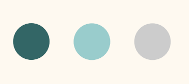


An overview of Temporal DBs

Letizia Tanca
(from various resources on the Web (*), and with the kind support of Rosalba Rossato)

(*) see acknowledgements in the last slide

Research area in temporal databases

Aims at:

- Characterizing the semantics of temporal data
- Providing expressive and efficient ways to:
  - Model
  - Store
  - Query

  temporal data

Application examples

- **Academic**: Transcripts record courses taken in previous and the current semester or term and grades for previous courses
- **Accounting**: What bills were sent out and when, what payments were received and when?
  - Delinquent accounts, cash flow over time
  - Money-management software e.g., account balance over time.
- **Budgets**: Previous and projected budgets, multi-quarter or multi-year budgets

Application examples (cont.)

- **Data Warehousing**: Historical trend analysis for decision support
- **Financial**: Stock market data
- **Audit**: why were financial decisions made, and with what information available?
- **GIS**: Geographic Information Systems
  - Land use over time: boundary of parcels changeover time, as parcels get partitioned and merged.
  - Title searches
- **Law**: Which law was in effect at each point in time, and what time periods did that law cover?

Application examples (cont.)

- **Medical records**: Patient records, drug regimes, lab tests.Tracking course of disease
- **Process control**: Timestamping sensor findings, previsions for future readings, future scenarios
- **Capacity planning** for roads and utilities. Configuring new routes, ensuring high utilization
- **Project scheduling**: Milestones, task assignments
- **Reservation systems:** airlines, hotels, trains.
- **Scientific**: Timestamping satellite images. Dating archeological finds

Temporal DBs Applications: Conclusion

- It is difficult to identify applications that do not involve the management of temporal data.
- These applications would benefit from built-in, knowledge independent temporal support
- Main benefits:
  - More efficient application development
  - Potential increase in performance (general-purpose optimization)

Time Datatype in SQL-2

- DATE: four digits for the year and two for month and day. Multiple formats allowed:
  - E.g., 2001-12-08 or 12/08/2001 or 12.08.2001
  - ISO, USA, EUR, JIS representations supported---DBA selects which one is used in specific system.
  - Internal representation is the same, independent of external ones. Basically an 8-byte string
- TIME: 2 digits for hour, 2 for minutes, and 2 for seconds, plus optional fractional digits (system dependent). E.g., 13:50:00, 13:50, 1:50 PM denote the same time.

Internal Representation (DB2)

- **A *date* is a three-part value (*year, month*, and *day*). The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to *x*, where *x* depends on the month.**
- **The internal representation of a date is a string of 4 bytes. Each byte consists of *2 packed decimal* digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.**
- **The length of a DATE column, as described in the SQLDA, is 10 bytes, which is the appropriate length for a character string representation of the value.**
- **A time is a three-part value (*hour, minute*, and *second*) designating a time of day under a 24-hour clock. The range of the hour part is 0 to 24, while the range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications will be zero (from then on "TIME 0" starts)**
- **The internal representation of a time is a string of 3 bytes. Each byte is 2 packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.**
- **The length of a TIME column, as described in the SQLDA, is 8 bytes, which is the appropriate length for a character string representation of the value.**

Motivating example (1)

- A relation storing information about employees and their assignment to departments

  Employee(Name,Salary,Department)
- It is easy to know the salary of an employee

```
SELECT Salary
FROM    Employee
WHERE  Name='John'
```

Motivating example (2)

- A certain kind of time information is not critical: e.g. add the date of birth

  Employee(Name,Salary,Department,
  DateofBirth DATE)
- It is easy to know the date of birth of an employee

```
SELECT     DateofBirth
FROM       Employee
WHERE      Name='John'
```

- Each employee has only one date of birth

When is a TDB useful

- We want to keep the employment history
  Employee(Name,Salary,Department, DateofBirth DATE, Start DATE, End DATE)
- This time unenvisaged consequences may happen

Name	Salary	Dept	DateB	Start	End
John	60.000	Shipping	9/9/65	1/1/95	1/6/95
John	70.000	Shipping	9/9/65	1/6/95	1/10/95
John	70.000	Loading	9/9/65	1/10/95	1/2/96
John	70.000	Research	9/9/65	1/2/96	1/1/97

Example: Determine the Salary

- To know the employee's **current** salary /just like the birthdate case):

```
SELECT Salary
FROM Employee
WHERE Name ='John'
      AND Start <= CURRENT_TIMESTAMP
      AND CURRENT_TIMESTAMP <= End
```

Determine the Salary (2)

- We want to determine the **salary history**
- Result: for each employee, the maximal intervals of each salary

Name	Salary	Start	End
John	60.000	1/1/95	1/6/95
John	70.000	1/6/95	1/1/97

Determine the Salary History (2)

- Alternative 1
  - Give the user a printout of Salary and Dept information; the user has to determine when the salary changed
- Alternative 2
  - Use SQL as much as possible
  - Find those intervals that overlap or are adjacent and that should be merged

SQL embedded into programming language

```
CREATE TABLE Temp(Salary,Start,End) AS SELECT Salary,Start,End
          FROM Employee
          WHERE Name='John'

Repeat
    UPDATE Temp T1
     SET (T1.End) = (SELECT MAX(T2.End)
                            FROM Temp AS T2
                            WHERE T1.Salary = T2.Salary
                                  AND T1.Start <T2.Start
                                  AND T1.End >= T2.Start
                                  AND T1.End < T2.End)
    WHERE EXISTS ( SELECT *
                  FROM Temp AS T2
                  WHERE T1.Salary = T2.Salary
                        AND T1.Start <T2.Start
                        AND T1.End >= T2.Start
                        AND T1.End < T2.End)
Until no tuples updated
```

Iterative loop

Overlapping intervals

If such overlapping intervals exist

Interval merging

Initial intervals

After One pass

After Two pass

SQL

- 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

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

Same functionality entirely in SQL

```
CREATE VIEW Temp(Salary,Start,End) AS
    SELECT Salary, Start, End
    FROM Employee
    WHERE Name='John'

SELECT DISTINCT F.Salary, F.Start, L.End
FROM Temp AS F, Temp AS L
WHERE F.Start < L.End AND F.Salary = L.Salary
AND NOT EXISTS (SELECT *
                FROM Temp AS T
                WHERE T.Salary = F.Salary
                AND F.Start < T.Start AND T.Start < L.End
                AND NOT EXISTS ( SELECT *
                                 FROM Temp AS T1
                                 WHERE T1.Salary = F.Salary
                                 AND T1.Start < T.Start
                                     AND T.Start <= T1.End))
AND NOT EXISTS (SELECT *
                FROM Temp AS T2
                WHERE T2.Salary = F.Salary
                AND ((T2.Start < F.Start AND F.Start <= T2.End)
                  OR (T2.Start < L.End AND L.End < T2.End)))
```

No holes between F.start and L.end

Only maximal periods result

Same query in Calculus

$$
\begin{aligned}
\{ & f.\text{Start}, l.\text{Stop} \mid \text{Temp}(f) \wedge \text{Temp}(l) \wedge f.\text{Start} < l.\text{Stop} \wedge f.\text{Salary} = l.\text{Salary} \wedge \\
& \quad (\forall t)(\text{Temp}(t) \wedge t.\text{Salary} = f.\text{Salary} \wedge f.\text{Start} < t.\text{Start} \wedge t.\text{Start} < l.\text{Stop} \rightarrow \\
& \quad\quad (\exists t_1)(\text{Temp}(t_1) \wedge t_1.\text{Salary} = f.\text{Salary} \wedge t_1.\text{Start} < t.\text{Start} \wedge \\
& \quad\quad\quad t.\text{Start} <= t_1.\text{Stop})) \wedge \\
& \quad \neg(\exists t_2)(t_2.\text{Salary} = f.\text{Salary} \wedge \\
& \quad\quad (\ (t_2.\text{Start} < f.\text{Start} \wedge f.\text{Start} <= t_2.\text{Stop}) \vee \\
& \quad\quad\ (t_2.\text{Start} < l.\text{Stop} \wedge l.\text{Stop} < t_2.\text{Stop}))\ )\ \}
\end{aligned}
$$

Solution 3: SQL and a cursor to build chain of intervals in the table

```
Maintain a linked list of intervals, for each salary;
Initialize this linked list to empty;
DECLARE emp_cursor CURSOR FOR
   SELECT Salary, Start, Stop
   FROM Employee
   WHERE Name = 'Bob'
OPEN emp_cursor;
loop:
   FETCH emp_cursor INTO :salary, :start, :stop;
   if no-data returned, then go to finish;
   find position in list to insert this information
   go to loop;
finish:
   CLOSE emp_cursor
   iterate through linked list for printing dates and salary
```

The linked list is only needed if the cursor is not ordered by START

Another solution

- Reorganize the schema in order to separate information about Salary from information about Dept

Employee1(Name,Salary,Start DATE, End DATE)
Employee2(Name,Dept,Start DATE, End DATE)

Another solution (2)

- Reorganize the schema
  Employee1(Name,Salary,Start DATE, End DATE)
  Employee2(Name,Dept,Start DATE, End DATE)
- Determining information about the salary is easy now!

```
SELECT Salary, Start, End
FROM    Employee1
WHERE Name ='John'
```

Another solution (3)

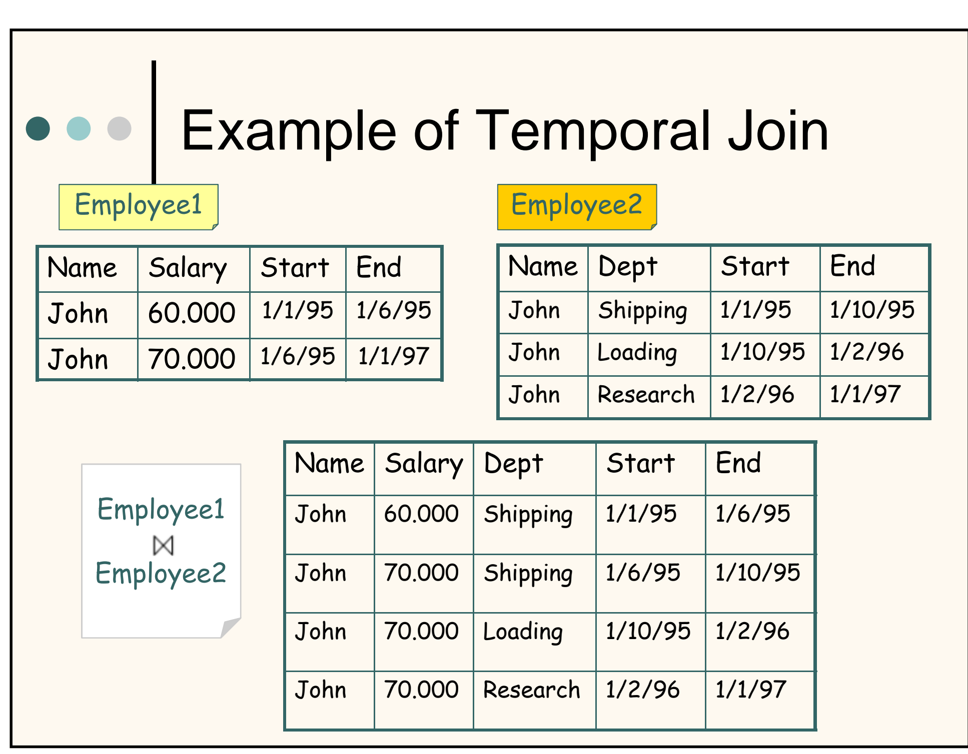
- But now, given

Employee1(Name,Salary,Start DATE, End DATE)
Employee2(Name,Dept, Start DATE, End DATE)

How do we obtain (again) a table of

SALARY, DEPARTMENT

intervals?



Example of Temporal Join

Employee1

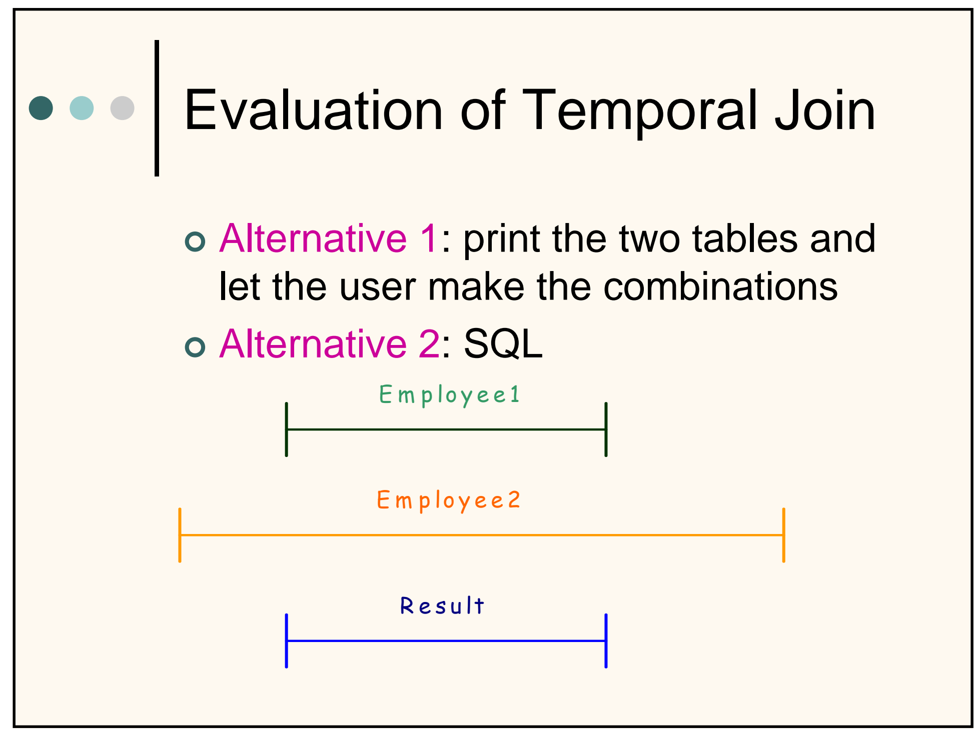
Name	Salary	Start	End
John	60.000	1/1/95	1/6/95
John	70.000	1/6/95	1/1/97

Employee2

Name	Dept	Start	End
John	Shipping	1/1/95	1/10/95
John	Loading	1/10/95	1/2/96
John	Research	1/2/96	1/1/97

Employee1 ⋈ Employee2

Name	Salary	Dept	Start	End
John	60.000	Shipping	1/1/95	1/6/95
John	70.000	Shipping	1/6/95	1/10/95
John	70.000	Loading	1/10/95	1/2/96
John	70.000	Research	1/2/96	1/1/97



Evaluation of Temporal Join

- **Alternative 2: SQL**

```
SELECT Employee1.Name, Salary, Dept,
              Employee1.Start, Employee1.End
FROM Employee1,Employee2
WHERE Employee1.Name=Employee2.Name
  AND Employee2.Start <= Employee1.Start
  AND Employee1.End < Employee2.End
```

→ This is only ONE of the possible relationships between the two intervals!

Temporal Join in SQL

```
SELECT Employee1.Name, Salary, Dept, Employee1.Start, Employee1.End
FROM  Employee1, Employee2
WHERE Employee1.Name=Employee2.Name
   AND Employee2.Start <= Employee1.Start
   AND Employee1.End <= Employee2.End
UNION ALL
SELECT Employee1.Name, Salary, Dept, Employee1.Start, Employee2.End
FROM  Employee1, Employee2
WHERE Employee1.Name=Employee2.Name
   AND Employee1.Start > Employee2.Start
   AND Employee2.End < Employee1.End
   AND Employee1.Start < Employee2.End
UNION ALL
SELECT Employee1.Name, Salary, Dept, Employee2.Start, Employee1.End
FROM  Employee1, Employee2
WHERE Employee1.Name=Employee2.Name
   AND Employee2.Start > Employee1.Start
   AND Employee1.End < Employee2.End
   AND Employee2.Start < Employee1.End
UNION ALL
SELECT Employee1.Name, Salary, Dept, Employee2.Start, Employee2.End
FROM  Employee1, Employee2
WHERE Employee1.Name=Employee2.Name
   AND Employee2.Start >= Employee1.Start
   AND Employee2.End <= Employee1.End
```


Evaluation of Temporal Join

- Alternative 3: use embedded SQL
- TSQL2: Give the salary and the history of employees

```
SELECT Employee1.Name, Salary, Dept,
FROM Employee1,Employee2
WHERE Employee1.Name=Employee2.Name
```

Desiderata for a Temporal Data Model

- Capture the semantics of time-varying information
- Retain the simplicity of the relational model: in practice, a strict superset
- Present all the information concerning an object in a coherent fashion
- Ensure ease of implementation
- Ensure high performance

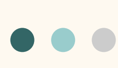
In a temporal DB, information is timestamped

- A *timestamp* is a seven-part value (*year, month, day, hour, minute, second,* and *microsecond*) that designates a date and time as defined above, except that the time includes a fractional specification of microseconds.
- The internal representation of a timestamp is a string of 10 bytes, each of which consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds.
- The length of a TIMESTAMP column, as described in the SQLDA, is 26 bytes, which is the appropriate length for the character string representation of the value.

What should be timestamped?

Temporal DBMSs manage time-referenced data:
timestamps are associated to database entities

- Individual attribute value
- Group of attribute values
- Individual tuple
- Object
- Set of tuples
- Schema item



What is the semantics of a timestamp?

Database facts have at least two relevant aspects:

- ***Valid Time*** of a fact: it times when the fact is true in the modeled reality – thus valid time captures the *time-varying states* ***of the real world***
- ***Transaction Time*** of a fact: when it was recorded in the database – thus transaction time captures the *time-varying states* ***of the database***

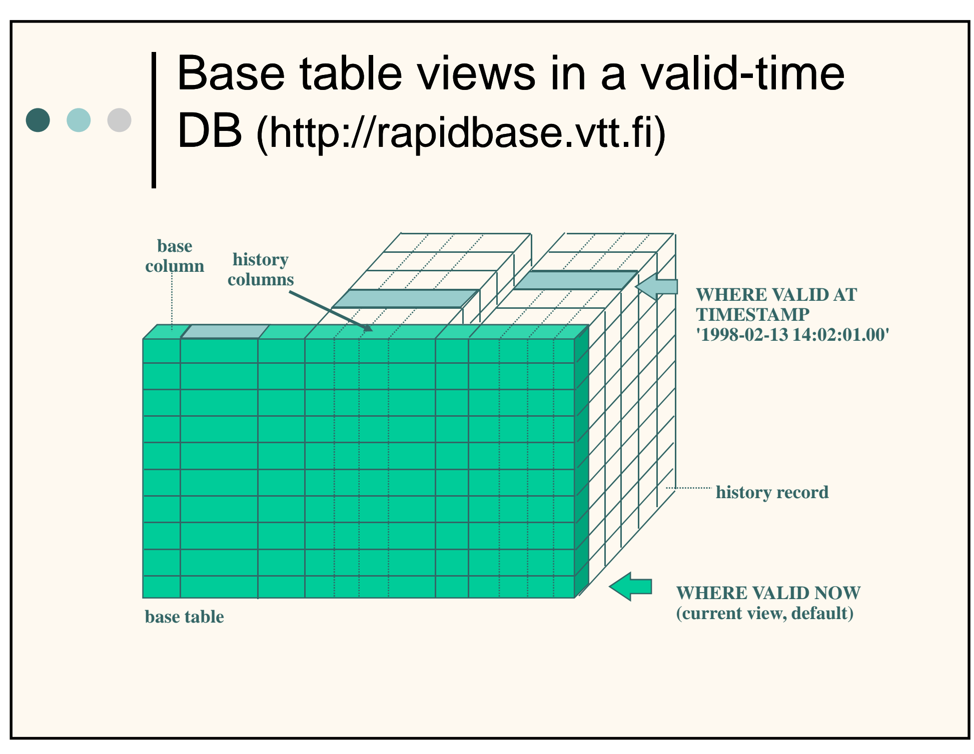
Applications that demand traceability of DB changes require transaction time

Valid time

Valid time concerns the time when a fact was true in the modelled reality

- It can be **in the past or in the future** and can be changed frequently
- Although **all facts have a valid time**, the valid time of a fact may not necessarily be recorded in the DB (e.g. unknown or irrelevant to the application)
- If a database models different worlds, database facts might have several valid times, **one for each world**

Base table views in a valid-time DB (http://rapidbase.vtt.fi)



Transaction Time

Transaction time concerns when a fact was current in the database

- It **cannot extend beyond the current time and cannot be changed**
- TT may be associated with any database entity, not only facts
- From the TT viewpoint, an entity has a duration: from insertion to deletion, with multiple insertions and deletions being possible for the same entity !
- Deletion is pure logical (not physically removed) but the entity ceases to be part of the database's current state

Note…

- Transaction time may be associated not only to real world facts, but also to other DB concepts, like attribute values, which are updated at a given time

E.g.

Name	Salary	Bdate	City
Bob	60000	1943-01-01	Milan
John	70000	1955-06-01	London

Name	Salary	Bdate	City
Bob	60000	1943-01-01	Milan
John	80000	1955-06-01	London

Modified Dec 15, 2005

Queries and Updates

- A transaction time table is append only: it keeps the history of the updates made on the database.
- Transaction time tables support rollback queries, such as:
  - On October 1, what rank was our database showing for Tom?
- A valid time table can be updated: e.g., Tom's past record is changed once his rank is changed retroactively.
- Valid time tables support historical queries, such as:
  - What was Tom's rank on October 1 (according to our current database)?

Valid and Transaction time are orthogonal temporal dimensions

- They could be independently recorded or not and are associated with specific properties
- TT, unlike VT, is well-behaved and may be supplied automatically by the DBMS
- Both TT and VT values are drawn from a time domain, which may or may not stretch infinitely into past and future
- Time domain may be discrete or continuous
- In databases, a finite and discrete time domain is typically assumed

Valid Time and Transaction Time

Thus we can have four different kinds of tables:

1. ***Snapshot***
2. ***Valid-time***
3. ***Transaction-time***
4. ***Bitemporal***

Example: Tom's Employment History

- On January 1, 1984, Tom joined the faculty as an Instructor.
- On December 1, 1984, Tom completed his doctorate, and so was promoted to Assistant Professor effective *retroactively* on July 1, 1984.
- On March 1, 1989, Tom was promoted to Associate Professor, effective July 1, 1989 (*proactive update*).

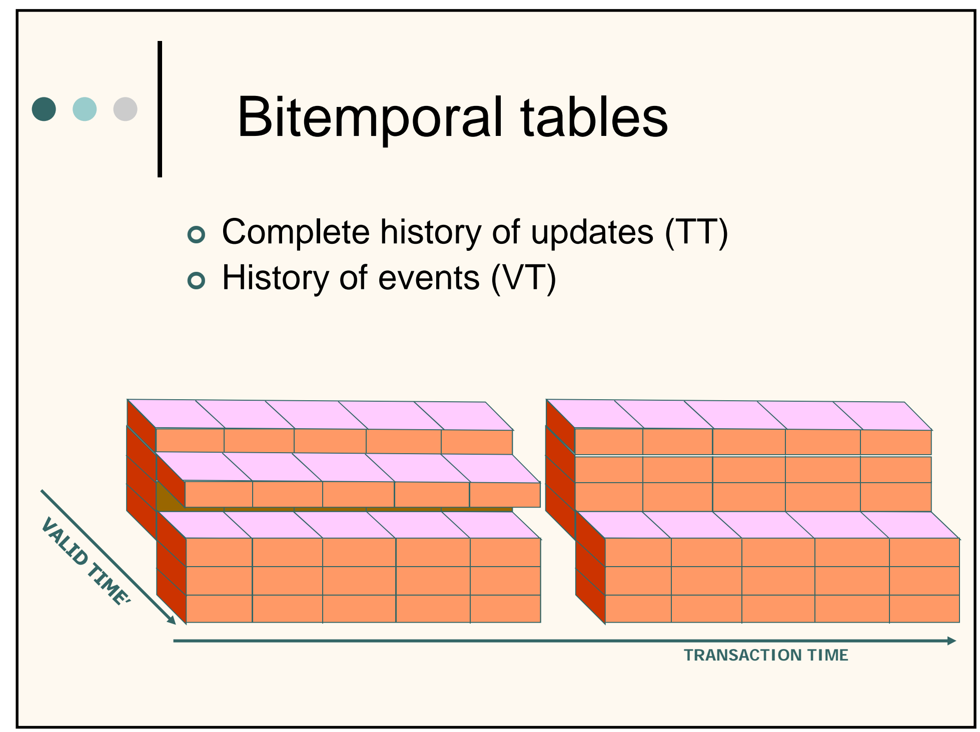
Bitemporal Tables

- Bitemporal Tables are append-only and support queries of both kinds (rollback&historical) such as:
  - On October 1, 1984, what did we think Tom's rank was at that date?
- TSQL3:

```
SELECT Rank
FROM Faculty AS F
WHERE Name = 'Tom'
        AND
VALID(F) OVERLAPS DATE  '1984-10-01'
        AND
TRANSACTION(F) OVERLAPS DATE '1984-10-01'
```

Bitemporal tables

- Complete history of updates (TT)
- History of events (VT)



MAIN PHILOSOPHICAL ISSUES

- LINEAR vs. CIRCULAR → structure
- FINITE vs. INFINITE → boundedness
- OPEN vs. CLOSED intervals
- DISCRETE vs. CONTINUOUS → density
- ABSOLUTE (past, present, future) vs. RELATIVE (before, concurrent-with, after)
- OBJECTIVE vs. SUBJECTIVE

Main Time Properties in Temporal Database research (inherited from philosophy)

- Structure
- Boundedness
- Density
- Time Data Types
- Time and Facts
- In more sophisticated ways, time related to context may be:
  - ABSOLUTE vs. RELATIVE
  - OBJECTIVE vs. SUBJECTIVE

Time structure

- Linear Time
- Branching Time
- Directed Acyclic Graph
- Periodic/Cyclic Time e.g., days of the week.

Time structure

- Linear: total order on instants
- Hypothetical (possible futures): tree rooted on *now*

Time structure

- **Linear: total order on instants**
- **Hypothetical (possible futures): tree rooted on *now***
- **Direct Acyclic graph**
- **Periodic/cyclic time: week, months,…., for recurrent processes**

Time is normally assumed to be totally ordered

Time Boundedness

- Unbounded
- Time origin exists (bounded from the left)
- Bounded time (bounds on both ends)

Time Boundedness

- Assume a linear time structure
- Boundedness
  - Unbounded
  - Time origin exists (bounded from left)
  - Bounded time (bounds on both ends)
- Nature of bound
  - Unspecified
  - Specified

Time Density

- ***Discrete***:
  - Time line is isomorphic to the integers
  - Time line 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.

A *bounded discrete representation* of time is the simplest option, used in SQL-2 and most temporal DBs.

Time Density

- ***Dense:*** *(difficult to implement)*
  - Time line is isomorphic to the rational numbers.
  - Between any two chronons is an infinite number of other instants.
- ***Continuous:*** *(very difficult to implement)*
  - Time line is isomorphic to the real numbers.
  - Between each pair of instants is an infinite number of other instants.

NOW: a peculiar concept

- NOW:
  - Ever increasing
  - Whatever activity happens now
  - It separates the past from the future
- HERE: the main difference is that you can't "reuse" time
- The uniqueness of now is one of the reasons why techniques from other research areas are not readily (or not completely) applicable to temporal data

Various Temporal Types used in temporal DBs

- A **time instant** is a time point on the time line (a chronon).
- An **event** is an instantaneous fact, i.e, something occurring at an instant. The event occurrence time is the (valid time) instant at which the event occurs in the real world.
- An **instant set** is a set of instants.
- A **time period or interval** is the set of time instants between two instants (start time and end time).
- The name "interval" conflicts with SQL data type INTERVAL

The SQL INTERVAL data type holds the internal (binary) format of an interval value. It encodes a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals. A year-month interval can represent a span of years and months, and a day-time interval can represent a span of days, hours, minutes, seconds, and fractions of a second.

Temporal Types (2)

- Time interval: an oriented 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

Intervals

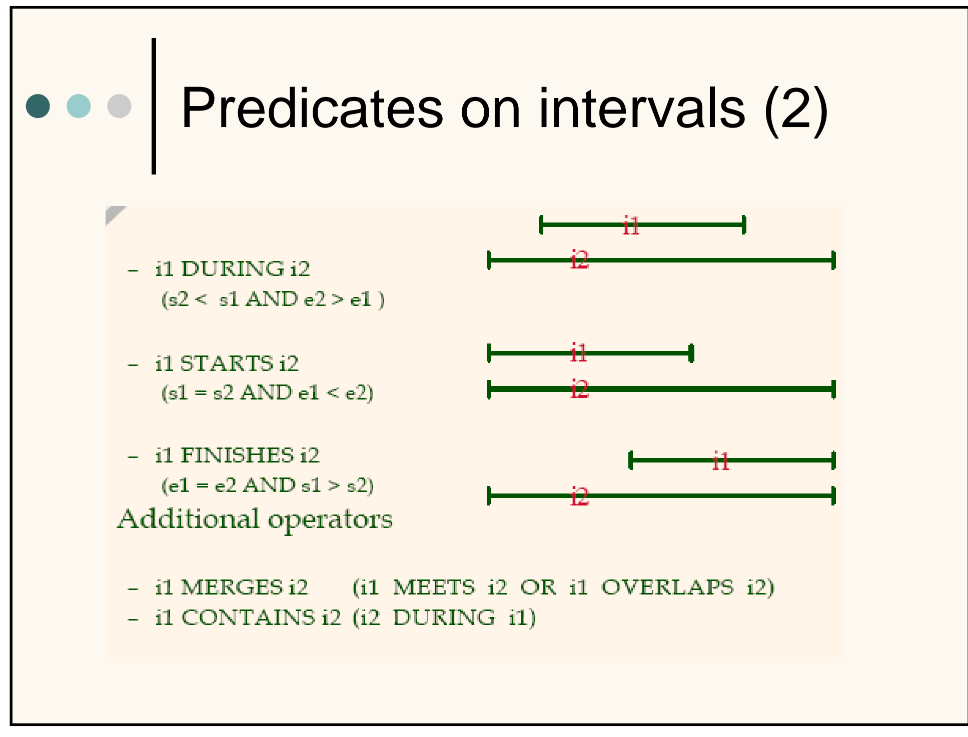
- An interval [s,e] is a set of times from time s to time e.
- Does interval [s,e] represent an infinite set?
- Consider it as a finite sequence of chronons
- An interval is treated as a single type, not as a pair of separate values.
- Intervals can be open/closed w.r.t. start point/end point. eg.

```
[d04,d10],[d04,d11),(d03,d10],(d03,d11)
```

all represent the sequence of days from day4 to day10 inclusive.

Predicates on intervals

- i1 BEFORE i2 (e1<s2)
- i1 MEETS i2 (s2 = e1)
- i1 EQUALS i2 (s1 = s2 AND e1 = e2)
- i1 OVERLAPS i2 (s2 < s1 < e2 AND e2 < e1)



Relational operators

- DURATION(i) - returns the number of time points in i
  - eg. DURATION ([d03,d07]) returns 5
- i1 UNION i2
  - returns [MIN(s1,s2),MAX(e1,e2)]

    if (i1 MERGES i2)

    otherwise undefined
- i1 INTERSECT i2
  - returns [MAX(s1,s2),MIN(e1,e2)]

    if (i1 OVERLAPS i2)

    otherwise undefined

Aggregate operators

- UNFOLD(X):

  Where X is a set. The output is also a set.

  Used to generate time quantum intervals.

  - The *unfolded form* of X is the set of all intervals of the form [p,p] where p is a time point in some interval in X.

  e.g.

  X1 = { [d01,d01],[d03,d05],[d04,d06] }

  X2 = { [d01,dp1],[d03,d04],[d05,d05],[d05,d06] }

  X3 = { [d01,d01],[d03,d03],[d04,d04],[d05,d05],[d06,d06] }

  X3 = UNFOLD (X1)

  = UNFOLD (X2)

Aggregate operators

- The *coalesced form* of X is the set Y of intervals of the same type such that

(a) X & Y have the same unfolded form.

(b) no two distinct members i1 and i2 of Y are such that (i1 MERGES i2) is true.

e.g.

X1 = { [D01,D01], [D03,D05], [D04,D06] }

X2 = { [D01,D01], [D03,D04], [D04,D06] }

X3 = { [D01,D01], [D03,D06]}

X3 = COALESCE (X1) = COALESCE(X2)

Coalesce

- R COALESCE A/P
  (Temporal Projection)
  - Get set of intervals which satisfy predicate P.
    Where R is a relational expression and A is an attribute (of some interval type) of the relation.
    The output is a relation with the desired attributes.

Example of coalesce

Get S#-DURING pairs for suppliers who have supplied parts P1 or P2 at some time.

SP

S#	P#	During
S1	P1	[d04,d10]
S1	P7	[d05,d10]
S1	**P2**	[d09,d10]
S1	P5	[d06,d10]
S2	P1	[d02,d04]
S2	P9	[d03,d03]
S2	P1	[d08,d10]
S2	P5	[d09,d10]
S3	P1	[d08,d10]
S4	P2	[d06,d09]
S4	P5	[d04,d08]
S4	P7	[d05,d10]

SP {S#,DURING} COALESCE DURING/

P#=P1 or P#=P2

S#	During
S1	[d04,d10]
S2	[d02,d04]
S2	[d08,d10]
S3	[d08,d10]
S4	**[d06,d09]**

A relational example

Employee (Name, Salary, Title, DateofBirth,Start DATE, Stop DATE)

Name	Salary	Title	DateofBirth	Start	Stop
Bob	60000	AssistantProvost	1945-04-19	1993-01-01	1993-06-01
Bob	70000	AssistantProvost	1945-04-19	1993-06-01	1993-10-01
Bob	70000	Provost	1945-04-19	1993-10-01	1994-02-01
Bob	70000	Professor	1945-04-19	1994-02-01	1995-01-01

Extracting the Salary History in TSQL2

```
SELECT Salary,startDATE,stopDATE
FROM Employee
WHERE Name = 'Bob'
```

There is no explicit mention of time in the query. By default the system returns the **coalesced time history**

Name	Salary	Start	End
Bob	60.000	1/1/95	1/6/95
Bob	70.000	1/6/95	1/1/97

Unfold

R UNFOLD A

- Used to generate set of all datum (time samples) from a given relation R.

  Where R is a relational expression and A is an attribute (of some interval type) of the relation.

  The output is another relation with the same headers.

Example of unfold

SP

S#	During
S1	[d04,d10]
S1	[d05,d10]
S1	[d09,d10]
S1	[d06,d10]
S2	[d02,d04]
S2	[d03,d03]
S2	[d08,d10]
S2	[d09,d10]
S3	[d08,d10]
S4	[d06,d09]
S4	[d04,d08]
S4	[d05,d10]

SP UNFOLD DURING

S#	During
S1	[d04,d04]
S1	[d05,d05]
S1	[d06,d06]
S1	[d07,d07]
S1	[d08,d08]
S1	[d09,d09]
S1	[d10,d10]
S2	[d02,d02]
S2	[d03,d03]
S2	[d04,d04]
.	.
.	.
.	.

Temporal difference

Used to answer negated queries.

R1 I_MINUS R2 ON **DURING**

- where R1 and R2 are relations of the same type and A is an attribute of some interval type common to these relations.
- The output is a relation with the same heading as R1 or R2
- Equivalent to:

((R1 UNFOLD DURING) MINUS
(R2 UNFOLD DURING)) COALESCE DURING

Temporal Joins: Example of a difficult task

Employee1:

Name	Salary	Start	Stop
Bob	60000	1993-01-01	1993-06-01
Bob	70000	1993-06-01	1995-01-01

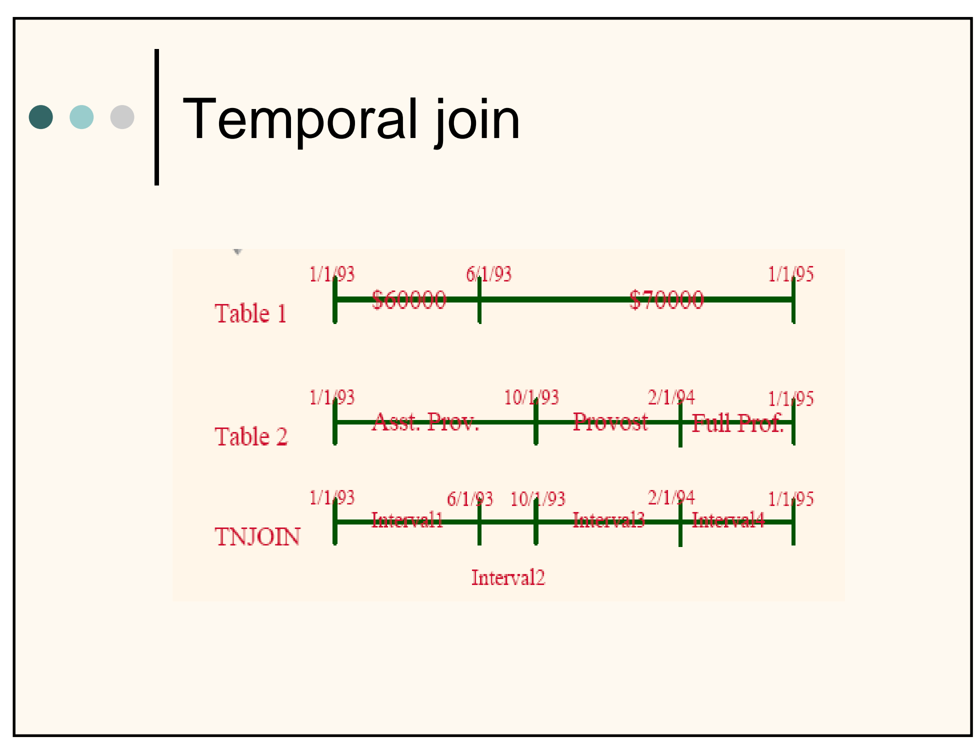
Employee2:

Name	Title	Start	Stop
Bob	AssistantProvost	1993-01-01	1993-10-01
Bob	Provost	1993-10-01	1994-02-01
Bob	FullProfessor	1994-02-01	1995-01-01

Their Temporal Join:

Name	Salary	Title	Start	Stop
Bob	60000	AssistantProvost	1993-01-01	1993-06-01
Bob	70000	AssistantProvost	1993-06-01	1993-10-01
Bob	70000	Provost	1993-10-01	1994-02-01
Bob	70000	FullProfessor	1994-02-01	1995-01-01

Temporal join



Temporal Join in SQL

```
SELECT E1.Name, Salary, Title,
       E1.Start, E1.Stop
FROM Employee1 AS E1,
       Employee2 AS E2
WHERE E1.Name=E2.Name AND
    E2.Start <= E1.Start AND
    E1.Stop <= E2.Stop
UNION ALL
SELECT E1.Name, Salary, Title,
    E1.Start, E2.Stop
FROM Employee1 AS E1,
      Employee2 AS E2
WHERE  E1.Name = E2.Name AND
    E1.Start > E2.Start AND
    E2.Stop< E1.Stop AND
    E1.Start < E2.Stop
UNION ALL
SELECT E1.Name, Salary, Title
            E2.Start, E1.Stop
FROM Employee1 AS E1,
      Employee2 AS E2
WHERE
    E1.Name = E2.Name AND
    E2.Start > E1.Start AND
    E1.Stop <= E2.Stop AND E2.Start < E1.Stop
UNION ALL
SELECT E1.Name, Salary, Title
            E2.Start, E2.Stop
FROM Employee1 AS E1,
      Employee2 AS E2
WHERE
    E1.Name = E2 Name AND E2.Start => E1.Start
    AND E2.Stop <= E1.Stop AND   NOT
    (E1.Start = E2.Start AND E1.Stop = E2.Stop)
```

Temporal Joins in TSQL2

```
SELECT E1.Name, Salary, Title
FROM Employee1 AS E1, Employee2 AS E2
WHERE E1.Name = E2.Name
```

Note: in TSQL2, the ***basic temporal element*** is a finite union of intervals.

Constraints

Consider the S_DURING relation.

- What should be the primary key?
- Is { S#, during } primary key sufficient?
- Consider:

S2	Jones	10	Paris	[d02,d08]
S2	Jones	10	Paris	[d07,d10]

Primary key fails to prevent redundancy (overlap). (information about day7 and day 8 is recorded twice)

Constraints

An example of a contradiction.

S2	Jones	10	Paris	[d02,d08]
S2	Jones	20	Paris	[d07,d10]

- The status of S2 was both 10 and 20 on days 7 and 8

We need to add another constraint

- If two distinct S_DURING tuples with the same S# values and other possible attributes have DURING values i1 and i2, and (i1 OVERLAPS i2), then those two tuples must be identical except possibly for their DURING values.

This can be enforced by keeping the relation **unfolded** at all times on attribute DURING.

- i.e. {S#,DURING} as key for S_DURING UNFOLD DURING
- The concept of '*temporal key*'

Reviewing the Situation

- The importance of temporal applications has motivated much research on temporal DBs: but no satisfactory solution has been found yet:
  - SQL3 does not support temporal queries, yet the standard contains a "Part 7: SQL/Temporal"
  - Temporal DBMSs have not reached a satisfactory performance level and remain an open research problem.
  - Several alternatives, in terms of data model and SQL extensions to be used

Temporal Data Models

- To extend a DBMS to become temporal, mechanisms must be provided for capturing valid and transaction times of the facts recorded by relations (temporal relations)
- More than 24 extended relational models have been proposed to add time to the relational model
- Most of them support only valid time

Temporal Query Languages

- In the literature, there are several query and definition languages managing temporal dimensions
  - TSQL, HSQL,TempSQL: only VT
  - TOSQL: only TT
  - TSQL2, TQuel, SQL3: both TT and VT

Business reality changes over time

- Software changes are actually to be accepted as a part of the software operational life:
  - call for modifications of the applications or of the database schema
  - require compatibility with the interfaces users were adopting at a previous time.
- The knowledge about reality is modified:
  - Appropriate documentation of such changes should provide change tracking and explanation.

Business reality changes over time

Modifications may be due to:
- an improved perception/knowledge of the reality of interest → need for application evolution. Examples:
  - simple lexical/terminological changes due, for instance, to the acquisition of new clients/customers or to the merge with another company or a richer carnet of functionalities offered by the same application
- a modification of the reality of interest, Examples:
  - the citizenship act of United Kingdom;
  - different enrolment policies at the University;
  - different national borders and laws;
  - different job regulations in a company [e.g. new medical data required for some specific patient categories];
  - new medical or pharmaceutical knowledge about diseases and therapies/protocols.

Schema evolution and versioning

- changes are seldom appropriately documented, and subsequent modifications must often rely on the designers' and the DBA's memory.
- need for seamless compatibility among different schema versions, in order for users to be redirected over the appropriate version depending on the application phase they are referring to.
- Same considerations apply to a temporal database:
  - all the previous versions of the data, schema and applications must be preserved
  - support is needed to answer historical as well as snapshot queries independently of instance and schema changes.

Schema evolution and versioning

- Deal with the need to retain current data and software system functionality in the face of changing database structure.
- Offer a solution to the problem by enabling intelligent handling of any temporal mismatch between data and data structure.

Schema evolution and versioning

- Schema evolution: permits modifications of the schema without the loss of extensional data
- Schema versioning: allows the querying of all data through appropriate version-based interfaces

→ Schema evolution is a particular case of schema versioning, where only the last version of the schema is retained

Schema Modification Operators (Curino, Zaniolo et al.)

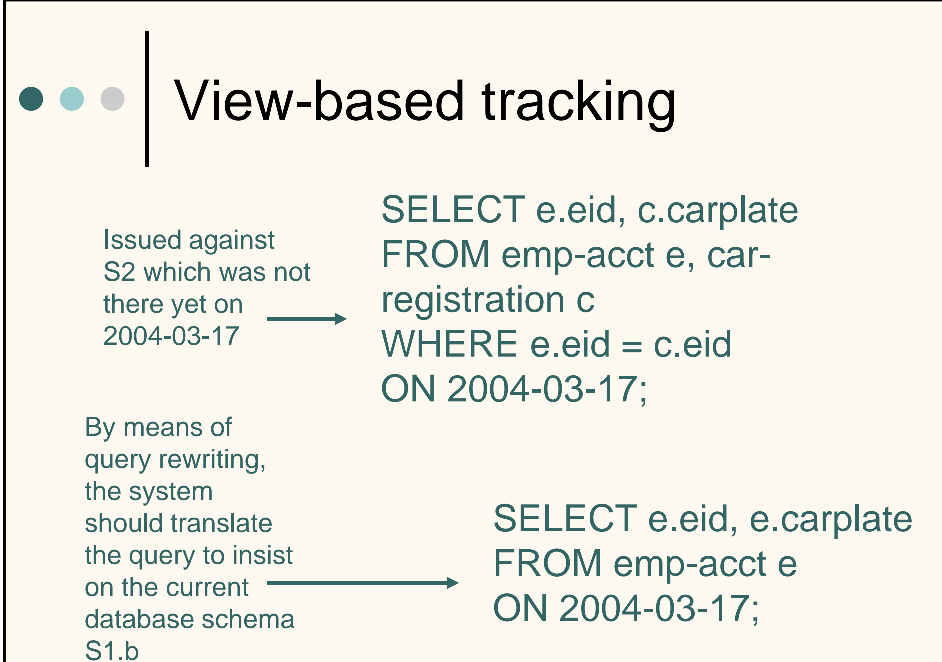
SMO	Description
CREATE TABLE	introduces a new, empty table to the database, as in SQL:2003 standard [2003, ,Eisenberg et al., 2004]
DROP TABLE	removes an existing table from the schema and deletes the data in the table, as in SQL:2003 standard
RENAME TABLE	renames a table, without affecting the data.as in SQL:2003 standard
PARTITION TABLE	takes as input a source table and distributed, according to the condition specified by the user, the tuple among 2 newly generated tables, the source table is then dropped.
MERGE TABLE	takes two source tables with the same schema and creates a new table with the same schema and a union of the two tables. It has to be checked that the two source tables do not present key conflicts.
ADD COLUMN	introduces a new column into the specified table, where the new column is filled with the values generated by a user specified function (NULL by default).
DROP COLUMN	removes an existing column from a table, deleting all data in the column.
RENAME COLUMN	changes the name of a column, without affecting the data.
COPY COLUMN	makes a copy of a column into another table, filling the value according to a join condition among source and target tables.
MOVE COLUMN	same as COPY COLUMN but the original column is dropped.

Track changes via imperative primitives: example

	(Intermediate) Schema Versions	SMO Sequence
S_1	emp-acct (eid, name, ethnicity, dept, job, carplate) emp-med (eid, weight, bp) salary (job, amount)	*M1*. PARTITION TABLE emp-med INTO high-emp-med WITH high-emp-med.eid = emp-acct.eid AND emp-acct.job ='production'; low-emp-med WITH low-emp-med.eid = emp-acct.eid AND emp-acct.job != 'production' *M2*. ADD COLUMN lung, skin INTO high-emp-med
S_{1a}	emp-acct (eid, name, ethnicity, dept, job, carplate) high-emp-med (eid, weight, bp, lung, skin) low-emp-med (eid, weight, bp) salary (job, amount)	*M3*. DROP COLUMN ethnicity FROM emp-acct *M4*. COPY COLUMN amount FROM salary INTO emp-acct WHERE salary.job = emp-acct.job *M5*. RENAME TABLE salary INTO job-salary-ref *M6*. RENAME COLUMN amount IN emp-acct TO salary
S_{1b}	emp-acct (eid, name, dept, job, carplate, salary) high-emp-med (eid, weight, bp, lung, skin) low-emp-med (eid, weight, bp) job-salary-ref (job, amount)	*M7*. CREATE TABLE car-registration *M8*. COPY COLUMN eid FROM emp-acct INTO car-registration *M9*. MOVE COLUMN carplate FROM emp-acct INTO car-registration WHERE emp-acct.eid = car-registration.eid
S_2	emp-acct (eid, name, dept, job, salary) high-emp-med (eid, weight, bp, lung, skin) low-emp-med (eid, weight, bp) job-salary-ref (job, amount) car-registration (eid, carplate)	

View-based tracking approach

- Changes between two different versions are represented by means of appropriate queries (views)
- Applications and users interact with such views, which give place to different mappings w.r.t. the different versions



Acknowledgements and Temporal DBs pointers

- http://www.scism.sbu.ac.uk/cios/paul/Research/tdb_links.html
- http://www.seas.smu.edu/~mario/tdb-general.html
- http://www.cs.aau.dk/~csj/Thesis/pdf/chapter1.pdf
- Rick Snodgrass, Ilsoo Ahn: "Temporal Databases", IEEE Computer, Sept. 1986
- This set of slides is derived from various resources found on the web (Zaniolo, Course at Purdue Univ., etc.)