Snapshot Tables**

- May be modified
- Used for static queries
- What is John’s title?

  ```sql
  SELECT Title
  FROM Faculty
  WHERE Name = 'John'
  ```

---

**Snapshot Tables, cont.**

- Analogy: Nameplate on door

<table>
<thead>
<tr>
<th>John</th>
<th>John</th>
<th>John</th>
<th>John</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assistant</td>
<td>1st Assistant</td>
<td>1st Assistant</td>
<td>Lecturer</td>
</tr>
<tr>
<td>Jan. 84</td>
<td>Dec. 87</td>
<td>March 89</td>
<td>July 89</td>
</tr>
</tbody>
</table>

- On January 1st, 1984, John is hired as assistant
- On December 1st, 1987, John finishes his doctorate and is promoted as 1st Assistant retroactively on July 1st, 1987
- On March 1st, 1989, John is promoted as Lecturer, proactively on July 1st, 1989
Append-only: correction to previous snapshot states is not permitted

Allow retrospective queries ("rollback")

What did we believe John’s rank was on October 1st, 1984?

```sql
SELECT Title
FROM Faculty
WHERE Name = 'John' AND
TRANSACTION(Faculty) OVERLAPS DATE '01-10-1984'
```

---

Analogy: Pay stubs

- John Assistant 1-1-84
- John Assistant 1-11-87
- John 1st Assistant 1-12-87
- John 1st Assistant 1-6-89
- John Lecturer 1-7-89
May be modified
Allow historical queries
What was John’s title on October 1st, 1984 (as best known)?

```
SELECT Title
FROM Faculty
WHERE Name = 'John' AND
      VALID(Faculty) OVERLAPS DATE '01-10-1984'
```

Valid Time Tables, cont.

Analogy: Curriculum Vitæ

<table>
<thead>
<tr>
<th>John</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Titles</strong></td>
</tr>
<tr>
<td>Lecturer</td>
</tr>
<tr>
<td>1st Assistant</td>
</tr>
<tr>
<td>Assistant</td>
</tr>
</tbody>
</table>
Bitemporal Tables

- Append-only
- Transaction and valid time
- Allow coupled historical and retrospective queries
- On October 1st, 1984, what did we think John’s rank was at that date?

```sql
SELECT Title
FROM Faculty AS E
WHERE Name = 'John' AND
  VALID(E) OVERLAPS DATE '01-10-1984' AND
  TRANSACTION(E) OVERLAPS DATE '01-10-1984'
```

---

Bitemporal Tables, cont.

- Analogy: Stack of CVs

<table>
<thead>
<tr>
<th>Titles</th>
<th>January 1984 Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Assistant</td>
<td>July 1987</td>
</tr>
<tr>
<td>Assistant</td>
<td>January 1984</td>
</tr>
<tr>
<td>John</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Titles</th>
<th>December 1987 Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td></td>
</tr>
<tr>
<td>Lecturer</td>
<td>July 1989</td>
</tr>
<tr>
<td>1st Assistant</td>
<td>July 1987</td>
</tr>
<tr>
<td>Assistant</td>
<td>January 1984</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Titles</th>
<th>March 1989 Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td></td>
</tr>
<tr>
<td>1st Assistant</td>
<td>July 1987</td>
</tr>
<tr>
<td>Assistant</td>
<td>January 1984</td>
</tr>
</tbody>
</table>
**Time Ontology: Summary**

- Several different structures of time
  - Linear is simplest and most common
- 5 fundamental temporal data types
- Several dimensions of time
  - TSQL2 supports transaction and valid time

**Temporal Databases: Topics**

- Introduction
- Time Ontology
  - **Temporal Conceptual Modeling**
- Manipulating Temporal Databases with SQL-92
- Temporal Support in SQL 2011
- Summary
Why Conceptual Modeling?

- Focuses on the application
- Technology independent
  - portability, durability
- User oriented
- Formal, unambiguous specification
- Supports visual interfaces
  - data definition and manipulation
- Best vehicle for information exchange/integration

The Conceptual Manifesto (1)

- Semantically powerful data structures
- Simple (understandable) data model
  - few clean concepts, with standard, well-known semantics
- No artificial time objects
- Time orthogonal to data structures
- Various granularities
- Clean, visual notations
- Intuitive icons / symbols
**Orthogonality**

- **Project**: project_no, project_name
- **Employee**: SSN, emp_name, projects (1,n)
- **EmpDep**: (1,n)
- **Department**: dept_no, dept_name

---

**The Conceptual Manifesto (2)**

- Explicit temporal relationships and integrity constraints
- Support of valid time and transaction time
- Past to future
- Co-existence of temporal and traditional data
- Query languages
- Complete and precise definition of the model
Temporal Information Describes ...

- Life cycles of objects and relationships
- Validity of information values
  - Timestamps
- Temporal relationships
  - Temporal links
  - Temporal integrity constraints
MADS Temporal Data Types

- **Time**, **Simple Time**, and **Complex Time** are abstract classes.

Temporal Objects

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>birthDate</td>
</tr>
<tr>
<td>address</td>
</tr>
<tr>
<td>salary</td>
</tr>
<tr>
<td>projects (1,n)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e221 Peter</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/9/64</td>
</tr>
<tr>
<td>Rue de la Paix</td>
</tr>
<tr>
<td>5000</td>
</tr>
<tr>
<td>{MADS, HELIOS}</td>
</tr>
</tbody>
</table>

- [7/94-6/96] [7/97-6/98] active
- [7/96-6/97] suspended
- Life cycle information
Object / Relationship Life Cycle

- Continuous

create ————> kill

- Discontinuous

create ————> suspend ————> reactivate ————> kill

Non-Temporal Objects?

- No life cycle, or
- Default life cycle
  - active $\rightarrow [0, \text{now}]$
  - active $\rightarrow [\text{now}, \text{now}]$
  - active $\rightarrow [0, \infty]$
- Coexistence
  - temporal $\rightarrow$ non temporal (snapshot)
  - non-temporal $\rightarrow$ temporal (default life cycle)
**TSQL2 Policy**

- Temporal operators not allowed on non-temporal relations
  - no life cycle
- Joins between temporal and non-temporal relations are allowed
  - default life cycle: active → [0, ∞]

SELECT Department.Name, COUNT (PID)
FROM Department, Employee
WHERE Employee.dept # = Department.dept #
  AND VALID(Employee) OVERLAPS PERIOD ' [1/1/96-31/12/96]'
GROUP BY dept #

---

**Temporal Attributes**

<table>
<thead>
<tr>
<th>Employee</th>
<th>name</th>
<th>birthDate</th>
<th>address</th>
<th>salary</th>
<th>projects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1,n)</td>
</tr>
</tbody>
</table>

Peter
8/9/64

Bd St Germain
[1/85-12/87]
Bd St Michel
[1/88-12/94]
Rue de la Paix
[1/95-now]

4000
[7/94-7/95]
5000
[8/95-now]
(MADS)
[7/94-8/95]
(MADS, HELIOS)
[9/95-now]
Temporal Complex Attributes (1)

<table>
<thead>
<tr>
<th>Laboratory</th>
<th>LBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>name projects (1,n)</td>
<td>{(MADS, Chris,1500)) {Helios, Martin,2000) {1/1/95 -31/12/95] [1/1/95 -now]</td>
</tr>
<tr>
<td>name manager budget</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Laboratory</th>
<th>LBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>name projects (1,n)</td>
<td>{ MADS, Stef [x/x/x -- x/x/x] , 1500), Chris [x/x/x -- x/x/x] , Martin [x/x/x -- x/x/x] , 2000)</td>
</tr>
<tr>
<td>name manager budget</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Laboratory</th>
</tr>
</thead>
<tbody>
<tr>
<td>name project name manager</td>
</tr>
</tbody>
</table>

“Updating” manager ⇒ add element to manager history
“Updating” project.name (name of project has changed) ⇒ update name
“Updating” project (laboratory changed project) ⇒ update name, start new history for manager
**Attribute Timestamping Properties**

- Attribute types / timestamping
  - none, irregular, regular, instants, durations, ...
- Cardinalities
  - snapshot and DBlifespan
- Identifiers
  - snapshot or DBlifespan

**Attribute Timestamping Issues**

- Constraints?
  - the validity period of an attribute must be within the life cycle of the object it belongs to
  - the validity period of a complex attribute is the union of the validity periods of its components
- MADS : no implicit constraint
Temporal Generalization

Person
- SSN
- name
- birthDate
- address

Employee
- salary
- projects

John
- 3/7/55
- High Street
- Victoria Street
- [3/93-2/95]
- [3/95-now]

Peter
- 8/9/64
- East Terrace
- Flinders Street
- [1/87-6/94]
- [7/94-now]

4000 [7/94-7/95]
5000 [8/95-now]
{MADS} [7/94-8/95]
{MADS, HELIOS} [9/95-now]

Static Temporal Generalization

Employee

Temporary
Permanent

- Temporary and Permanent are implicitly temporal
  - they inherit their life cycle from Employee
Dynamic Temporal Generalization

- **Student** and **Faculty** have two life cycles:
  - an inherited one (the one of **Person**)
  - a redefined one (the one of **Student/Faculty**)
- The redefined life cycle has to be included in the one of the corresponding **Person**
  - lifespan and active periods

Temporal Relationships (1)

<table>
<thead>
<tr>
<th>Employee</th>
<th>WorksOn</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>hours/week</td>
<td>name</td>
</tr>
<tr>
<td>birthDate</td>
<td></td>
<td>manager</td>
</tr>
<tr>
<td>address</td>
<td></td>
<td>budget</td>
</tr>
</tbody>
</table>

- **John**
  - 3/7/55
  - Bd Haussman
- **Peter**
  - 8/10/64
  - Rue de la Paix

- **MADS**
  - Christine
  - 5000
- **HELIOS**
  - Yves
  - 6000
Relationship Timestamping Issues

- Constraints?
  - the validity period of a relationship must be within the intersection of the life cycles of the objects it links
  - a temporal relationship can only link temporal objects

- MADS: no implicit constraint

Temporal Relationships (2)

- Only currently valid couples are kept in the relationship
Temporal Relationships (3)

- Only currently valid objects participate in the relationship.

Synchronization Relationships (1)

- Describe temporal constraints between the life cycles of two objects.
- Expressed with Allen’s operator extended for temporal elements.

before ——- 
meets ——- 
overlaps ——- 
during ——- 
starts ——- 
finishes ——-
Synchronization Relationships (2)

- contains
- isContainedIn
- Takes (0,n)
- Photo
- Reporter
- Photo
- Person
- Ancestor
- Ancestor
- follows
- precedes
- ancestor
descendent

<table>
<thead>
<tr>
<th>Synchronization Relationship</th>
<th>Icon</th>
<th>Synchronization Relationship</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncGeneric</td>
<td></td>
<td>SyncStart</td>
<td></td>
</tr>
<tr>
<td>SyncDisjoint</td>
<td></td>
<td>SyncFinishes</td>
<td></td>
</tr>
<tr>
<td>SyncOverlap</td>
<td></td>
<td>SyncEqual</td>
<td></td>
</tr>
<tr>
<td>SyncWithin</td>
<td></td>
<td>SyncPrecede</td>
<td></td>
</tr>
<tr>
<td>SyncMeet</td>
<td></td>
<td>SyncFollow</td>
<td></td>
</tr>
</tbody>
</table>

Synchronization Relationships (3)

- Express a temporal constraint between
  - the whole life cycles, or
  - the active periods
- Temporal constraint defined with
  - extended Allen’s operators
  - application-defined operators, e.g., 9 months later
- They are relationships
  - may have attributes, cardinalities
Temporal Conceptual Models: Conclusion

- Conceptual models must be extended with temporal features
- Orthogonality is the answer for achieving maximal expressive power
- Semantics of temporal features must be explicitly defined
- This semantics generalizes that of the traditional conceptual models
- Temporal conceptual models are easily understood by users

Temporal Databases: Topics

- Introduction
- Time Ontology
- Temporal Conceptual Modeling
- Manipulating Temporal Databases with SQL-92
- Temporal Support in SQL 2011
- Summary
Defining Valid-Time Tables in SQL

<table>
<thead>
<tr>
<th>Employee</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>FirstName</td>
</tr>
<tr>
<td>Incumbents</td>
<td>SSN</td>
</tr>
</tbody>
</table>

- **Incumbents** and **Salary** are valid-time tables
  - **FromDate** indicates when the information in the row is valid, i.e. when the employee was assigned to that position
  - **ToDate** indicates when the information in the row was no longer valid

- Data type for periods is not available in SQL-92 ⇒ a period is simulated with two **Date** columns

Example of a Valid-Time Table

<table>
<thead>
<tr>
<th>Incumbents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
</tr>
<tr>
<td>111223333</td>
</tr>
<tr>
<td>111223333</td>
</tr>
<tr>
<td>111223333</td>
</tr>
<tr>
<td>111223333</td>
</tr>
<tr>
<td>111223333</td>
</tr>
</tbody>
</table>

- Special date '3000-01-01' denotes currently valid
- Closed-open periods used, e.g., validity of first tuple is [1996-01-01, 1996-06-01)
- Table can be viewed as a compact representation of a sequence of snapshot tables, each valid on a particular day
- Constraint: Employees do not have gaps in their position history
- Last two rows may be replaced with a single row valid at [1996-06-01, 3000-10-01)
Manipulating Temporal Tables: Semantics

Types of Temporal Statements

- Applies to queries, modifications, views, integrity constraints
- **Current**: Applies to the current point in time (now)
  - What is Bob’s current position?
- **Time-sliced**: Applies to some point in time in the past or the future
  - What was Bob’s position on January 1st, 2007?
- **Sequenced**: Applies to each point in time
  - What is Bob’s position history?
- **Non-sequenced**: Applies to all points in time, ignoring the time-varying nature of tables
  - When did Bob change history?
Temporal Keys

<table>
<thead>
<tr>
<th>SSN</th>
<th>PCN</th>
<th>FromDate</th>
<th>ToDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>111223333</td>
<td>900225</td>
<td>1996-01-01</td>
<td>1996-06-01</td>
</tr>
<tr>
<td>111223333</td>
<td>900225</td>
<td>1996-04-01</td>
<td>1996-10-01</td>
</tr>
</tbody>
</table>

- **Constraint**: Employees have only one position at a point in time
- In the corresponding non-temporal table the key is (SSN, PCN)
- Candidate keys on `Incumbents`: (SSN, PCN, FromDate), (SSN, PCN, ToDate), and (SSN, PCN, FromDate, ToDate)
- None captures the constraint: there are overlapping periods associated with the same SSN
- What is needed: sequenced constraint, applied at each point in time
- All constraints specified on a snapshot table have sequenced counterparts, specified on the analogous valid-time table

---

Sequenced Primary Key

- **Constraint**: Employees have only one position at a point in time

```sql
CREATE TRIGGER Seq_Primary_Key ON Incumbents
FOR INSERT, UPDATE AS
IF EXISTS (
    SELECT * FROM Incumbents AS I1
    WHERE 1 <
        ( SELECT COUNT(I2.SSN) FROM Incumbents AS I2
          WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
          AND I1.FromDate < I2.ToDate
          AND I2.FromDate < I1.ToDate )
)
OR
EXISTS (
    SELECT * FROM Incumbents AS I
    WHERE I.SSN IS NULL OR I.PCN IS NULL )
BEGIN
    RAISERROR('Violation of sequenced primary key constraint',1,2)
    rollback transaction
END
```
Handling Now

- What should the timestamp be for current data?
- One alternative: using NULL
- Allows to indentify current records: WHERE Incumbents.ToDate IS NULL
  - Disadvantages
    - users get confused with a data of NULL
    - in SQL any comparison with a null value returns false
      ⇒ rows with null values will be absent from the result of many queries
    - other uses of NULL are not available
- Another approach: set the end date to largest value in the timestamp domain, e.g., ‘3000-01-01’
  - Disadvantages
    - DB states that something will be true in the far future
    - represent ‘now’ and ‘forever’ in the same way

Types of Duplicates

<table>
<thead>
<tr>
<th>Incumbents</th>
<th>SSN</th>
<th>PCN</th>
<th>FromDate</th>
<th>ToDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111223333</td>
<td>1200033</td>
<td>1996-01-01</td>
<td>1996-06-01</td>
</tr>
<tr>
<td>2</td>
<td>111223333</td>
<td>1200033</td>
<td>1996-04-01</td>
<td>1996-10-01</td>
</tr>
<tr>
<td>3</td>
<td>111223333</td>
<td>1200033</td>
<td>1996-04-01</td>
<td>1996-10-01</td>
</tr>
<tr>
<td>4</td>
<td>111223333</td>
<td>1200033</td>
<td>1996-10-01</td>
<td>1998-01-01</td>
</tr>
<tr>
<td>5</td>
<td>111223333</td>
<td>1200033</td>
<td>1997-12-01</td>
<td>1998-01-01</td>
</tr>
</tbody>
</table>

- Two rows are value equivalent if the values of their nontimestamp columns are equivalent
- Two rows are sequenced duplicates if they are duplicates at some instant: 1+2 ⇒ employee has two positions for the months of April and May of 1996
- Two rows are current duplicates if they are sequenced duplicates at the current instant: 4+5 ⇒ in December 1997 a current duplicate will suddenly appear
- Two rows are nonsequenced duplicates if the values of all columns are identical: 2+3
Preventing Duplicates (1)

- Preventing value-equivalent rows: define secondary key using `UNIQUE(SSN, PCN)`
- Preventing nonsequenced duplicates: `UNIQUE(SSN, PCN, FromDate, ToDate)`
- Preventing current duplicates: No employee can have two identical positions at the current time

```sql
CREATE TRIGGER Current_Duplicates ON Incumbents
FOR INSERT, UPDATE, DELETE AS
IF EXISTS ( SELECT I1.SSN FROM Incumbents AS I1 WHERE 1 <
( SELECT COUNT(I2.SSN) FROM Incumbents AS I2
WHERE I1.SSN = I2.SSN AND I1.PCN=I2.PCN
AND I1.FromDate <= CURRENT_DATE
AND CURRENT_DATE < I1.ToDate
AND CURRENT_DATE < I2.ToDate ) )
BEGIN
RAISERROR('Transaction allows current duplicates',1,2)
rollback transaction
END
```

Preventing Duplicates (2)

- Preventing current duplicates, assuming no future data: current data will have the same `ToDate`
  (`'3000-01-01'`) ⇒ `UNIQUE(SSN, PCN, ToDate)`
- Preventing sequenced duplicates: since a primary key is a combination of `UNIQUE` and `NOT NULL`, remove the `NOT NULL` portion of code for keys in the previous trigger

```sql
CREATE TRIGGER Seq_Primary_Key ON Incumbents
FOR INSERT, UPDATE, DELETE AS
IF EXISTS ( SELECT I1.SSN FROM Incumbents AS I1 WHERE 1 <
( SELECT COUNT(I2.SSN) FROM Incumbents AS I2
WHERE I1.SSN = I2.SSN AND I1.PCN=I2.PCN
AND I1.FromDate < I2.ToDate
AND I2.FromDate < I1.ToDate ) )
BEGIN
RAISERROR('Transaction allows sequenced duplicates',1,2)
rollback transaction
END
```

- Preventing sequenced duplicates, assuming only current modifications: `UNIQUE(SSN, PCN, ToDate)`
Uniqueness (1)

- Constraint: Each employee has at most one position
- Snapshot table: UNIQUE(SSN)
- Sequenced constraint: At any time each employee has at most one position, i.e., Incumbents.SSN is sequenced unique

```sql
CREATE TRIGGER Seq_Unique ON Incumbents
FOR INSERT, UPDATE, DELETE AS
IF EXISTS ( SELECT I1.SSN FROM Incumbents AS I1 WHERE 1 <
    ( SELECT COUNT(I2.SSN) FROM Incumbents AS I2
    WHERE I1.SSN = I2.SSN
    AND I1.FromDate < I2.ToDate
    AND I2.FromDate < I1.ToDate ) )
    OR
    EXISTS ( SELECT * FROM Incumbents AS I
    WHERE I.SSN IS NULL )
BEGIN
    RAISERROR('Transaction violates sequenced unique constraint',1,2)
    rollback transaction
END
```

Uniqueness (2)

- Nonsequenced constraint: an employee cannot have more than one position over two identical periods, i.e., Incumbents.SSN is nonsequenced unique:

```
CREATE TRIGGER Current_Unique ON Incumbents
FOR INSERT, UPDATE, DELETE AS
IF EXISTS ( SELECT I1.SSN FROM Incumbents AS I1 WHERE 1 <
    ( SELECT COUNT(I2.SSN) FROM Incumbents AS I2
    WHERE I1.SSN = I2.SSN
    AND I1.FromDate <= CURRENT_DATE
    AND CURRENT_DATE < I1.ToDate ) )
BEGIN
    RAISERROR('Transaction violates current unique constraint',1,2)
    rollback transaction
END
```
**Referential Integrity (1)**

- **Incumbents.PCN** is a foreign key for **Position.PCN**

- **Case 1**: Neither table is temporal
  
  ```sql
  CREATE TABLE Incumbents ( ... 
  PCN CHAR(6) NOT NULL REFERENCES Position, ... )
  ```

- **Case 2**: Both tables are temporal
  
  The PCN of all current incumbents must be listed in the current positions
  
  ```sql
  CREATE TRIGGER Current_Referential_Integrity ON Incumbents 
  FOR INSERT, UPDATE, DELETE AS 
  IF EXISTS ( SELECT * FROM Incumbents AS I 
  WHERE I.ToDate = '3000-01-01' 
  AND NOT EXISTS ( 
    SELECT * FROM Position AS P 
    WHERE I.PCN = P.PCN AND P.ToDate = '3000-01-01' ) ) 
  BEGIN 
    RAISERROR('Violation of current referential integrity',1,2) 
    ROLLBACK TRANSACTION 
  END
  ```

**Referential Integrity (2)**

- **Incumbents.PCN** is a sequenced foreign key for **Position.PCN**

  ```sql
  CREATE TRIGGER Sequenced_Ref_Integrity ON Incumbents 
  FOR INSERT, UPDATE, DELETE AS 
  IF EXISTS ( SELECT * FROM Incumbents AS I 
  WHERE NOT EXISTS ( 
    SELECT * FROM Position AS P 
    WHERE I.PCN = P.PCN AND P.FromDate <= I.FromDate 
    AND I.FromDate < P.ToDate ) 
  OR NOT EXISTS ( 
    SELECT * FROM Position AS P 
    WHERE I.PCN = P.PCN AND P.ToDate <= I.ToDate 
    AND I.ToDate < P.FromDate ) 
  OR EXISTS ( 
    SELECT * FROM Position AS P2 
    WHERE P2.PCN = P.PCN AND P2.FromDate <= P.ToDate 
    AND P.ToDate < P2.FromDate 
    AND P.ToDate < P2.ToDate ) ) 
  BEGIN 
    RAISERROR('Violation of sequenced referential integrity',1,2) 
    ROLLBACK TRANSACTION 
  END
  ```
Contiguous History

- **Incumbents.PCN** defines a contiguous history

```
CREATE TRIGGER Contiguous_History ON Position
FOR INSERT, UPDATE, DELETE AS
IF EXISTS (
    SELECT * FROM Position AS P1, Position AS P2
    WHERE P1.PCN = P2.PCN AND P1.ToDate < P2.FromDate
    AND NOT EXISTS (
        SELECT * FROM Position AS P3
        WHERE P3.PCN = P1.PCN
        AND ( ( P3FromDate <= P1ToDate
            AND P1ToDate < P3ToDate )
            OR ( P3FromDate < P2FromDate
                AND P2ToDate <= P3ToDate ) ) )
BEGIN
    RAISERROR('Transaction violates contiguous history',1,2)
    ROLLBACK TRANSACTION
END
```

- This is a **nonsequenced constraint**: it requires examining the table at multiple points of time

---

Referential Integrity (3)

- **Incumbents.PCN** is a sequenced foreign key for **Position.PCN**, and **Incumbents.PCN** defines a contiguous history

```
CREATE TRIGGER Sequenced_Ref_Integrity ON Incumbents
FOR INSERT, UPDATE, DELETE AS
IF EXISTS (
    SELECT * FROM Incumbents AS I
    WHERE NOT EXISTS (
        SELECT * FROM Position AS P WHERE I.PCN = P.PCN
        AND P.FromDate <= I.ToDate AND I.FromDate < P.ToDate )
    OR NOT EXISTS (
        SELECT * FROM Position AS P WHERE I.PCN = P.PCN
        AND PFromDate < I.ToDate AND IToDate <= P.ToDate )
BEGIN
    RAISERROR('Violation of sequenced referential integrity',1,2)
    ROLLBACK TRANSACTION
END
```

---

83

84
**Referential Integrity (4)**

- **Case 4:** Only the referenced table is temporal
- Incumbents.PCN is a current foreign key for Position.PCN

```sql
CREATE TRIGGER Current_Referential_Integrity ON Incumbents
FOR INSERT, UPDATE, DELETE AS
IF EXISTS (SELECT * FROM Incumbents AS I
WHERE NOT EXISTS (SELECT * FROM Position AS P
WHERE I.PCN = P.PCN AND P.ToDate = '3000-01-01') )
BEGIN
RAISERROR('Violation of current referential integrity',1,2)
ROLLBACK TRANSACTION
END
```

---

**Querying Valid-Time Tables**

<table>
<thead>
<tr>
<th>Employee</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>PCN</td>
</tr>
<tr>
<td>FirstName</td>
<td>JobTitle</td>
</tr>
<tr>
<td>LastName</td>
<td></td>
</tr>
<tr>
<td>BirthDate</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incumbents</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>PCN</td>
</tr>
<tr>
<td>FromDate</td>
<td>ToDate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee</th>
<th>Incumbents</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>PCN</td>
<td>FromDate</td>
</tr>
</tbody>
</table>

- As for constraints, queries and modifications can be of three kinds
  - **current**, **sequenced**, and **nonsequenced**
- Extracting the current state: What is Bob’s current position
  ```sql
  SELECT JobTitle
  FROM Employee E, Incumbents I, Position P
  WHERE E.FirstName = 'Bob'
  AND E.SSN = I.SSN AND I.PCN = P.PCN
  AND I.ToDate = '3000-01-01'
  ```
Extracting Current State (1)

- Another alternative for obtaining Bob’s current position
  
  ```sql
  SELECT JobTitle
  FROM Employee E, Incumbents I, Position P
  WHERE E.FirstName = 'Bob'
  AND E.SSN = I.SSN AND I.PCN = P.PCN
  AND IFromDate <= CURRENT_DATE AND CURRENT_DATE < IToDate
  ```

- Current joins over two temporal tables are not too difficult

- What is Bob’s current position and salary?
  
  ```sql
  SELECT JobTitle, Amount
  FROM Employee E, Incumbents I, Position P, Salary S
  WHERE FirstName = 'Bob'
  AND E.SSN = I.SSN AND I.PCN = P.PCN AND E.SSN = S.SSN
  AND IFromDate <= CURRENT_DATE AND CURRENT_DATE < IToDate
  AND SFromDate <= CURRENT_DATE AND CURRENT_DATE < SToDate
  ```

Extracting Current State (2)

- What employees currently have no position?
  
  ```sql
  SELECT FirstName
  FROM Employee E
  WHERE NOT EXISTS (
    SELECT *
    FROM Incumbents I
    WHERE E.SSN = I.SSN
    AND IFromDate <= CURRENT_DATE AND CURRENT_DATE < IToDate
  )
  ```
Extracting Prior States

- **Timeslice queries**: extracts a state at a particular point in time
- Timeslice queries over a previous state requires an additional predicate for each temporal table
- What was Bob’s position at the beginning of 1997?
  
  ```sql
  SELECT JobTitle
  FROM Employee E, Incumbents I, Position P
  WHERE E.FirstName = 'Bob'
  AND E.SSN = I.SSN AND I.PCN = P.PCN
  AND IFromDate <= '1997-01-01' AND '1997-01-01' < IToDate
  ```

Sequenced Queries

- Queries whose result is a valid-time table
- Use sequenced variants of basic operations
  - Selection, projection, union, sorting, join, difference, and duplicate elimination
- **Sequenced selection**: no change is necessary
- Who makes or has made more than 50K annually
  
  ```sql
  SELECT *
  FROM Salary
  WHERE Amount > 50000
  ```

- **Sequenced projection**: include the timestamp columns in the select list
- List the social security numbers of current and past employees
  
  ```sql
  SELECT SSN, FromDate, ToDate
  FROM Salary
  ```

- Duplications resulting from the projection are retained
- To eliminate them coalescing is needed (see next)
Coalescing while Removing Duplicates

SELECT DISTINCT F.SSN, F.FromDate, L.ToDate
FROM Salary F, Salary L
WHERE F.FromDate < L.ToDate
AND F.SSN = L.SSN
AND NOT EXISTS ( SELECT * FROM Salary AS M
WHERE M.SSN = F.SSN
AND F.FromDate < M.FromDate AND M.FromDate <= L.ToDate
AND NOT EXISTS ( SELECT * FROM Salary AS T1
WHERE T1.SSN = F.SSN AND
AND T1.FromDate < M.FromDate AND M.FromDate <= T1.ToDate ) )
AND NOT EXISTS ( SELECT * FROM Salary AS T2
WHERE T2.SSN = F.SSN AND
AND ( (T2.FromDate < F.FromDate AND F.FromDate <= T2.ToDate)
OR (T2.FromDate <= L.ToDate AND L.ToDate < T2.ToDate) ) )

Sequenced Sort

- Requires the result to be ordered at each point in time
- This can be accomplished by appending the start and end time columns in the ORDER BY clause
- Sequenced sort Incumbents on the position code (first version)
  ```sql
  SELECT *
  FROM Incumbents
  ORDER BY PCN, FromDate, ToDate
  ```
- Sequenced sorting can also be accomplished by omitting the timestamp columns
  ```sql
  SELECT *
  FROM Incumbents
  ORDER BY PCN
  ```
Sequenced Union

- A `UNION ALL` (retaining duplicates) over temporal tables is automatically sequenced if the timestamp columns are kept.
- Who makes or has made annually more than 50,000 or less than 10,000?
  
  ```
  SELECT *
  FROM Salary
  WHERE Amount > 50000
  UNION ALL
  SELECT *
  FROM Salary
  WHERE Amount < 10000
  ```

- A `UNION` without `ALL` eliminates duplicates but is difficult to express in SQL (see later).

Sequenced Join (1)

- Example: determine the salary and position history for each employee.
- Implies a sequenced join between `Salary` and `Incumbents`.
- It is supposed that there are no duplicate rows in the tables: at each point in time an employee has one salary and one position.
- In SQL a sequenced join requires four select statements and complex inequality predicates.
- The following code does not generate duplicates.
- For this reason `UNION ALL` is used which is more efficient than `UNION`, which does a lot of work for remove the nonoccurring duplicates.
**Sequenced Join (2)**

```sql
SELECT S.SSN, Amount, PCN, S.FromDate, S.ToDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND I.FromDate < S.FromDate AND S.ToDate <= I.ToDate
UNION ALL
SELECT S.SSN, Amount, PCN, S.FromDate, I.ToDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND S.FromDate >= I.FromDate
AND S.FromDate < I.ToDate AND I.ToDate < S.ToDate
UNION ALL
SELECT S.SSN, Amount, PCN, I.FromDate, S.ToDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND I.FromDate >= S.FromDate
AND I.FromDate < S.ToDate AND S.ToDate < I.ToDate
UNION ALL
SELECT S.SSN, Amount, PCN, I.FromDate, I.ToDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND I.FromDate > S.FromDate AND I.ToDate < S.ToDate
```

**Sequenced Join using CASE**

```sql
SELECT S.SSN, Amount, PCN,
CASE WHEN S.FromDate > I.FromDate
    THEN S.FromDate ELSE I.FromDate
END AS StartDate,
CASE WHEN S.ToDate > I.ToDate
    THEN I.ToDate ELSE S.ToDate
END AS EndDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND (CASE WHEN S.FromDate > I.FromDate
    THEN S.FromDate ELSE I.FromDate
END) < (CASE WHEN S.ToDate > I.ToDate
    THEN I.ToDate ELSE S.ToDate
END)
```

- CASE allows to write this query in a single statement
- First CASE simulates a `maxDate` function of the two arguments, the second one a `minDate` function
- Condition in the WHERE ensures that the period of validity is well formed
Sequenced Join using Functions: SQL Server Example

```sql
create function minDate(@one smalldatetime, @two smalldatetime) returns smalldatetime as
begin
    return CASE WHEN @one < @two then @one else @two end
end

create function maxDate(@one smalldatetime, @two smalldatetime) returns smalldatetime as
begin
    return CASE WHEN @one > @two then @one else @two end
end

SELECT S.SSN, Amount, PCN,
    maxDate(S.FromDate,I.FromDate) AS StartDate,
    minDate(S.ToDate,I.ToDate) AS EndDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND maxDate(S.FromDate,I.FromDate) < minDate(S.ToDate,I.ToDate)
```

Difference

- Implemented in SQL with `EXCEPT`, `NOT EXISTS`, or `NOT IN`
- List the employees who are department heads (`PCN=1234`) but are not also professors (`PCN=5555`)
- Nontemporal version
  ```sql
  SELECT SSN
  FROM Incumbents I1
  WHERE I1.PCN = 1234
  AND NOT EXISTS ( SELECT * FROM Incumbents I2
                   WHERE I1.SSN = I2.SSN AND I2.PCN = 5555 )
  ```
- Using `EXCEPT`
  ```sql
  SELECT SSN
  FROM Incumbents
  WHERE PCN = 1234
  EXCEPT
  SELECT SSN
  FROM Incumbents
  WHERE PCN = 5555
  ```
Sequenced Difference (1)

Result

Case 1

Department head

Professor

Case 2

Department head

Professor

Case 3

Professor

Department head

Case 4

Department head

◆ Sequenced version: Identify when the department heads were not professors
◆ Four possible cases should be taken into account
◆ Each of them requires a separate SELECT statement

Sequenced Difference (2)

◆ List the employees who are or were department heads (PCN=1234) but not also professors (PCN=5555)

SELECT I1.SSN, I1.FromDate, I2.FromDate AS ToDate
FROM Incumbents I1, Incumbents I2
WHERE I1.PCN = 1234 AND I2.PCN = 5555 AND I1.SSN = I2.SSN
AND I1.FromDate < I2.FromDate AND I2.FromDate < I1.ToDate
AND NOT EXISTS ( SELECT * FROM Incumbents I3
WHERE I1.SSN = I3.SSN AND I3.PCN = 5555
AND I1.FromDate < I3.ToDate AND I3.FromDate < I2.ToDate )
UNION
SELECT I1.SSN, I2.ToDate AS FromDate, I1.ToDate
FROM Incumbents I1, Incumbents I2
WHERE I1.PCN = 1234 AND I2.PCN = 5555 AND I1.SSN = I2.SSN
AND I1.FromDate < I2.ToDate AND I2.ToDate < I1.ToDate
AND NOT EXISTS ( SELECT * FROM Incumbents I3
WHERE I1.SSN = I3.SSN AND I3.PCN = 5555
AND I2.ToDate < I3.ToDate AND I3.FromDate < I1.ToDate )
UNION
...
Sequenced Difference (3)

```
... 
SELECT I1.SSN, I2.ToDate AS FromDate, I3.FromDate AS ToDate
FROM Incumbents I1, Incumbents I2, Incumbents I3
WHERE I1.PCN = 1234 AND I2.PCN = 5555 AND I3.PCN = 5555
AND I1.SSN = I2.SSN AND I1.SSN = I3.SSN
AND I2.ToDate < I3.FromDate
AND I1.FromDate < I2.ToDate
AND I3.FromDate < I1.ToDate
AND NOT EXISTS ( SELECT * FROM Incumbents I4
WHERE I1.SSN = I4.SSN AND I4.PCN = 5555
AND I2.ToDate < I4.ToDate AND I4.FromDate < I3.FromDate )
UNION
SELECT SSN, FromDate, ToDate
FROM Incumbents I1
WHERE I1.PCN = 1234
AND NOT EXISTS ( SELECT * FROM Incumbents I4
WHERE I1.SSN = I4.SSN AND I4.PCN = 5555
AND I1.FromDate < I4.ToDate AND I4.FromDate < I1.ToDate )
```

Nonsequenced Variants

- Nonsequenced operators (selection, join, ...) are straightforward
  - They ignore the time-varying nature of tables
- List all the salaries, past and present, of employees who had been lecturer at some time
  ```
  SELECT Amount
  FROM Incumbents I, Position P, Salary S
  WHERE I.SSN = S.SSN AND I.PCN = P.PCN
  AND JobTitle = 'Lecturer'
  ```
- When did employees receive raises?
  ```
  SELECT S2.SSN, S2.FromDate AS RaiseDate
  FROM Salary S1, Salary S2
  WHERE S2.Amount > S1.Amount
  AND S1.SSN = S2.SSN
  AND S1.ToDate = S2FromDate
  ```
Eliminating Duplicates

- Remove nonsequenced duplicates from **Incumbents**
  
  ```sql
  SELECT DISTINCT *
  FROM Incumbents
  ```

- Remove value-equivalent rows from **Incumbents**
  
  ```sql
  SELECT DISTINCT SSN, PCN
  FROM Incumbents
  ```

- Remove current duplicates from **Incumbents**
  
  ```sql
  SELECT DISTINCT SSN, PCN
  FROM Incumbents
  WHERE ToDate = '3000-01-01'
  ```

Sequenced Aggregation Functions

<table>
<thead>
<tr>
<th>Affiliation</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>DNumber</td>
<td>FromDate</td>
<td>ToDate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Salary</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Amount</td>
<td>FromDate</td>
<td>ToDate</td>
</tr>
</tbody>
</table>

- SQL provides aggregation functions: **COUNT, MIN, MAX, AVG, ...**

- List the maximum salary: non-temporal version
  
  ```sql
  SELECT MAX(Amount)
  FROM Salary
  ```

- List by department the maximum salary: non-temporal version
  
  ```sql
  SELECT DNumber, MAX(Amount)
  FROM Affiliation A, Salary S
  WHERE A.SSN = S.SSN
  GROUP BY DNumber
  ```
**Maximum Salary: Temporal Version (1)**

<table>
<thead>
<tr>
<th></th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>35</td>
<td>35</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>35</td>
<td>35</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>35</td>
<td>35</td>
<td>30</td>
</tr>
</tbody>
</table>

**First step:** Compute the periods on which a maximum must be calculated

CREATE VIEW SalChanges(Day) AS
SELECT DISTINCT FromDate FROM Salary
UNION
SELECT DISTINCT ToDate FROM Salary
CREATE VIEW SalPeriods(FromDate, ToDate) AS
SELECT P1.Day, P2.Day
FROM SalChanges P1, SalChanges P2
WHERE P1.Day < P2.Day
AND NOT EXISTS ( SELECT * FROM SalChanges P3

**Second step:** Compute the maximum salary for these periods

CREATE VIEW TempMax(MaxSalary, FromDate, ToDate) AS
SELECT MAX(E.Amount), I.FromDate, I.ToDate
FROM Salary E, SalPeriods I
WHERE E.FromDate <= I.FromDate AND I.ToDate <= E.ToDate
GROUP BY I.FromDate, I.ToDate

**Third step:** Coalesce the above view (as seen before)
Number of Employees: Temporal Version

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>E2</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>E3</td>
<td>30</td>
<td>35</td>
</tr>
</tbody>
</table>

COUNT 1 2 3 3 3 2 0 2 1

**Second step:** Compute the number of employees for these periods

```sql
CREATE VIEW TempCount(NbEmp, FromDate, ToDate) AS
SELECT COUNT(*), P.FromDate, P.ToDate
FROM Salary S, SalPeriods P
WHERE S.FromDate<=P.FromDate AND P.ToDate<=S.ToDate
GROUP BY P.FromDate, P.ToDate
UNION ALL
SELECT 0, P.FromDate, P.ToDate
FROM SalPeriods P
WHERE NOT EXISTS (SELECT * FROM Salary S
WHERE S.FromDate<=P.FromDate AND P.ToDate<=S.ToDate)
```

**Third step:** Coalesce the above view (as seen before)

Maximum Salary by Department: Temporal Version (1)

<table>
<thead>
<tr>
<th></th>
<th>D1</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>E2</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>E3</td>
<td>30</td>
<td>35</td>
</tr>
</tbody>
</table>

MAX(D1) 20 25 35 35
MAX(D2) 25 30 30 30

**Hypothesis:** Employees have salary only while they are affiliated to a department
Maximum Salary by Department: Temporal Version (2)

◆ **First step:** Compute by department the periods on which a maximum must be calculated

```sql
CREATE VIEW Aff_Sal(DNumber, Amount, FromDate, ToDate) AS
    SELECT DISTINCT A.DNumber, S.Amount,
               maxDate(S.FromDate, A.FromDate), minDate(S.ToDate, A.ToDate)
    FROM Affiliation A, Salary S
    WHERE A.SSN=S.SSN
    AND maxDate(S.FromDate, A.FromDate) < minDate(S.ToDate, A.ToDate)

CREATE VIEW SalChanges(DNumber, Day) AS
    SELECT DISTINCT DNumber, FromDate FROM Aff_Sal
    UNION
    SELECT DISTINCT DNumber, ToDate FROM Aff_Sal

CREATE VIEW SalPeriods(DNumber, FromDate, ToDate) AS
    SELECT P1.DNumber, P1.Day, P2.Day
    FROM SalChanges P1, SalChanges P2
    WHERE P1.DNumber = P2.DNumber AND P1.Day < P2.Day
    AND NOT EXISTS ( SELECT * FROM SalChanges P3
                     WHERE P1.DNumber = P3.DNumber AND P1.Day < P3.Day
                     AND P3.Day < P2.Day )
```

Maximum Salary by Department: Temporal Version (3)

◆ **Second step:** Compute the maximum salary for these periods

```sql
CREATE VIEW TempMaxDep(DNumber, MaxSalary, FromDate, ToDate) AS
    SELECT P.DNumber, MAX(Amount), P.FromDate, P.ToDate
    FROM Aff_Sal A, SalPeriods P
    WHERE A.DNumber = P.DNumber
    AND A.FromDate <= P.FromDate AND P.ToDate <= A.ToDate
    GROUP BY P.DNumber, P.FromDate, P.ToDate

CREATE VIEW MaxDep(DNumber, MaxSalary, FromDate) AS
    SELECT DNumber, MaxSalary, FromDate FROM TempMaxDep
```

◆ **Third step:** Coalesce the above view (as seen before)
**Sequenced Division**

<table>
<thead>
<tr>
<th>Affiliation</th>
<th>SSN</th>
<th>DNumber</th>
<th>FromDate</th>
<th>ToDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controls</td>
<td>PNumber</td>
<td>DNumber</td>
<td>FromDate</td>
<td>ToDate</td>
</tr>
<tr>
<td>WorksOn</td>
<td>SSN</td>
<td>PNumber</td>
<td>FromDate</td>
<td>ToDate</td>
</tr>
</tbody>
</table>

- Implemented in SQL with two nested `NOT EXISTS`
- List the employees that work in all projects of the department to which they are affiliated: non-temporal version

```sql
SELECT SSN
FROM Affiliation A
WHERE NOT EXISTS (
    SELECT * FROM Controls C
    WHERE A.DNumber = C.DNumber AND NOT EXISTS (
        SELECT * FROM WorksOn W
        WHERE C.PNumber = W.PNumber AND A.SSN = W.SSN )
)
```

**Sequenced Division: Case 1 (1)**

- Only `WorksOn` is temporal
- **First step:** Construct the periods on which the division must be computed

```
CREATE VIEW ProjChangesC1(SSN,Day) AS
    SELECT SSN,FromDate FROM WorksOn
UNION
    SELECT SSN,ToDate FROM WorksOn
CREATE VIEW ProjPeriodsC1(SSN,FromDate,ToDate) AS
    SELECT P1.SSN,P1.Day,P2.Day
    FROM ProjChangesC1 P1, ProjChangesC1 P2
    WHERE P1.SSN=P2.SSN AND P1.Day<P2.Day AND NOT EXISTS (
        SELECT * FROM ProjChangesC2 P3
```

```
CREATE VIEW ProjChangesC1(SSN,Day) AS
    SELECT SSN,FromDate FROM WorksOn
UNION
    SELECT SSN,ToDate FROM WorksOn
CREATE VIEW ProjPeriodsC1(SSN,FromDate,ToDate) AS
    SELECT P1.SSN,P1.Day,P2.Day
    FROM ProjChangesC1 P1, ProjChangesC1 P2
    WHERE P1.SSN=P2.SSN AND P1.Day<P2.Day AND NOT EXISTS (
        SELECT * FROM ProjChangesC2 P3
```

```
CREATE VIEW ProjChangesC1(SSN,Day) AS
    SELECT SSN,FromDate FROM WorksOn
UNION
    SELECT SSN,ToDate FROM WorksOn
CREATE VIEW ProjPeriodsC1(SSN,FromDate,ToDate) AS
    SELECT P1.SSN,P1.Day,P2.Day
    FROM ProjChangesC1 P1, ProjChangesC1 P2
    WHERE P1.SSN=P2.SSN AND P1.Day<P2.Day AND NOT EXISTS (
        SELECT * FROM ProjChangesC2 P3
```
Sequenced Division: Case 1 (2)

**Second step**: Compute the division

```sql
CREATE VIEW TempUnivQuantC1(SSN, FromDate, ToDate) AS
SELECT DISTINCT P.SSN, P.FromDate, P.ToDate
FROM ProjPeriodsC1 P, Affiliation A
WHERE P.SSN = A.SSN AND NOT EXISTS (SELECT * FROM Controls C
WHERE A.DNumber = C.DNumber AND NOT EXISTS (SELECT * FROM WorksOn W
WHERE C.PNumber = W.PNumber AND P.SSN = W.SSN
AND W.FromDate <= P.FromDate AND PToDate <= W.ToDate ) )
```

**Third step**: Coalesce the above view

Sequenced Division: Case 2 (1)

**Only Controls and WorksOn are temporal**

**Employees may work in projects controlled by departments different from the department to which they are affiliated**

**First step**: Construct the periods on which the division must be computed

```
C1    D,P1
C2    D,P2
W1    E,P1                      Affiliation(E,D)
W2    E,P2
Result  ✓ ✓ ✗ ✓ ✓ ✓ ✗ ✗
```
Sequenced Division: Case 2 (2)

CREATE VIEW ProjChangesC2(SSN,Day) AS
SELECT SSN,FromDate
FROM Affiliation A, Controls C
WHERE A.DNumber=C.DNumber
UNION
SELECT SSN,ToDate
FROM Affiliation A, Controls C
WHERE A.DNumber=C.DNumber
UNION
SELECT SSN,FromDate FROM WorksOn
UNION
SELECT SSN,ToDate FROM WorksOn
CREATE VIEW ProjPeriodsC2(SSN,FromDate,ToDate) AS
SELECT P1.SSN,P1.Day,P2.Day
FROM ProjChangesC2 P1, ProjChangesC2 P2
WHERE P1.SSN=P2.SSN AND P1.Day<P2.Day AND NOT EXISTS (
SELECT * FROM ProjChangesC2 P3

Sequenced Division: Case 2 (3)

◆ Second step: Compute the division of these periods
CREATE VIEW TempUnivC2(SSN,FromDate,ToDate) AS
SELECT DISTINCT P.SSN,P.FromDate,P.ToDate
FROM ProjPeriodsC2 P, Affiliation A
WHERE P.SSN=A.SSN AND NOT EXISTS ( 
SELECT * FROM Controls C
WHERE A.DNumber=C.DNumber AND C.FromDate<=P.FromDate
AND P.ToDate<=C.ToDate AND NOT EXISTS ( 
SELECT * FROM WorksOn W
WHERE C.PNumber=W.PNumber AND P.SSN=W.SSN
AND W.FromDate<=P.FromDate AND P.ToDate<=W.ToDate ) )

◆ Third step: Coalesce the above view
Sequenced Division: Case 3 (1)

- Only **Affiliation** and **WorksOn** are temporal
- Employees may work in projects controlled by departments different from the department to which they are affiliated
- **First step:** Construct the periods on which the division must be computed

```
A  E,D
W1  E,P1  Controls(D,P1)
  E,P2  Controls(D,P2)
W2
ProjChanges
```

Sequenced Division: Case 3 (2)

```
CREATE VIEW Aff_WO(SSN, DNumber, PNumber, FromDate, ToDate) AS
SELECT DISTINCT A.SSN, A.DNumber, W.PNumber,
    maxDate(A.FromDate,W.FromDate), minDate(A.ToDate,W.ToDate)
FROM Affiliation A, WorksOn W
WHERE A.SSN=W.SSN
    AND maxDate(A.FromDate,W.FromDate) < minDate(A.ToDate,W.ToDate)

CREATE VIEW ProjChangesC3(SSN, DNumber, Day) AS
SELECT SSN, DNumber, FromDate FROM Aff_WO UNION
SELECT SSN, DNumber, ToDate FROM Aff_WO UNION
SELECT SSN, DNumber, FromDate FROM Affiliation UNION
SELECT SSN, DNumber, ToDate FROM Affiliation

CREATE VIEW ProjPeriodsC3(SSN, DNumber, FromDate, ToDate) AS
SELECT P1.SSN, P1.DNumber, P1.Day, P2.Day
FROM ProjChangesC3 P1, ProjChangesC3 P2
WHERE P1.SSN = P2.SSN AND P1.DNumber = P2.DNumber
    AND P1.Day < P2.Day AND NOT EXISTS ( 
        SELECT * FROM ProjChangesC3 P3
        WHERE P1.SSN = P3.SSN AND P1.DNumber = P3.DNumber
```
Sequenced Division: Case 3 (3)

- **Second step**: Compute the division of these periods
  
  CREATE VIEW TempUnivQuant(SSN, FromDate, ToDate) AS
  SELECT DISTINCT P.SSN, P.FromDate, P.ToDate
  FROM ProjPeriodsC3 P
  WHERE NOT EXISTS (  
    SELECT * FROM Controls C
    WHERE P.DNumber=C.DNumber AND NOT EXISTS (  
      SELECT * FROM WorksOn W
      WHERE C.PNumber=W.PNumber AND P.SSN=W.SSN
      AND W.FromDate<=P.FromDate AND P.ToDate<=W.ToDate  )  )

- **Third step**: Coalesce the above view

Sequenced Division: Case 4 (1)

- **Affiliation, Controls, and WorksOn** are all temporal

- **First step**: Construct the periods on which the division must be computed

```
A    E,D
C1   D,P1
C2   D,P2
W1   E,P1
W2   E,P2
```

Result: ✓ ✓ ✓ ✓ ✓ ✓ ✓
Sequenced Division: Case 4 (2)

CREATE VIEW Aff_Cont(SSN, DNumber, PNumber, FromDate, ToDate) AS
SELECT DISTINCT A.SSN, A.DNumber, C.PNumber,
    maxDate(A.FromDate,C.FromDate), minDate(A.ToDate,C.ToDate)
FROM Affiliation A, Controls C WHERE A.DNumber=C.DNumber
AND maxDate(A.FromDate,C.FromDate) < minDate(A.ToDate,C.ToDate)

CREATE VIEW Aff_Cont_WO(SSN, DNumber, PNumber, FromDate, ToDate) AS
SELECT DISTINCT A.SSN, A.DNumber, W.PNumber,
    maxDate(A.FromDate,W.FromDate), minDate(A.ToDate,W.ToDate)
FROM Aff_Cont A, WorksOn W WHERE A.PNumber=W.PNumber AND A.SSN=W.SSN
AND maxDate(A.FromDate,W.FromDate) < minDate(A.ToDate,W.ToDate)

CREATE VIEW ProjChangesC4(SSN, DNumber, Day) AS
SELECT SSN, DNumber, FromDate FROM Aff_Cont UNION
SELECT SSN, DNumber, ToDate FROM Aff_Cont UNION
SELECT SSN, DNumber, FromDate FROM Aff_Cont_WO UNION
SELECT SSN, DNumber, ToDate FROM Aff_Cont_WO UNION
SELECT SSN, DNumber, FromDate FROM Affiliation UNION
SELECT SSN, DNumber, ToDate FROM Affiliation

CREATE VIEW ProjPeriodsC4(SSN, DNumber, FromDate, ToDate) AS
SELECT P1.SSN, P1.DNumber, P1.Day, P2.Day
FROM ProjChangesC4 P1, ProjChangesC4 P2 WHERE P1.SSN = P2.SSN
AND P1.DNumber = P2.DNumber AND P1.Day < P2.Day
AND NOT EXISTS ( SELECT * FROM ProjChangesC4 P3
    WHERE P1.SSN = P3.SSN AND P1.DNumber = P3.DNumber

Sequenced Division: Case 4 (3)

*Second step:* Compute the division of these periods

CREATE VIEW TempUnivQuant(SSN, FromDate, ToDate) AS
SELECT DISTINCT P.SSN, P.FromDate, P.ToDate
FROM ProjPeriodsC4 P
WHERE NOT EXISTS ( SELECT * FROM Controls C
    WHERE P.DNumber = C.DNumber AND C.FromDate <= P.FromDate
    AND P.ToDate <= C.ToDate AND NOT EXISTS ( SELECT * FROM WorksOn W
        WHERE C.PNumber = W.PNumber AND P.SSN=W.SSN
        AND W.FromDate <= P.FromDate AND P.ToDate <= W.ToDate ) )

*Third step:* Coalesce the above result
Temporal Databases: Topics

- Introduction
- Time Ontology
- Temporal Conceptual Modeling
- Manipulating Temporal Databases with SQL-92
- Temporal Support in Current DBMSs and in SQL 2011
- Summary

Temporal Support in Oracle

- Oracle 9i, released in 2001, included support for transaction time
- Flashback queries allow the application to access prior transaction-time states of their database; they are transaction timeslice queries
- Database modifications and conventional queries are temporally upward compatible
- Oracle 10g, released in 2006, extended flashback queries to retrieve all the versions of a row between two transaction times (a key-transaction-time-range query)
- It also allowed tables and databases to be rolled back to a previous transaction time, discarding all changes after that time
- Oracle 10g Workspace Manager includes the period data type, valid-time support, transaction-time support, bitemporal support, and support for sequenced primary keys, sequenced uniqueness, sequenced referential integrity, and sequenced selection and projection
- These facilities permit tracing of actions on data as well as the ability to perform database forensics
- Oracle 11g, released in 2007, does not rely on transient storage like the undo segments, it records changes in the Flashback Recovery Area
- Valid-time queries were also enhanced
Temporal Support in Teradata

- Teradata Database 13.10, released October 2010, introduced the period data type, valid-time support, transaction-time support, timeslices, temporal upward compatibility, sequenced primary key and temporal referential integrity constraints, nonsequenced queries, and sequenced projection and selection.
- Teradata Database 14, released February 29, 2012, adds capabilities to create a global picture of an organization’s business at any point in time.

Temporal Support in DB2

- IBM DB2 10, released in October 2010, includes the period data type, valid-time support (termed business time), transaction-time support (termed system time), timeslices, temporal upward compatibility, sequenced primary keys, and sequenced projection and selection.
Temporal Facilities in the SQL 2011

- **Application-time period tables** (essentially valid-time tables)
  - Have sequenced primary and foreign keys
  - Support single-table valid-time sequenced insertions, deletions, and updates
  - Nonsequenced valid-time queries are supported

- **System-versioned tables** (essentially transaction-time tables)
  - Have transaction-time current primary and foreign keys
  - Support transaction-time current insertions, deletions, and updates
  - Support transaction-time current and nonsequenced queries

- **System-versioned application-time period tables** (essentially bitemporal tables)
  - Support temporal queries and modifications of combinations of the valid-time and transaction-time variants

Temporal Support in the SQL Standard: A Short History

- First work started in July 1993 under the TSQL2 initiative led by Richard Snodgrass
- Definitive version of the TSQL2 Language Specification published in September 1994
- Then work to transfer some of the constructs and insights of TSQL2 into SQL3 started
- A new part to SQL3, termed SQL/Temporal, was accepted in January, 1995 as Part 7 of the SQL3 specification
- Discussions then commenced on adding valid-time and transaction-time support to SQL/Temporal. Two change proposals, ANSI-96-501 and ANSI-96-502, were unanimously accepted by ANSI and forwarded to ISO in early 1997
- Due to disagreements within the ISO committee, the project responsible for temporal support was canceled in 2001
- Concepts and constructs from SQL/Temporal were subsequently included in SQL:2011 and have been implemented in IBM DB2, Oracle, Teradata Database, and PolarLake
- Other products have included temporal support
**Brief Description of the SQL Standard (1)**

- ISO/IEC 9075, Database Language SQL is the dominant database language de-jure standard
- Multi-part standard with 9 Parts
  - Part 1 - Framework (SQL/Framework)
  - Part 2 - Foundation (SQL/Foundation)
  - Part 3 - Call-Level Interface (SQL/CLI)
  - Part 4 - Persistent Stored Modules (SQL/PSM)
  - Part 9 - Management of External Data (SQL/MED)
  - Part 10 - Object Language Bindings (SQL/OLB)
  - Part 11 - Information and Definition Schemas (SQL/Schemata)
  - Part 13 - SQL Routines and Types using the Java Programming Language (SQL/JRT)
  - Part 14 - XML-Related Specifications (SQL/XML)
- Parts 3, 9, 10, and 13 are currently inactive

**Brief Description of the SQL Standard (2)**

- Part 2 - SQL/Foundation: Largest and the most important part SQL
  - General-purpose programming constructs: Data types, expressions, predicates, etc.
  - Data definition: `CREATE/ALTER/DROP` of tables, views, constraints, triggers, stored procedures, stored functions, etc.
  - Query constructs: `SELECT`, joins, etc.
  - Data manipulation: `INSERT`, `UPDATE`, `MERGE`, `DELETE`, etc.
  - Access control: `GRANT`, `REVOKE`, etc.
  - Transaction control: `COMMIT`, `ROLLBACK`, etc.
  - Connection management: `CONNECT`, `DISCONNECT`, etc.
  - Session management: `SET SESSION` statement
  - Exception handling: `GET DIAGNOSTICS` statement
**Brief Description of the SQL Standard (3)**

- For conformance purpose, SQL is divided into a list of “features”, grouped under two categories:
  - Mandatory features
  - Optional features
- To claim conformance, an implementation must conform to all mandatory features
- An implementation may conform to any number of optional features
- Both are listed in Annex F of each part of the SQL standard
- SQL/Foundation:2008 specifies 164 mandatory features and 280 optional features
- SQL/Foundation:2011 added a total 34 new features, including
  - System-versioned tables
  - Application-time period tables

---

**Application-Time Period Tables**

- Contain a `PERIOD` clause (newly-introduced) with an user-defined period name
- Currently restricted to temporal periods only; may be relaxed in the future
- Must contain two additional columns, to store the start time and the end time of a period associated with the row
- Values of both start and end columns are set by the users
- Users can specify primary key/unique constraints to ensure that no two rows with the same key value have overlapping periods
- Users can specify referential constraints to ensure that the period of every child row is completely contained in the period of exactly one parent row or in the combined period of two or more consecutive parent rows
- Queries, inserts, updates and deletes on application-time period tables behave exactly like queries, inserts, updates and deletes on regular tables
- Additional syntax is provided on `UPDATE` and `DELETE` statements for partial period updates and deletes
Creating an Application-Time Period Table

CREATE TABLE employees
(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
departments (department_id, PERIOD dept_period));

- PERIOD clause automatically enforces the constraint end_date > start_date
- The name of the period can be any user-defined name
- The period starts on the start_date value and ends on the value just prior to end_date value
- This corresponds to the [closed, open) encoding of periods

Inserting Rows into an Application-Time Period Table (1)

- On an insertion, user provides the start and end time of the period for each row
- User-supplied time values can be either in the past, current, or in the future
- Example

```sql
INSERT INTO employees (emp_name, department_id, start_date, end_date)
('Tracy', 'K25', DATE '1996-01-01', DATE '1997-11-15')
```

- Periods are encoded as [closed, open)
Inserting Rows into an Application-Time Period Table (2)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

- Given the above table, the following `INSERT` will succeed

  ```sql
  INSERT INTO employees (emp_name, dept_id, start_date, end_date)
  ```

- The following `INSERT` will not, because of the inclusion of `emp_period WITHOUT OVERLAPS` in the primary key definition

  ```sql
  INSERT INTO employees (emp_name, dept_id, start_date, end_date)
  VALUES ('John', 'J13', DATE '1996-01-01', DATE '1996-12-31')
  ```

Updating Rows in an Application-Time Period Table (1)

- All rows can be potentially updated
- Users are allowed to update the start and end columns of the period associated with each row
- When a row from an application-time period table is updated using the regular `UPDATE` statements, the regular semantics apply
- Additional syntax is provided for `UPDATE` statements to specify the time period during which the update applies
- Only those rows that lie within the specified period are impacted
- May lead to row splits, i.e., update of a row may cause insertion of up to two rows to preserve the information for the periods that lie outside the specified period
- Users are not allowed to update the start and end columns of the period associated with each row under this option
Updating Rows in an Application-Time Period Table (2)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

Given the above table, the following `UPDATE`

```sql
UPDATE employees
SET dept_id = 'J15'
WHERE emp_name = 'John'
```

will lead the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

No changes to the period values

Updating Rows in an Application-Time Period Table (3)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

Given the above table, the following `UPDATE`

```sql
UPDATE employees
FOR PORTION OF emp_period FROM
DATE '1996-03-01' TO DATE '1996-07-01'
SET dept_id = 'M12'
WHERE emp_name = 'John'
```

will lead the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>15/11/1995</td>
<td>01/03/1996</td>
</tr>
<tr>
<td>John</td>
<td>M12</td>
<td>01/03/1996</td>
<td>01/07/1996</td>
</tr>
<tr>
<td>John</td>
<td>J15</td>
<td>01/07/1996</td>
<td>15/11/1996</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

Automatic row splitting: 1 update and 2 inserts
Deleting Rows from an Application-Time Period Table (1)

- All rows can be potentially deleted
- When a row from an application-time period table is deleted using the regular DELETE statements, the regular semantics apply
- Additional syntax is provided for DELETE statements to specify the time period during which the delete applies
- Only those rows that lie within the specified period are impacted
- May lead to row splits, i.e., delete of a row may cause insertion of up to two rows to preserve the information for the periods that lie outside the specified period

Given the above table, the following DELETE

`DELETE FROM employees FOR PORTION OF emp_period FROM
DATE '1996-08-01' TO DATE '1996-09-01'
WHERE emp_name = 'John'
`  

will lead the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>15/11/1995</td>
<td>01/03/1996</td>
</tr>
<tr>
<td>John</td>
<td>M12</td>
<td>01/03/1996</td>
<td>01/07/1996</td>
</tr>
<tr>
<td>John</td>
<td>J15</td>
<td>01/07/1996</td>
<td>15/11/1996</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

- Automatic row splitting: 1 delete and 2 inserts
Deleting Rows from an Application-Time Period Table (3)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>15/11/1995</td>
<td>01/03/1996</td>
</tr>
<tr>
<td>John</td>
<td>M12</td>
<td>01/03/1996</td>
<td>01/07/1996</td>
</tr>
<tr>
<td>John</td>
<td>J15</td>
<td>01/07/1996</td>
<td>01/08/1996</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

Given the above table, the following DELETE

```
DELETE FROM employees
WHERE emp_name = 'John'
```

will lead the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>15/11/1997</td>
</tr>
</tbody>
</table>

All rows pertaining to John are deleted

Querying an Application-Time Period Table (1)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>31/03/2000</td>
</tr>
</tbody>
</table>

Existing syntax for querying regular tables is applicable to application-time period tables also

Which department was John in on Dec. 1, 1997?

```
SELECT dept_id
FROM employees
WHERE emp_name = 'John' AND start_date <= DATE '1997-12-01'
AND end_date > DATE '1997-12-01'
```

Answer: J13
### Querying an Application-Time Period Table (2)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>31/03/2000</td>
</tr>
</tbody>
</table>

- Which department is John in currently?
  ```sql
  SELECT dept_id
  FROM employees
  WHERE emp_name = 'John' AND start_date <= CURRENT_DATE
  AND end_date > CURRENT_DATE;
  ```
- Answer: M24

---

### Querying an Application-Time Period Table (3)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>01/01/1996</td>
<td>31/03/2000</td>
</tr>
</tbody>
</table>

- How many departments has John worked in since Jan. 1, 1996?
  ```sql
  SELECT count(distinct dept_id)
  FROM employees
  WHERE emp_name = 'John' AND start_date <= CURRENT_DATE
  AND end_date > DATE '1996-01-01';
  ```
- Answer: 2
Benefits of Application-Time Period Tables

- Most business data is time sensitive, i.e., need to track the time period during when a data item is deemed valid or effective from the business point of view
- Database systems today offer no support for
  - Associating user-maintained time periods with rows
  - Enforcing constraints such as “an employee can be in only one department in any given period”
- Updating/deleting a row for a part of its validity period
- Currently, applications take on the responsibility for managing such requirements
- Major issues
  - Complexity of code
  - Poor performance
- Use of application-time period tables provides
  - Significant simplification of application code
  - Significant improvement in performance
  - Transparent to legacy applications

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 columns, to store the start time and the 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 for these columns
- 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
- All constraints are enforced on current system rows only
Creating a System-Versioned Table

CREATE TABLE employees
(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 departments (dept_id);
) WITH SYSTEM VERSIONING;

- PERIOD clause automatically enforces the constraint system_end > system_start
- The name of the period must be SYSTEM_TIME
- The period starts on the system_start value and ends on the value just prior to system_end value
- This corresponds to the [closed, open) model of periods

Inserting Rows into a System-Versioned Table

- When a row is inserted into a system-versioned table, the SQL-implementation sets the start time to the transaction time and the end time to the largest timestamp value
- All rows inserted in a transaction will get the same values for the start and end columns
- The following INSERT executed at timestamp 15/11/1995
  INSERT INTO emp (emp_name, dept_id)
  VALUES ('John', 'J13'), ('Tracy','K25')
leads to the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J13</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>

- Values of system_start and system_end are set by DBMS
- N.B. Any date components of system_start and system_end values are shown for simplifying display
Updating Rows in a System-Versioned Table

- When a row from a system-versioned table is updated, the SQL-implementation inserts the “old” version of the row into the table before updating the row.
- SQL-implementation sets the end time of the old row and the start time of the updated row to the transaction time.
- Users are not allowed to update the start and end columns.
- The following UPDATE executed at 31/01/1998

```
UPDATE emp
SET dept_id = 'M24'
WHERE emp_name = 'John'
```

leads to the following table:

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>

Deleting Rows from a System-Versioned Table

- When a row from a system-versioned table is deleted, the SQL-implementation does not actually delete the row; it simply sets its end time to the transaction time.
- The following DELETE executed on 31/03/2000

```
DELETE FROM emp
WHERE emp_name = 'Tracy'
```

leads to the following table:

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/03/2000</td>
</tr>
</tbody>
</table>
Querying System-Versioned Tables (1)

- Existing syntax for querying regular tables is applicable to system-versioned tables also.
- Additional syntax is provided for expressing queries involving system-versioned tables in a more succinct manner:
  - FOR SYSTEM_TIME AS OF <datetime value expression>
  - FOR SYSTEM_TIME BETWEEN <datetime value expression 1> AND <datetime value expression 2>
  - FOR SYSTEM_TIME FROM <datetime value expression 1> TO <datetime value expression 2>

Querying System-Versioned Tables (2)

```
<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/03/2000</td>
</tr>
</tbody>
</table>
```

- Which department was John in on Dec. 1, 1997?

```sql
SELECT Dept
FROM employees
FOR SYSTEM_TIME AS OF DATE '1997-12-01'
WHERE emp_name = 'John'
```

- Answer: J13
Querying System-Versioned Tables (3)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/03/2000</td>
</tr>
</tbody>
</table>

- Which department is John in currently?
  ```
  SELECT Dept
  FROM employees
  WHERE emp_name = 'John'
  ```
- Answer: M24
- If AS OF clause is not specified, only current system rows are returned
  - `FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP` is the default

Querying System-Versioned Tables (4)

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M24</td>
<td>31/01/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/03/2000</td>
</tr>
</tbody>
</table>

- How many departments has John worked in since Jan. 1, 1996?
  ```
  SELECT count(distinct dept_id)
  FROM employees
  FOR SYSTEM_TIME BETWEEN DATE '1996-01-01' AND CURRENT_DATE
  WHERE emp_name = 'John'
  ```
- Answer: 2
Benefits of System-Versioned Tables

- Today’s database systems focus mainly on managing current data; they provide almost no support for managing historical data.
- Some applications have an inherent need for preserving old data. Examples: job histories, salary histories, account histories, etc.
- Regulatory and compliance laws require keeping old data around for certain length of time.
- Currently, applications take on the responsibility for preserving old data.

Major issues
- Complexity of code
- Poor performance

System-versioned tables provides
- Significant simplification of application code
- Significant improvement in performance
- Transparent to legacy applications

System-Versioned Application-Time Period Tables

- A table that is both an application-time period table and a system-versioned table.
- Such a table supports features of both application-time period tables and system-versioned tables.
- Creating a system-versioned application-time period table

```sql
CREATE TABLE employees
(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
departments (department_id, PERIOD department_period)
) WITH SYSTEM VERSIONING;
```
Insert

- On 11/01/1995, employees table was updated to show that John and Tracy will be joining the departments J13 and K25, respectively, starting from 15/11/1995

  ```
  INSERT INTO employees (emp_name, dept_id, start_date, end_date)
  VALUES ('John', 'J13', DATE '1995-11-15', DATE '9999-12-31'),
  ('Tracy', 'K25', DATE '1995-11-15', DATE '9999-12-31')
  ```

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J13</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>

- `system_start` and `system_end` values are set by the system

- N.B. `DATE` type is used in examples instead of `TIMESTAMP` type to simplify display

Update

- Current state of the table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J13</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>

- On 11/10/1995, it was discovered that John was assigned to the wrong department; it was changed to department J15 on that day

  ```
  UPDATE employees
  SET dept_id = 'J15'
  WHERE emp_name = 'John'
  ```

- This leads to the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/10/1995</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>
Partial Period Update

- Current state of the table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/10/1995</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>

- On 15/12/1997, John is loaned to department M12 starting from 01/01/1998 to 01/07/1998

UPDATE employees FOR PORTION OF emp_period
FROM DATE '1998-01-01' TO DATE '1998-07-01'
SET dept_id = 'M12' WHERE emp_name = 'John'

- This leads to the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>01/07/1998</td>
<td>31/12/9999</td>
<td>15/12/1997</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>John</td>
<td>M12</td>
<td>01/01/1998</td>
<td>01/07/1998</td>
<td>15/12/1997</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>John</td>
<td>J15</td>
<td>15/11/1995</td>
<td>01/01/1998</td>
<td>15/12/1997</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>

Partial Period Delete

- On 15/12/1998, John is approved for a leave of absence from 1/1/1999 to 1/1/2000

DELETE FROM employees
FOR PORTION OF emp_period FROM DATE '1999-01-01' TO DATE '2000-01-01'
WHERE emp_name = 'John'

- This leads to the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>01/01/2000</td>
<td>31/12/9999</td>
<td>15/12/1998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>John</td>
<td>J15</td>
<td>01/07/1998</td>
<td>01/01/1999</td>
<td>15/12/9998</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>John</td>
<td>M12</td>
<td>01/01/1998</td>
<td>01/07/1998</td>
<td>15/12/1997</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>John</td>
<td>J15</td>
<td>15/11/1995</td>
<td>01/01/1998</td>
<td>15/12/1997</td>
<td>31/12/9999</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>
Delete

◆ On 1/6/2000, John resigns from the company
  
  DELETE FROM employees
  WHERE emp_name = 'John'

◆ This leads to the following table

<table>
<thead>
<tr>
<th>emp_name</th>
<th>dept_id</th>
<th>start_date</th>
<th>end_date</th>
<th>system_start</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>J15</td>
<td>01/01/2000</td>
<td>31/12/9999</td>
<td>15/12/1998</td>
<td>01/06/2000</td>
</tr>
<tr>
<td>John</td>
<td>J15</td>
<td>01/07/1998</td>
<td>01/01/1999</td>
<td>15/12/1998</td>
<td>01/06/2000</td>
</tr>
<tr>
<td>John</td>
<td>M12</td>
<td>01/01/1998</td>
<td>01/07/1998</td>
<td>15/12/1997</td>
<td>01/06/2000</td>
</tr>
<tr>
<td>Tracy</td>
<td>K25</td>
<td>15/11/1995</td>
<td>31/12/9999</td>
<td>11/01/1995</td>
<td>31/12/9999</td>
</tr>
</tbody>
</table>

Temporal Databases: Conclusion

◆ Temporal information is ubiquitous in every application domain

◆ Such information should be included in the overall software lifecycle: from design to implementation

◆ Necessity of a temporal conceptual model for discussing requirements with users

◆ Manipulating temporal information in standard SQL is
  
  • Very difficult to program
  • Very inefficient

◆ Native temporal capabilities are needed in DBMSs

◆ Recent SQL standard has introduced such capabilities after more than a decade of debates

◆ Such capabilities have still to be implemented in the different platforms

◆ Data warehouses have included temporal capabilities since their beginning a few decades ago

◆ Temporal capabilities are usually combined with spatial capabilities ⇒ spatio-temporal databases