Temporal Extensions of SQL

David Toman

Dept. of Computer Science
University of Toronto

Why Yet Another Temporal Extension of SQL?

(1) To avoid semantic problems in TSQL

(2) To provide a *natural* extension of SQL

Plan of the talk:

- Temporal Databases (+ running example)
- Problems with TSQL
- SQL/TP
  - ⇒ Syntax & Semantics
  - ⇒ Examples
- SQL/TP Query Evaluation

List of Slides

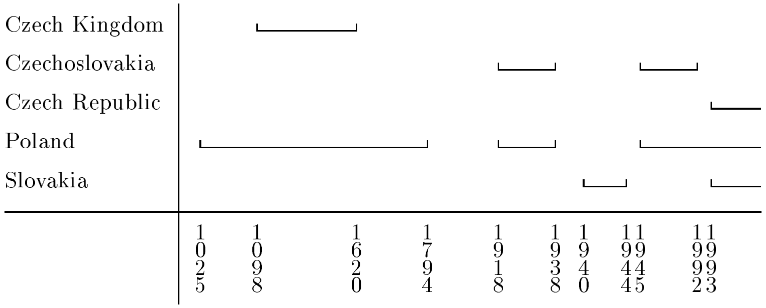
Notes:

The currently most prominent proposals of temporal extensions of SQL are based on TSQL derivatives.

This choice would “standardize” all problematic issues present in TSQL’s semantics (e.g., representation-dependent queries).

Also such an extension introduces very cumbersome syntax—the constructs depend on the concrete encoding of sets of time instants.

Running Example



Notes:

Graph plots the independence relation for various Central European countries.

The time is integer-like with granularity 1 year.

The graph itself is infinite (due to the open periods on the right side).

We could have chosen different granularity or dense order with only minor changes.

Abstract Temporal Databases

Relational DB

+

Temporal Domain $(T, <)$

⇒ Discrete linear order

⇒ Infinite relations

with *finite* duplication

indep	
Name	Year
Czech Kingdom	1198
...	...
Czech Kingdom	1620
Czechoslovakia	1918
...	...
Czechoslovakia	1938
Czechoslovakia	1945
...	...
Czechoslovakia	1992
Czech Republic	1993
...	...
Slovakia	1938
...	...
Slovakia	1944
Slovakia	1993

Notes:

The relation is equivalent to the previous graph and is infinite in general.

The abstract relations capture the content of the data.

Even if we insisted on closed periods in the graph a finer granularity would lead to much "bigger" relations.

We allow only finite duplication to make relational operations meaningful.

Concrete TDB: Interval Encoding

- Compact encoding of "validity" periods
- Semantic mapping $\|.\|$
- Images under $\|.\|$ $\Rightarrow$ finitary databases

indep	
`name`	`yearmin` ... `yearmax`
Czech Kingdom	1198 ... 1620
Czechoslovakia	1918 ... 1938
Czechoslovakia	1945 ... 1992
Czech Republic	1993 ... ∞
Slovakia	1940 ... 1944
Slovakia	1993 ... ∞
Poland	1025 ... 1794
Poland	1918 ... 1938
Poland	1945 ... ∞

Notes:

Concrete relations: finite encodings of (subclass) of the abstract relations (finitary ones).

$\|.\|$ keeps us straight during query evaluation.

Not necessarily coalesced: the $\|.\|$-image is a *additive* bag-union of images of the individual tuples. This allows duplicates.

Temporal Query Languages

- Point-based:
  - Two-sorted First Order Logic (Calculus)
  - First Order Temporal Logic (FOTL)
  - Temporal Relational Algebra (TRA)
- Point-based, but with attributes ranging over intervals:
  - TQuel
  - TSQL, TSQL/2, ATSQL2, SQL/Temporal
- Interval-based:
  - ?

Notes:

The first group of languages defines its semantics over the abstract databases: clearly representation-independent (no encoding is involved), but not too practical: works over infinite or extremely large relations.

The second group defines its semantics over the concrete databases while trying to mimic a point based semantics over the meanings (not very successfully—we see examples later).

The third group is essentially a two-dimensional version of (1); anyway, we'll see that we can't restrict ourselves to a fixed dimension and thus (1) and (3) are essentially the same.

From SQL to TSQL

SQL over concrete temporal databases:

- temporal attributes range over intervals
- selection uses Allen's interval algebra operators
- implicit interval operations
  $\Rightarrow$ intersection during joins
- implicit coalescing of temporal attribute(s)
  $\Rightarrow$ during updates
  $\Rightarrow$ during query evaluation
- fixed temporal dimension (1 or 2)

Notes:

TSQL is essentially SQL over the concrete temporal databases.

This description more or less characterizes all the languages in the second group (the "practical" ones).

Simple to implement (disregarding the higher-dimensional indices and joins, which we can't avoid in any case).

TSQL Example

"List all countries that lost and regained their independence"

$$\{n \mid \exists t_1, t_2, t_3.\quad t_1 < t_2 < t_3 \wedge \texttt{indep}(n, t_1) \wedge \neg\texttt{indep}(n, t_2) \wedge \texttt{indep}(n, t_3)\}$$

TSQL2:

```
select   r.name
from     indep r, indep s
where    r.name=s.name
  and    validtime(r) precedes validtime(s)
```

Does not work in general:

$\Rightarrow$ `indep` has to be coalesced and can not contain duplicates.

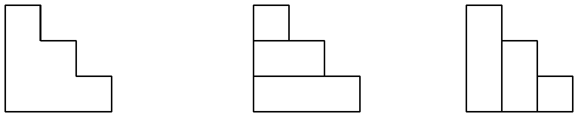
Notes:

In the above example we pay for the third join during updates: we need to re-coalesce (also `indep` may be a result of a query—even more problematic).

Problems with TSQL

Known facts:

- there is no unique coalescing for temporal dimension > 1
- FO-complete query languages can not avoid arbitrarily large dimensionality in queries [AHVb96, TN96]



- queries depend on representation
  even worse for duplicate semantics and aggregation

Notes:

The figure represents a two-dimensional time region (e.g., an answer to a previous query) and the two standard ways of coalescing its interval-based representation.

It is very difficult to mimic point-based semantics in a language with interval-typed attributes (see ex. above).

Coalescing doesn't save us (it would if we could stick to dimension 1 (but we can not if we want all FO queries to be expressible).

Extremely hard to avoid representation dependency.

Aggregation is even worse (e.g., counting the # of regions along the axes).

This is inherent to all languages with point-based semantics and attributes ranging over the encoding (intervals).

An alternative: SQL/TP

$\boxed{\text{Temporal attributes range over individual time instants}}$

- Syntax and Semantics:
  $\Rightarrow$ an extension of SQL by a new data type
  $\Rightarrow$ supports finite duplication
- Encoding of relations:
  $\Rightarrow$ compact representation of temporal relations using intervals
  $\Rightarrow$ syntactic restrictions to maintain closure (safety)
- Query evaluation:
  $\Rightarrow$ efficiency: depends on the (size of the) encoding only
  $\Rightarrow$ allows compilation to SQL/92

Notes:

The goals:

(1) The syntax and semantics of standard SQL has to be extended in a *natural way*: by adding a new data type `time` that behaves (almost) identically as the existing data types, and

(2) The syntax and semantics are *independent* of the chosen encoding of timestamps.

The chosen language is derived from calculus with finite duplicates.

The choice of encoding is not critical: we aim on subsuming TSQL, so we have chosen TSQL's encoding of temporal databases.

Efficient of query evaluation means that it uses only elements in the active domain of the encoding (plus some small neighborhoods of `Time` values).

SQL/TP: syntax

```
<exp>     ::= <ddl_exp> | <dql_exp> | <dml_exp>

<ddl_exp> ::= create table <rid> ( <signat> ) |
              create view  <rid> ( <dql_exp> )

<dql_exp> ::= select [all] <s_list> [from <f_list>]
                     [where <cond>] [group by <g_list>] |
              ( <dql_exp> ) union [all] ( <dql exp> ) |
              ( <dql_exp> ) except [all] ( <dql exp> ) |
              ( <dql_exp> ) intersect [all] ( <dql exp> )

<dml_exp> ::= insert [all] into <rid> ( <dql_exp> ) |
              delete [all] from <rid> ( <dql_exp> )
```


Notes:

Standard SQL, all nesting of queries is realized in the `from` clause.

The `having` clause and nesting in the `where` clause are easily definable using nesting in the `from` clause and thus are considered to be just a syntactic sugar.

Where is the "temporal extension"? In the data definition language: when we define an attribute of type `time` (next slide)...

SQL/TP: encoding of time

```
<type>    ::= time [using <enc>] | integer | char(N) | ...
```

Modifiers `<enc>`:

- `points`
- `bounded intervals`
- `unbounded intervals`

Other classes of constraints can be readily incorporated

Notes:

`points` is for atomic events (e.g., transaction time).

`bounded intervals` is for facts with bounded "duration".

`unbounded intervals` is for facts with unbounded duration (e.g., x holds from now on)

Easy to introduce other constraint classes (back-door for the CDB folks): If you can't beat them (SQL), join them.

SQL/TP: semantics

- standard SQL semantics

  ⇒ select-project-join, aggregation, difference, union
  ⇒ infinite bags w/finite duplication

- no duplicate-preserving projection of unbounded attributes:

  ⇒ `select all` name `from indep`

  Result:
  $$\{\ldots, \underbrace{(\text{Poland}), \ldots, (\text{Poland})}_{\infty}, \ldots\}$$

- duplicate preserving projection of *bounded* types may be allowed

Notes:

The semantics of the new language is exactly SQL semantics on the $\|.\|$-images (the semantics is naturally extended to infinite relations).

We need to restrict projections to maintain finite duplication (infinite duplication is not very interesting nor useful).

Note the difference between infinite and unbounded (we can "measure" infinite bounded sets, e.g., in dense case; we may get non-integral "counts", e.g., 1.5 years)

SQL/TP: features

- representation independence
- simple syntax and semantics
- fulfills migration requirements (ISO/ANSI)
- first-order complete
- can express all representation-independent TSQL queries

- easily extensible to other encodings
- has an efficient query evaluation for interval-based encoding

- ...

Notes:

Semantics is defined on the abstract relations.

Syntax and semantics are taken straight from SQL.

Allows execution of non-temporal queries at the current point or as LISP's "map" for every time instant.

Examples

"List all countries independent while Czech Kingdom was independent".

```
select  r1.name
from    indep as r1, indep as r2
where   r2.name = 'Czech Kingdom' and r1.year = r2.year
```

Result:

```
name
--------------------
Czech Kingdom
Poland
```


Notes:

This query can is a single PSJ query (a single select block). Note also, that the result is a standard non-temporal relation.

The equality between the temporal attributes needs to be "compiled" into selection condition depending only on the endpoints (however, this is done automatically behind the scene).

Examples (cont.)

"List all periods when no country was independent".

```
( select t as year from true ) except
                    ( select year from indep )
```

Result:

```
  yearmin      yearmax
----------------------
-infinity         1024
     1795         1917
     1939         1939
```


Notes:

The query can be answered by a complementation of the remainder of the projection on `year` from the `indep` relation. Note the use of the `true` pseudo-relation to achieve signature compatibility.

Note that the results of the queries are *concrete* temporal relations.

Extremely hard in TSQL (go and try, if you don't believe me!)

Example 2 in (T)SQL

```
select r1.yearmax+1, r2.yearmin-1
from   indep r1, indep r2
where  r1.yearmax+1 <= r2.yearmin+1
  and  not exists
    ( select *
      from   indep r3
      where  ( r1.yearmax < r3.yearmin and
                        r3.yearmin < r2.yearmin )
         or  ( r1.yearmax < r3.yearmax and
                        r3.yearmax < r2.yearmin )
    )
```


Notes:

Examples (cont.)

“List all countries that lost and regained their independence”.

```
select   r1.name
from     indep r1, indep r3,
( ( select  name, true.t as year from true, indep )
    except ( select  name, year from indep ) ) r2
where    r1.name = r2.name and r2.name = r3.name and
         r1.year < r2.year and r2.year < r3.year
```

Result:

```
name
--------------------
Czechoslovakia
Slovakia
Poland
```


Notes:

This query is a three-way self join, where one of the tables is complemented. Cf. the slides in the beginning of the talk.

Also it naturally corresponds to the calculus formula.

Examples (cont.)

"List all countries that were independent before Slovakia".

```
select  name
from    indep, ( select min(year) as y0
                 from   indep
                 where  name = 'Slovakia' )
where   year < y0
```

Result:

```
name
--------------------
Czech Kingdom
Czechoslovakia
Poland
```


Notes:

The `min` and `max` aggregates allow us to find the extreme points in time (and the cost is exactly the same as in TSQL).

Note that we can not find "endpoints of intervals" this way. This is a *feature* that prevents us from writing representation-dependent queries.

Examples (cont.)

"For every country list the number of years of independence during the 20th century"

```
select   name, count(year) as years
from     indep
where    1900  <=  year  <  2000
group by name
```

Result:

```
name                      years
------------------------------
Czechoslovakia               67
Czech Republic                7
Slovakia                     11
Poland                       75
```


Notes:

Counting along the temporal dimensions leads to measuring the lengths of intervals (and volumes of hyper-rectangles in general).

The aggregation is possible due to the `where` clause: it restricts the otherwise unbounded attribute `year`.

Examples (cont.)

“Every year list the number of independent countries”

```
select year, count(name) as num from indep group by year
```

Result:

yearmin	yearmax	num
1025	1197	1
1198	1620	2
1621	1794	1
1918	1938	2
1940	1944	1
1945	1992	2
1993	infinity	3

Notes:

Grouping by the temporal dimension is much harder: it is not clear, how this result can be directly obtained in TSQL.

Coalescing does not help to answer this query: the input is coalesced and re-coalescing after projecting out the `name` attribute would erase the needed duplicates. This is the crucial difference between coalescing and our normalization procedure.

SQL/TP over Intervals: closure

Queries have to preserve closure over the class of finitary databases

$\Rightarrow$ Syntactic (safety) restrictions for SQL/TP queries:

- top level signature: all attributes independent [CGK96]
- aggregated attributes independent of grouping attributes
- no duplicate preserving projections of unbounded attributes

Notes:

Can all SQL/TP queries be asked over interval encoding? NO: the results must be representable as concrete relations. Similar to safety restrictions.

The attribute independence is required *only* for the attributes in the answer, not for all attributes in the query.

Note that most temporal query languages allow only a single temporal attribute in the answers: in these cases the condition is trivial (true).

Queries and Duplicates

Duplicate-preserving projection (for bounded types)

$\Rightarrow$ problem with exponential blowup:

$$\pi_{\texttt{name}}(\underbrace{\{(a,[0,2^n-1])\}}_{|a|+n}) = \underbrace{\{(a),\ldots,(a)\}}_{|a|\cdot 2^n}$$

$\Rightarrow$ disallowed

$\Rightarrow$ aggregate queries can often be rewritten w/o such projections:

$$\begin{aligned}&\mathrm{count}(\pi_{\texttt{name}}(\sigma_{1800\le t<1900}(\texttt{indep})) - \pi_{\texttt{name}}(\sigma_{1900\le t<2000}(\texttt{indep}))) = \\ &\quad \mathrm{count}(\pi_{\texttt{name}}(\sigma_{1800\le t<1900}(\texttt{indep}))) - \mathrm{count}(\pi_{\texttt{name}}(\sigma_{1900\le t<2000}(\texttt{indep})))\end{aligned}$$

Notes:

This is a tradeoff needed for efficiency (allowing the above projection would lead to exponential explosion).

The only legitimate use of duplicates is for aggregation: we use a transformation to avoid the actual creation of the duplicates by keeping explicit counts (the example isn't quite right: the subtraction has to be restricted to positive integers only).

We allow duplicate-preserving projection of buunded interval attributes is the duplicates are later "consumed" by an aggregate. However, if they should appear in the answer to the query, we must reject the query; otherwise we would have to produce the duplicates explicitly—not possible.

SQL/TP: Query Evaluation

$$\text{compile} : \text{SQL/TP} \longrightarrow \text{SQL/92}$$

$$\begin{tikzcd}
\boxed{\text{Abstract TDB}} \arrow[d, "Q_{\text{SQL/TP}}"] & \boxed{\text{Concrete TDB}} \arrow[l, "\|\cdot\|"'] \arrow[d, "Q'_{\text{SQL/92}} = \text{compile}(Q)"] & \boxed{\text{Physical DB}} \arrow[l] \arrow[d, "\text{eval}(Q')"] \\
\boxed{\text{Abstract Relation}} & \boxed{\text{Concrete Relation}} \arrow[l, "\|\cdot\|"'] & \boxed{\text{Physical Relation}} \arrow[l]
\end{tikzcd}$$

Notes:

So far we only hinted about the query evaluation of SQL/TP queries over the concrete databases (note that the semantics is defined over the abstract databases.

We chose compilation: easy implementation on top of an existing RDBMS.

Also we identify operations that we may want to add to a RDBMS to facilitate more efficient evaluation of temporal queries.

Data Definition Language

```
create table indep ( Name char(20), year time )
```

$\downarrow$ compile

```
create table indep ( Name char(20),
                     yearmin Time, yearmax Time)
```


Notes:

The data definition language is extremely easy to translate: we just replace the abstract signature with its concrete counterpart.

`Time` is an UDT (based on integers or a `DATE` type of the DBMS).

Conditional Queries

Subqueries of a SQL/TP query may not be attribute independent:

$\Rightarrow$ Conditional queries: $Q[\varphi]$

- Q is a SQL/92 query
- φ is a quantifier free formula in $\mathcal{L}_<$

Translation Invariant:

$$\text{compile}(Q) = \{Q_1[\varphi_1], \ldots, Q_n[\varphi_n]\}$$
$$\text{such that } \; Q(\|D\|) = \sigma_{\varphi_1}\|Q_1(D)\| \cup \ldots \cup \sigma_{\varphi_n}\|Q_n(D)\|$$
$$\text{and } \; \mathcal{L}_< \not\models \varphi_i \wedge \varphi_j, \; 0 < i < j \leq n$$

Notes:

While the top level temporal attributes have to be independent, this may not be true for all the subqueries. The conditional formulas allow us to propagate and eventually eliminate such dependencies (they are constraints "global" for the whole subquery: in a CDB approach such a constraint would be attached to every tuple in the answer to the subquery).

"compile" is defined by structural induction on the syntax of SQL/TP queries in such a way that it preserves the above invariant.

Select Block: Join and Selection

...	...	...
Czech Kingdom	1098	1620
...	...	...

$\times$

Poland	1025	1794
...	...	...
Slovakia	1940	1944

$\Downarrow$

...	...	...	...	...	...
Czech Kingdom	1098	1620	Poland	1025	1794
Czech Kingdom	1098	1620	Slovakia	1940	1944
...	...	...	...	...	...

$\Downarrow \sigma_{n_1='\text{Czech Kingdom}', t_1\mathtt{min} \leq t_2\mathtt{max}, t_2\mathtt{min} \leq t_1\mathtt{max},}$

...	...	...	...	...	...
Czech Kingdom	1098	1620	Poland	1025	1794
...	...	...	...	...	...

Notes:

Select block is translated in two steps: first we translate the product–selection part (cf. query in Example 1).

The selection condition on the temporal attributes is derived using quantifier elimination of the point attributes from

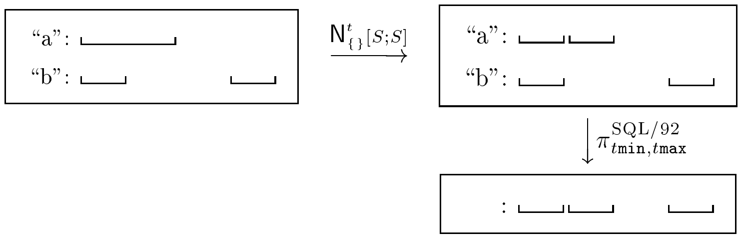
$$\exists t_1, t_2 . t_1\mathtt{min} \leq t_1 \leq t_1\mathtt{max} \wedge t_2\mathtt{min} \leq t_2 \leq t_2\mathtt{max} \wedge t_1 = t_2$$

The input tables may be results to subqueries: we need to add the conditional formulas associated with these subqueries.

The product-selection is converted to joins by the underlying DBMS as for any other query.

Also we derive a resulting condition by eliminating all but the point temporal attributes (in this case $t_1 = t_2$).

Select Block: Duplicate Elimination



Notes:

The second step is the projection, duplicate elimination (and aggregation, if needed).

The top left and right tables are $\|.\|$- equivalent.

The standard projection with duplicate elimination applied on the top right table produces the desired result.

The normalization N guarantees that the set/bag operation of SQL (e.g. projection with duplicate elimination) works correctly.

Time-compatible Queries

Let $\{Q_1, \ldots, Q_k\}$ be a set of queries with compatible signatures, X a subset of their data attributes, and t a temporal attribute.

$Q_1, \ldots, Q_k$ are *t-compatible on X* if

$$a \in Q_i(D) \text{ and } b \in Q_j(D) \text{ such that } \pi_X(a) = \pi_X(b)$$
$$\text{then } \pi_{\{t\}}(\|a\|) \text{ and } \pi_{\{t\}}(\|b\|) \text{ are identical or disjoint.}$$

$Q_1, \ldots, Q_k$ are *time-compatible* on X if $Q_1, \ldots, Q_k$ are t-compatible on X for all temporal attributes t in the common signature.

There are first order queries $\mathsf{N}^t_X[Q_i; Q_1, \ldots, Q_k]$ such that

(1) $\|Q_i(D)\| = \| \mathsf{N}^t_X[Q_i; Q_1, \ldots, Q_k](D)\|$
(2) $\{\mathsf{N}^t_X[Q_i; Q_1, \ldots, Q_k] : 0 < i \leq k\}$ are t-compatible

Notes:

The t-compatibility guarantees that the interval values of t behave like points: they are identical when the representation is identical and disjoint when the representation differs.

This approach can (and is) used for all set/bag operations. It also allows the alternative use of merge-joins (cf. intersection) in the `from` clause (this is too tricky to be found by the underlying DBMS).

Select Block: Aggregation

a	$\texttt{max}(a)$	$\texttt{min}(a)$	$\texttt{count}(a)$	$\texttt{sum}(a)$
`data`	$\texttt{max}(a)$	$\texttt{min}(a)$	$\texttt{sum}(C)$	$\texttt{sum}(a \cdot C)$
`temporal`	$\texttt{max}(a\texttt{max})$	$\texttt{min}(a\texttt{min})$	$\texttt{sum}(C)$	N/A

Notes:

The aggregation uses the N in a similar fashion: cf. Example 6.

The C value is the measure of the number of duplicates represented by the non-grouped part of a tuple (different from tuple to tuple). It can be computed by an expression depending only on the endpoints (we also need to account for the conditionality of queries here).

For handling NULL values we can use conditional expressions.

Set Operations

Q_1 <setop> Q_2 is translated using the following Lemma:

Lemma. Let $\{Q_1, Q_2\}$ be time-compatible. Then

$$\|Q_1\| \ \texttt{op} \ \|Q_2\| = \|Q_1 \ \texttt{op} \ Q_2\|$$

where op $\in \{\cap, \cup, -\}$ is a set operator on relations.

$\Rightarrow$ compile(Q_1 op Q_2) $= \mathsf{N}[Q_1; Q_1, Q_2]$ op $\mathsf{N}[Q_2; Q_1, Q_2]$
$\Rightarrow$ in $O(n \log n)$ of the combined size of Q_1 and Q_2

Notes:

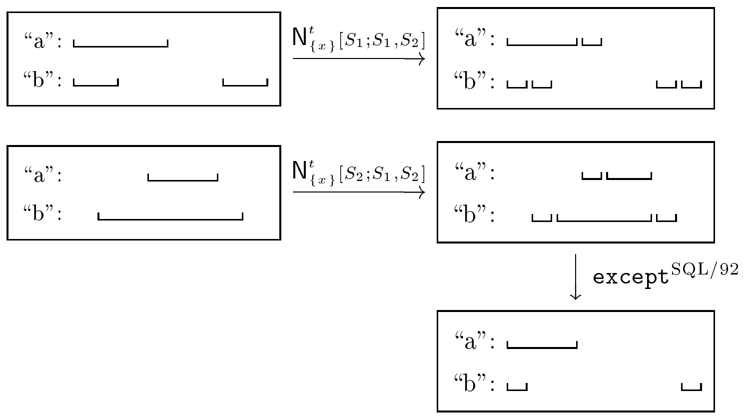
The set operations are based on N similarly to duplicate elimination.

N not necessary in the case of additive union (`union all`).

The intersection technique can be used for merge-joins in the translations of the select block.

N can be based on sorting: the cost may amortize in subsequent join and duplicate elimination steps.

Set Operations (cont.)



Notes:

This example shows how a difference is computed using N.

Note that it preserves duplicates, if necessary.

Set Operations (cont.)

$$
\begin{array}{lcll}
\{Q_i[\varphi_i]\}\ \texttt{union}\ \{R_j[\psi_j]\} & \mapsto & \{ & Q_i[\varphi_i \wedge \neg \bigwedge_j \psi_j], \\
 & & & R_j[\psi_j \wedge \neg \bigwedge_i \varphi_i], \\
 & & & Q_i \cup R_j[\varphi_i \wedge \psi_j] \quad \} \\
\{Q_i[\varphi_i]\}\ \texttt{except}\ \{R_j[\psi_j]\} & \mapsto & \{ & Q_i[\varphi_i \wedge \neg \bigwedge_j \psi_j], \\
 & & & Q_i - R_j[\varphi_i \wedge \psi_j] \quad \} \\
\{Q_i[\varphi_i]\}\ \texttt{intersect}\ \{R_j[\psi_j]\} & \mapsto & \{ & Q_i \cap R_j[\varphi_i \wedge \psi_j] \quad \}
\end{array}
$$

Notes:

The conditional queries are handled using the above rules: the goal is to push the set operation towards the unconditional parts of the queries.

All the conditional formulas are computed at compile time.

This transformation is at most quadratic (not exponential).

Size of the Translated Query

- the size of the result:

$$|\,\mathrm{compile}(Q)| \in O(2^{|Q|})$$

  $\Rightarrow$ exponential in depth (nesting) of the query.

- views:

```
create view <rid> as ( <query> )
```

  $\Rightarrow$ views have to obey the attribute independence requirement
  $\Rightarrow$ for queries that can be decomposed into *small* views:

$$|\,\mathrm{compile}(Q)| \in O(|Q|)$$

Notes:

This is slightly unpleasant. However, the exponentiality is in the depth of nesting, which is usually quite low.

Moreover, there aren't too many conditional formulas in low dimensions. Essentially, the conditional formulas define hyper-stripes parallel to the diagonal, with an offset defined by a constant present in the query.

Also for large classes of queries (e.g., queries equivalent to a query in TSQL, TRA, or FOTL) the exponential blowup can be avoided altogether.

Optimization

1. reduce the number of conditional queries:

   ⇒ combine queries with the same condition can in a single query:

   $$Q_1[\varphi], Q_2[\varphi] \to Q_1 \cup Q_2[\varphi]$$

   - for 1-dimensional time: the only condition is true
   - for 2-dimensional time: $t_1 < t_2 + c$
     ⇒ diagonal stripes

2. eliminate redundant N operations:

   $Q_1, \ldots, Q_k$ are t-compatible on X

   $$\Rightarrow Q_i(D) = \mathsf{N}[Q_i; Q_1, \ldots, Q_k](D)$$

3. eliminate redundant duplicate elimination.

Notes:

(1) The merging is possible due to the observation on the previous slide: there are relatively few different conditional formulas.

(2) N is (almost) idempotent (similar to rules for projection). thus we can use algebraic rewrites to limit the use of N to the necessary minimum (also we could have maintain the base relations normalized (w.r.t. frequent queries) and this way shift the burden of normalization to updates. We have rules that tell us how the t-compatibility interacts with relational operators.

(3) In many cases we do not have to perform duplicate elimination (however, its more complex than in the SQL case). We can detect these cases and remove unnecessary duplicate elimination steps automatically (this is also the reason for having the `select` clause to eliminate duplicates by default).

Future Work

- More complex temporal domain
- More expressive encoding
- Advanced optimization techniques
- Updates
- Indexing
- Tradeoff between defining N as SQL expression and a native implementation.

Notes:

Conclusion

1. New approach to temporal extension of SQL.
2. Clean syntax and semantics including finite duplicates.
3. Efficient implementation.
4. Translation to SQL/92.
5. Localization of extensions needed for temporal query processing to a single operation N.

Notes:

The database vendors ought to love this, especially points (4) asd (5).