

Temporal Extensions of SQL SL04

- ▶ Comparative analysis
 - ▶ **SQL + abstract data types**: extend SQL with abstract data type
 - ▶ **IXSQL**: two operators to convert periods to points and vice versa
 - ▶ **SQL/TP**: only points exist; periods are internal and for display
 - ▶ **TSQL2**: syntactic defaults to easily formulate temporal statements
 - ▶ **ATSQL**: semantic defaults to easily formulate temporal statements
- ▶ Implementation of the sequenced semantics
 - ▶ RA + normalize, split, extend, absorb, scale

Table of Contents

- ▶ Comparative analysis
 - ▶ **SQL + abstract data types**
 - ▶ **IXSQL**
 - ▶ **SQL/TP**
 - ▶ **TSQL2**
 - ▶ **ATSQL**
- ▶ Implementation of the sequenced semantics
 - ▶ Requirements
 - ▶ Algebraic primitives
 - ▶ Reduction rules
 - ▶ SQL implementation
 - ▶ Scaling

Temporal Query Languages

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

Language Design Criteria/1

- ▶ **Expressive power**
 - ▶ Suitable for intended applications
 - ▶ Economy of encoding is relevant
- ▶ **Clarity**
 - ▶ Syntax should reflect the semantics to facilitate the formulation and understanding of queries
 - ▶ Consistent naming style
- ▶ **Consistency**
 - ▶ Upward compatibility with standards, e.g., SQL standard
 - ▶ Consistent syntax (e.g., not mix up postfix and prefix notation)
 - ▶ Systematic extension (not a new construct for each query)

Language Design Criteria/2

► Minimality

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

► Orthogonality

- Possibility to freely combine query language constructs that are semantically independent
- No exceptions
- Zero-One-Infinity principle (the only reasonable numbers in a programming language design are zero, one, and infinity)

► Closed-form evaluation

- The result of a query is a proper object of the data model.

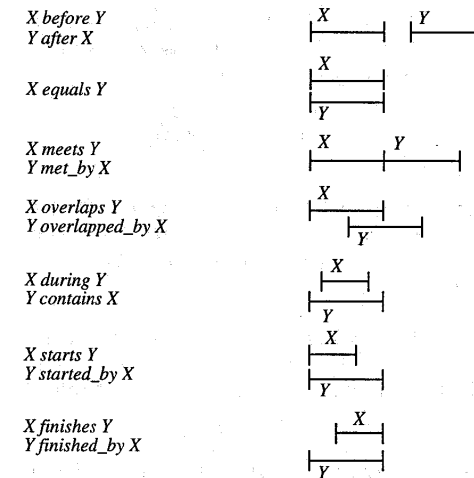
NDBS14, SL04

5/58

M. Böhlen, ifi@uzh

Comparison of Timestamps

- **Comparison of timestamps** is part of every temporal query language.
- Many query languages adopt (a variant of) **Allen's 13 period relations**:



NDBS14, SL04

6/58

M. Böhlen, ifi@uzh

Video Rental Example/1

- **Example:** Video store where customers (*CustID*) rent video tapes (*TapeNum*). The rentals are stored in a valid-time relation *Checkout*.

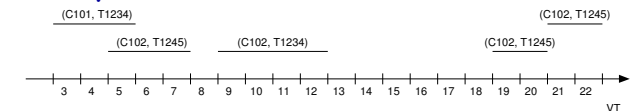


NDBS14, SL04

7/58

M. Böhlen, ifi@uzh

Video Rental Example/2



- Different queries on the *Checkout* table with their intended result:

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

CustID	TapeNum	T
C102	T1245	[5,7]
C102	T1234	[9,12]

Q2: All 2-day checkouts.

TapeNum	T
T1245	[19,20]
T1245	[21,22]

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

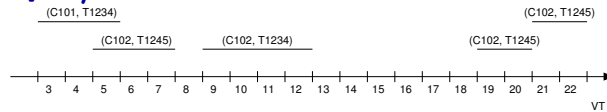
TapeNum	T
T1245	[5,5]

NDBS14, SL04

8/58

M. Böhlen, ifi@uzh

Video Rental Example/3



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

Cnt	T
1	[3,4]
2	[5,5]
1	[6,7]
0	[8,8]
1	[9,12]
0	[13,18]
⋮	⋮

Q5: How many checkouts were made in total?

Cnt
5

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

CustID	TapeNum
C101	T1234
C102	T1245

SQL + Abstract Data Types/1

- ▶ Extend existing language (e.g., SQL) with **time data types** and associated predicates and functions
 - ▶ e.g., predicates for timestamp comparison
- ▶ Earliest and (from a language design perspective) simplest approach.
- ▶ Has **limited impact on existing language** and is well understood technically.
- ▶ We assume **period timestamped** relations (or separate start and end points, which does not change much)

SQL + Abstract Data Types/2

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

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

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

Q4 too complicated

Q5 `SELECT COUNT(*) FROM CheckOut`

Q6 `SELECT CustID, TapeNum`
`FROM CheckOut`
`WHERE T OVERLAPS [NOW]`

CheckOut

CustID	TapeNum	T
C101	T1234	[3,5]
C102	T1245	[5,7]
C102	T1234	[9,12]
C102	T1245	[19,20]
C102	T1245	[21,22]

Assessment of SQL + Abstract Data Types

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

IXSQL/1

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

IXSQL/2

- ▶ The formulation of the first 3 queries are essentially SQL
- ▶ Unfold/fold mechanism is not required

CheckOut

CustID	TapeNum	T
C101	T1234	[3,5]
C102	T1245	[5,7]
C102	T1234	[9,12]
C102	T1245	[19,20]
C102	T1245	[21,22]

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

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

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

IXSQL/3

- ▶ Unfold can be used to formulate snapshot reducible queries.
 - ▶ Idea of fold and unfold:
convert to time points; answer query; convert to intervals
- Q4** `SELECT COUNT(*), T`
`FROM ( SELECT *`
`FROM CheckOut`
`REFORMAT AS UNFOLD T )`
`GROUP BY T`
`REFORMAT AS FOLD T`
- ▶ Without unfold we get regular SQL without any special processing of the temporal information.

CheckOut

CustID	TapeNum	T
C101	T1234	[3,5]
C102	T1245	[5,7]
C102	T1234	[9,12]
C102	T1245	[19,20]
C102	T1245	[21,22]

Q5 `SELECT COUNT(*)`
`FROM CheckOut`

Q6 `SELECT CustID, TapeNum`
`FROM CheckOut`
`WHERE T OVERLAPS [NOW]`

Assessment of IXSQL

- ▶ Only two functions, fold and unfold, are added to SQL. This is attractive.
- ▶ Unfold can be used when needed to formulate snapshot reducible queries (it is optional and not an invasive change).
- ▶ **Efficient evaluation** of queries formulated using fold/unfold has yet to be resolved.
- ▶ Neither a purely **point-based** nor **period based** view:
 - ▶ Sensitive to specific period representation of data (e.g., queries that do not use fold/unfold)
 - ▶ Different results for different but snapshot-equivalent relations
 - ▶ Fold/unfold only preserve information of a point-based view
 - ▶ Normalization step using unfold/fold loses period information
 - ▶ Fold is not the inverse of unfold (information about the original periods is lost)
- ▶ The combination of snapshot reducibility and periods is not supported (cf. temporal difference example for SQL/TP)

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

Q1 must compare neighboring DB states

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

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

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

CheckOut

SeqNo	CustID	TapeNum	T
1	C101	T1234	3
1	C101	T1234	4
1	C101	T1234	5
2	C102	T1245	5
2	C102	T1245	6
2	C102	T1245	7
3	C102	T1234	9
3	C102	T1234	10
3	C102	T1234	11
3	C102	T1234	12
4	C102	T1245	19
4	C102	T1245	20
5	C102	T1245	21
5	C102	T1245	22

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

Q3

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

Q4

```
SELECT COUNT(*), T
FROM CheckOut
GROUP BY T
```

CheckOut

SeqNo	CustID	TapeNum	T
1	C101	T1234	3
1	C101	T1234	4
1	C101	T1234	5
2	C102	T1245	5
2	C102	T1245	6
2	C102	T1245	7
3	C102	T1234	9
3	C102	T1234	10
3	C102	T1234	11
3	C102	T1234	12
4	C102	T1245	19
4	C102	T1245	20
5	C102	T1245	21
5	C102	T1245	22

- ▶ An additional attribute is needed for time-invariant aggregation.

- ▶ The following fails for Q5:
 - ▶

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

Q5

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

Q6

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

CheckOut

SeqNo	CustID	TapeNum	T
1	C101	T1234	3
1	C101	T1234	4
1	C101	T1234	5
2	C102	T1245	5
2	C102	T1245	6
2	C102	T1245	7
3	C102	T1234	9
3	C102	T1234	10
3	C102	T1234	11
3	C102	T1234	12
4	C102	T1245	19
4	C102	T1245	20
5	C102	T1245	21
5	C102	T1245	22

Assessment of SQL/TP

- ▶ **Additional attribute needed** to preserve information about periods of the argument relations.
- ▶ Additional attribute is **not a systematic approach** to obtain point-based semantics **and** associate information with periods.
- ▶ Example: *When was tape T1245, but not tape T1234, checked out?*
 - ▶ Expected answer: {[6, 7], [19, 20], [21, 22]}
 - ▶ Plausible solution:

```
SELECT T FROM CheckOut WHERE TapeNum = T1245
EXCEPT
SELECT T FROM Checkout WHERE TapeNum = T1234
```
 - ▶ Does not guarantee that (fragments of) individual checkout periods are returned because the sequence numbers are not included
 - ▶ Including the sequence number “disables” the difference

Assessment of SQL/TP

- ▶ **Period representation is still needed** for user interaction and internal representation:
 - ▶ What the user sees is periods (points are coalesced for presentations to the user).
 - ▶ For formulating queries the user needs to (conceptually) map periods to points.
- ▶ Some timestamps, e.g., (C101, T1234, [22, ∞]) lead to problems (infinite relations).

TSQL2/1

- ▶ Comprehensive extension of SQL2 to support **time periods** and **time points**.
- ▶ Adopts **syntactic defaults** that make the formulation of common temporal queries more convenient.
- ▶ A syntactic default is an abbreviation for a more complex expression. Instead of the abbreviation it is always possible to use the complex expression.
- ▶ A **default valid clause** is implicitly placed after the select clause:
 - ▶ Computes the intersection of the valid times of the argument relation(s) in the from clause
 - ▶ Intersection is returned as timestamp in the result relation

TSQL2/2

Q1

```
SELECT *
FROM CheckOut
WHERE VALID(T) OVERLAPS period '7-9'
```

Q2

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

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

Q3 Implicit default valid clause in Q3

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

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

CheckOut			
SeqNo	CustID	TapeNum	T
1	C101	T1234	[3,5]
2	C102	T1245	[5,7]
3	C102	T1234	[9,12]
4	C102	T1245	[19,20]
5	C102	T1245	[21,22]

TSQL2/3

Q4 not well defined (what should default valid clause be??)

Q5 `SELECT SNAPSHOT COUNT(*)
FROM CheckOut`

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

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

CheckOut

SeqNo	CustID	TapeNum	T
1	C101	T1234	[3,5]
2	C102	T1245	[5,7]
3	C102	T1234	[9,12]
4	C102	T1245	[19,20]
5	C102	T1245	[21,22]

Assessment of TSQL2

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

ATSQL/1

- ▶ ATSQL introduces **temporal statement modifiers** to add temporal support to SQL
- ▶ Statement modifiers are **semantic defaults** that indicate snapshot reducible semantics without specifying how to compute it.
- ▶ Provides a **systematic way** of constructing temporal queries from nontemporal queries:
 1. Formulate the corresponding nontemporal query
 2. Apply a statement modifier
- ▶ Example: Temporal join
 - ▶ Formulate the nontemporal join
 - ▶ Modifier ensures that the argument timestamps overlap and that the result timestamp is the intersection of the argument periods
- ▶ ATSQL assumes period-timestamped tuples:
 - ▶ Value-equivalent tuples with adjacent or overlapping periods are permitted
 - ▶ Periods have a meaning beyond a set of points

ATSQL/2

- ▶ The `seq vt` (“sequenced valid time”) modifier ensures that timestamps are returned as if the enclosed statement was evaluated on each state of the database. In addition interval boundaries (change points) are preserved.
- ▶ The rest of the query is standard SQL

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

Q2 `SEQ VT
SELECT TapeNum
FROM CheckOut
WHERE DURATION(T) = 2`

CheckOut

CustID	TapeNum	T
C101	T1234	[3,5]
C102	T1245	[5,7]
C102	T1234	[9,12]
C102	T1245	[19,20]
C102	T1245	[21,22]

ATSQL/3

- ▶ Snapshot reducibility is easy to specify and the specification is independent of the query that shall be evaluated with snapshot reducible semantics.

CheckOut

CustID	TapeNum	T
C101	T1234	[3,5]
C102	T1245	[5,7]
C102	T1234	[9,12]
C102	T1245	[19,20]
C102	T1245	[21,22]

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

Q4 `SEQ VT`
`SELECT COUNT(*) FROM CheckOut`

ATSQL/4

- ▶ The `nseq vt` (“nonsequenced valid time”) modifier indicates that what follows should be treated as regular SQL
- ▶ The rest of the query is standard SQL

CheckOut

CustID	TapeNum	T
C101	T1234	[3,5]
C102	T1245	[5,7]
C102	T1234	[9,12]
C102	T1245	[19,20]
C102	T1245	[21,22]

Q5 `NSEQ VT`
`SELECT COUNT(*)`
`FROM CheckOut`

- ▶ A query without a modifier considers only the current state of the argument relations (i.e., NOW)
- ▶ Ensures that legacy queries on nontemporal relations are unaffected if the nontemporal relations are made temporal

Q6 `SELECT *`
`FROM CheckOut`

Assessment of ATSQL

Task 4.1, Task 4.2

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

Table of Contents

- ▶ Comparative analysis
 - ▶ SQL + abstract data types
 - ▶ IXSQL
 - ▶ SQL/TP
 - ▶ TSQL2
 - ▶ ATSQL
- ▶ **Integration of temporal support in DBMS kernel**
 - ▶ Requirements
 - ▶ Algebraic primitives
 - ▶ Reduction rules
 - ▶ SQL implementation
 - ▶ Scaling

Integration of Temporal Support into DBMS Kernel

- ▶ Goal:
 - ▶ Determine and implement in the DBMS kernel the functionality that is required to offer comprehensive temporal support.
- ▶ Solution:
 - ▶ Three requirements that define “comprehensive temporal support”.
 - ▶ Two algebraic primitives that split periods into pieces, such that snapshot reducible queries only need equality for comparison.
 - ▶ The algebraic primitives satisfy the three requirements.
- ▶ The combination of the three requirements is called **sequenced semantics**.

R1. Snapshot Reducibility (SR)

ψ^T is **snapshot reducible** iff

$$\forall t(\tau_t(\psi^T(R_1, \dots, R_n)) \equiv \psi(\tau_t(R_1), \dots, \tau_t(R_n)))$$

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

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

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

- ▶ $D\vartheta_{SUM} \dots \Rightarrow D\vartheta_{SUM}^T \dots$
- ▶ $\bowtie_{\theta} \dots \Rightarrow \bowtie_{\theta}^T \dots$

R2. Extended Snapshot Reducibility (ESR)

ψ^T is **extended snapshot reducible** iff

$$\forall t(\tau_t(\psi^T(R_1, \dots, R_n)) \equiv \pi_E(\psi(\tau_t(\epsilon_{U_1}(R_1)), \dots, \tau_t(\epsilon_{U_n}(R_n))))$$

where $\psi \in \{\vartheta, \sigma, \pi, \times, \bowtie, \bowtie, \bowtie, \bowtie, \bowtie\}$, E = schema of $\psi^T(R_1, \dots, R_n)$

We can access **time intervals** in **snapshot reducible operators**.

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

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

- ▶ at each point in time the average duration of externally funded projects: $D\vartheta_{AVG}^T(DUR(T))$

R3. Change Preservation (CP)

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

$$\begin{aligned} & \forall t, t' \in z.T (L(z, t) = L(z, t')) \wedge \\ & (z.T_S - 1 \in z'.T \wedge z.A = z'.A \Rightarrow L(z', z.T_S - 1) \neq L(z, z.T_S)) \wedge \\ & (z.T_E \in z'.T \wedge z.A = z'.A \Rightarrow L(z', z.T_E) \neq L(z, z.T_S)) \end{aligned}$$

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

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

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

- ▶ $D\vartheta_{SUM}^T(scale(B))$

Temporal Primitives/1

- ▶ To implement the sequenced semantics **two new algebra operators** (primitives) for the **adjustment** of periods are needed:
 - ▶ Temporal normalization $\mathcal{N}$
 - ▶ Temporal alignment ϕ
- ▶ Adjustment respects the lineage.
- ▶ **Reduction rules** from temporal RA to nontemporal RA.
- ▶ **Period propagation** ϵ is possible and makes it possible to access the original periods.

NDBS14, SL04

37/58

M. Böhlen, ifi@uzh

Temporal Primitives/2

- ▶ The purpose of a temporal primitive is to break periods into pieces.
- ▶ Two temporal primitives are required since there are two classes of operators in relational DBMSs:
 - ▶ One input tuple contributes to **at most one** result tuple per time point.
 $\Rightarrow$ **temporal normalization**
 Example: Aggregation
 - ▶ One input tuple contributes to **more than one** result tuple per time point.
 $\Rightarrow$ **temporal alignment**
 Example: Joins

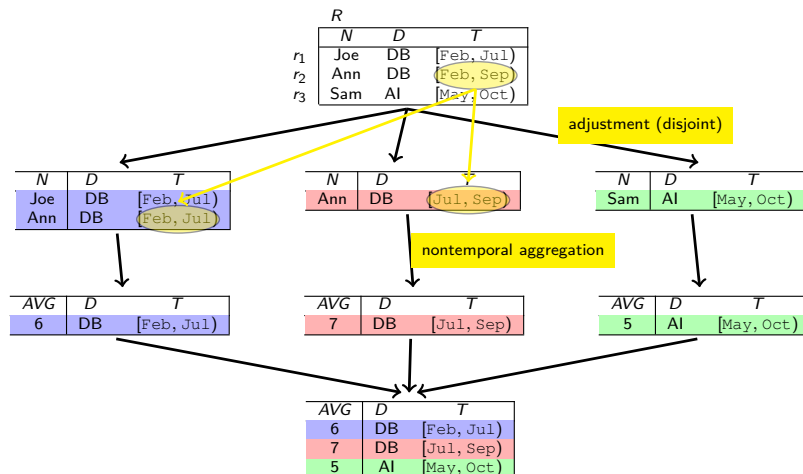
NDBS14, SL04

38/58

M. Böhlen, ifi@uzh

Temporal Normalization

- ▶ Average duration of contracts per department: $D \bowtie_{AVG(DUR(T))}^T(R)$



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

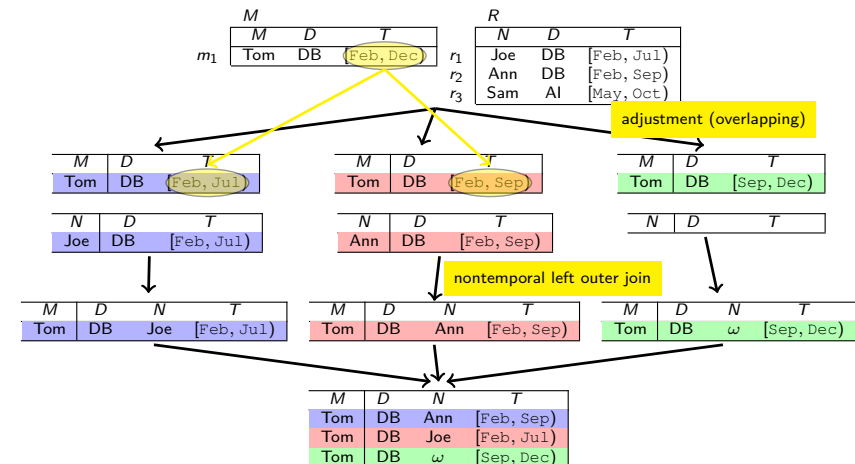
NDBS14, SL04

39/58

M. Böhlen, ifi@uzh

Temporal Alignment

- ▶ Employees managed by manager: $M \bowtie_{M.D=R.D}^T R$



- ▶ One input tuple contributes to more than one result tuple per month. E.g., m_1 contributes twice to month Feb.

NDBS14, SL04

40/58

M. Böhlen, ifi@uzh

Reduction Rules

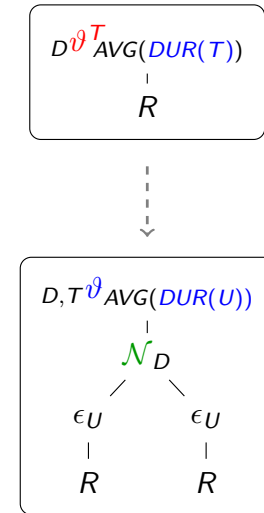
Operator	Reduction	
Selection	$\sigma_{\theta}^T(R)$	$= \sigma_{\theta}(R)$
Projection	$\pi_B^T(R)$	$= \pi_{B,T}(\mathcal{N}_B(R, R))$
Aggregation	$\mathbf{b}_{\mathcal{F}}^T(R)$	$= \mathbf{b}_{\mathcal{F},T}(\mathcal{N}_B(R, R))$
Difference	$R -^T S$	$= \mathcal{N}_A(R, S) - \mathcal{N}_A(S, R)$
Union	$R \cup^T S$	$= \mathcal{N}_A(R, S) \cup \mathcal{N}_A(S, R)$
Intersection	$R \cap^T S$	$= \mathcal{N}_A(R, S) \cap \mathcal{N}_A(S, R)$
Cart. Prod.	$R \times^T S$	$= \alpha(\phi_T(R, S) \bowtie_{R.T=S.T} \phi_T(S, R))$
Inner Join	$R \bowtie_{\theta}^T S$	$= \alpha(\phi_{\theta}(R, S) \bowtie_{\theta \wedge R.T=S.T} \phi_{\theta}(S, R))$
Left O. Join	$R \ltimes_{\theta}^T S$	$= \alpha(\phi_{\theta}(R, S) \ltimes_{\theta \wedge R.T=S.T} \phi_{\theta}(S, R))$
Right O. Join	$R \rtimes_{\theta}^T S$	$= \alpha(\phi_{\theta}(R, S) \rtimes_{\theta \wedge R.T=S.T} \phi_{\theta}(S, R))$
Full O. Join	$R \Join_{\theta}^T S$	$= \alpha(\phi_{\theta}(R, S) \Join_{\theta \wedge R.T=S.T} \phi_{\theta}(S, R))$
Anti Join	$R \triangleright_{\theta}^T S$	$= \phi_{\theta}(R, S) \triangleright_{\theta \wedge R.T=S.T} \phi_{\theta}(S, R)$

α ... temporal duplicate elimination.

Constructing Sequenced Algebra Expressions

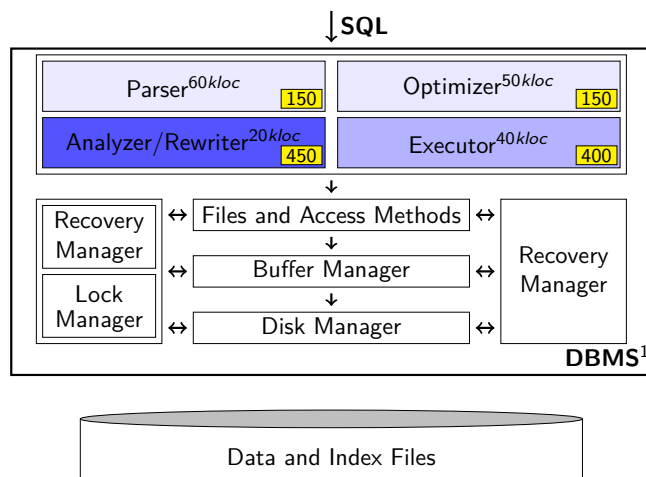
Query: $D \vartheta_{AVG(DUR(T))}^T(R)$

1. Timestamp propagation:
 $D \vartheta_{AVG(DUR(T))}^T(\epsilon_U(R))$
2. Timestamp substitution:
 $D \vartheta_{AVG(DUR(U))}^T(\epsilon_U(R))$
3. Temporal adjustment:
 $\mathcal{N}_D(\epsilon_U(R), \epsilon_U(R))$
4. Nontemporal aggregation:
 $D, T \vartheta_{AVG(DUR(U))}^T(\mathcal{N}_D(\epsilon_U(R), \epsilon_U(R)))$



PostgreSQL Implementation/1

- DBMS kernel integration of temporal primitives.



PostgreSQL Implementation/2

Task 4.6

- SQL extension that provides direct access to primitive operators:

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

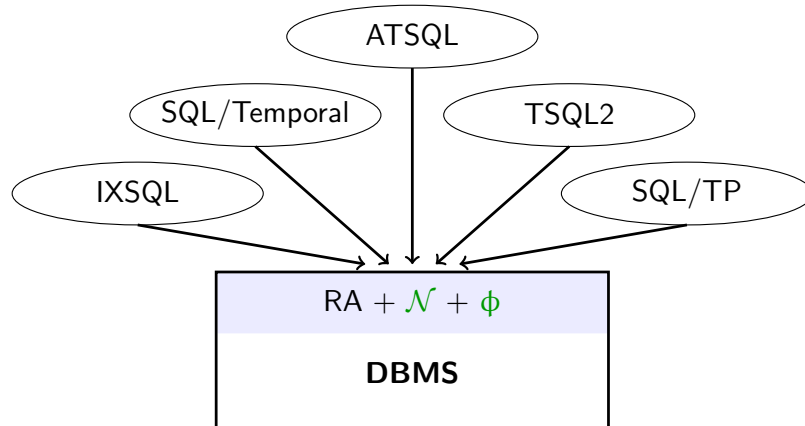
- Temporal SQL languages can be implemented in Parser/Analyzer.

Source Code:

<http://www.ifi.uzh.ch/dbtg/research/align.html>

Algebraic Basis for Sequenced Semantics

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



Timestamp Propagation in SQL

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

R		$\epsilon_U(r)$		
N	T	N	U	T
Ann	[1, 8)	Ann	[1, 8)	[1, 8)
Joe	[2, 6)	Joe	[2, 6)	[2, 6)
Ann	[8, 12)	Ann	[8, 12)	[8, 12)

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

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

Absorb in SQL

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

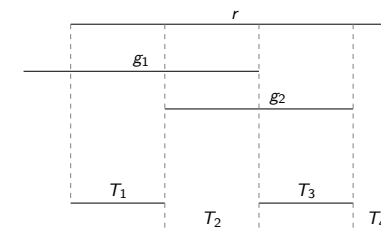
R			$\alpha(R)$		
A	B	T	A	B	T
a	c	[1, 9)	a	c	[1, 9)
a	c	[3, 7)	a	d	[3, 7)
a	d	[3, 7)	b	c	[3, 7)
b	c	[3, 7)	b	d	[3, 7)
b	d	[3, 7)			

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

```
SELECT ABSORB ...
```

Temporal Normalization in SQL

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

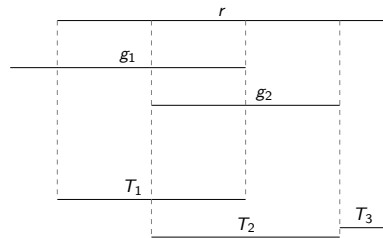


- Algebra: $\mathcal{N}_B(R, S)$
- SQL:

```
... FROM (R NORMALIZE S USING (B)) AS Rnorm ...
```

Temporal Alignment in SQL

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



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

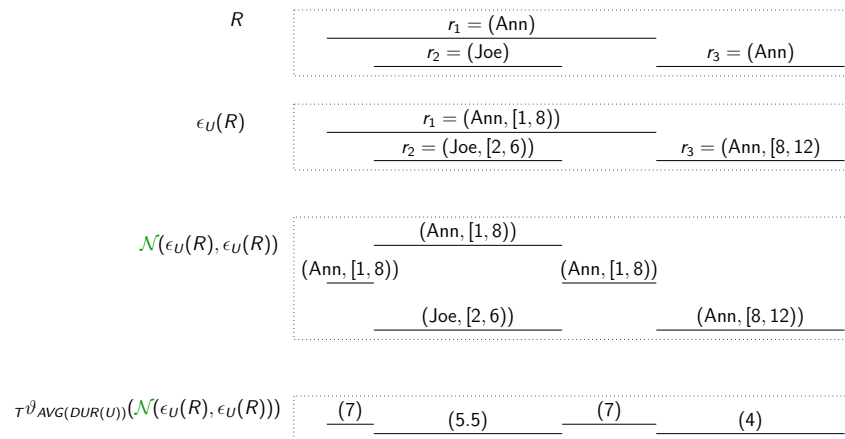
FROM (R ALIGN S ON theta) AS R

Answering Sequenced Queries

Approach for formulating sequenced queries:

- Formulate query without thinking about time/periods; make all operators temporal (e.g., ϑ^T).
 $\vartheta_{AVG(DUR(T))}^T(R)$
- Add copies of periods for periods that are used in conditions.
 $\vartheta_{AVG(DUR(T))}^T(\epsilon_U(R))$
- Replace all references to periods with references to these copies.
 $\vartheta_{AVG(DUR(U))}^T(\epsilon_U(R))$
- Apply reduction rules to get nontemporal relational algebra expression.
 $T\vartheta_{AVG(DUR(U))}(\mathcal{N}(\epsilon_U(R), \epsilon_U(R)))$

Example/1



Example/2

Translate resulting algebra expression to SQL:

- $T\vartheta_{AVG(DUR(U))}(\mathcal{N}(\epsilon_U(R), \epsilon_U(R)))$
- Schema: $R(N, Ts, Te)$
- **WITH x AS (SELECT Ts Us, Te Ue, * FROM R)
SELECT AVG(Ue - Us), Ts, Te
FROM (x NORMALIZE x USING()) Radj
GROUP BY Ts, Te;**

Scaling/1

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

p				
D	N	B	T_S	T_E
DB	1	181K	2013/2/1	2013/8/1
DB	2	196K	2013/5/1	2014/1/1
AI	1	153K	2013/4/1	2013/9/1
AI	2	120K	2013/4/1	2013/9/1

- When aggregating such data according to snapshot reducibility we usually want to scale the funding to the adjusted time periods.
- The result should be as follows:

r			
D	B	T_S	T_E
DB	89K	2013/2/1	2013/5/1
DB	165.6K	2013/5/1	2013/8/1
DB	122.4K	2013/8/1	2014/1/1
AI	273K	2013/4/1	2013/9/1

Scaling/2

- Scaled values are functions of the original value, the old period, and the new period.
- The following function is a simple user-defined function that uniformly scales values.

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

Scaling/3

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

1. propagate periods, ϵ ;
2. normalize, $\mathcal{N}_\theta$ (possibly scale values);
3. possibly scale values and remove propagated attributes;
4. apply the corresponding nontemporal operator, ψ .

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

1. propagate periods, ϵ ;
2. align, ϕ_θ (possibly scale values);
3. apply nontemporal operator, ψ (possibly scale values);
4. possibly scale values and remove propagated attributes.

Scaling/4

Task 4.8

Query: Determine amount of external funding per department.

$$\begin{aligned} & \vartheta_{SUM(scale(B))}^T(p) \\ \Rightarrow & \vartheta_{SUM(scale(B))}^T(\epsilon_U(p)) \\ \Rightarrow_{D,T} & \vartheta_{SUM(scale(B))}^T(\mathcal{N}_D(\epsilon_U(p), \epsilon_U(p))) \end{aligned}$$

```
WITH
P1 AS (SELECT Ts Us, Te Ue, * FROM P),
P2 AS (SELECT * FROM (P1 x NORMALIZE P1 y ON x.D=y.D) P),
P3 AS (SELECT D, N, scaleU(B, Ts, Te, Us, Ue) B, Ts, Te FROM P2)

SELECT  D, SUM(B), Ts, Te
FROM    P3
GROUP BY D, Ts, Te;
```

Summary/1

- ▶ Language design criteria: expressive power, minimality, orthogonality, closedness, clarity, consistency
- ▶ Comparison of periods: convenient but not a major issue
- ▶ Concrete SQL-based query languages
 - ▶ **SQL + abstract data types**: extend SQL with abstract data type; gain is limited
 - ▶ **IXSQL**: two operators to convert periods to points and vice versa; simple and attractive; issues with performance and semantics
 - ▶ **SQL/TP**: only points exist; periods are used internally and for display; based on points, which gives simple semantics; issue with infinity, and presentation
 - ▶ **TSQL2**: use syntactic defaults to easily formulate temporal statements; comprehensive language; no precise semantics
 - ▶ **ATSQL**: use semantic defaults to easily formulate temporal statements; adds modifiers to SQL, which is a significant change

Summary/2

- ▶ New algebraic primitives in the DBMS kernel that provide support for the sequenced semantics.
 - ▶ normalize
 - ▶ align
 - ▶ extend
 - ▶ absorb
- ▶ Direct mapping of the primitives to SQL for illustration purposes. Other SQL extensions are possible.
- ▶ Change point preservation makes it possible to scale attribute values to fit adjusted periods.