4. Update Propagation in SQL
Update Propagation in SQL

• We have seen how update propagation can be performed using Datalog.

• For a real implementation, however, it is necessary to transform specialized algebra equations into SQL views!

• Obviously, this must be possible but the following questions remain:

  1. How ought specialized equations to be transformed in SQL views?
  2. What transformations lead to the most efficient delta views?
  3. Which particularities of SQL have to be taken into account?
  4. How can aggregate functions be specialized?
4.1 Incremental SQL Views
How can update propagation be used for realized push-based stream processing in SQL?
From SQL to Datalog and Back

General scheme for specializing SQL expressions:

SQL

CREATE VIEW ..
SELECT * FROM ...

CREATE VIEW ..
SELECT * FROM ...

CREATE VIEW ..
SELECT * FROM ...

Datalog

p(X,Y) ← q(X,Z), not e(X), ...
p(X,Y) ← ...

q(X) ← w(X,Y,Z), ...
s(Y) ← s(X,Z), ....

Datalog *

p*(X,Y) ← q*(X,Z), not e(X), ...
p*(X,Y) ← ...

q*(X) ← w*(X,Y,Z), ...
s*(Y) ← s*(X,Z), ....

Specialized SQL

CREATE VIEW P+
SELECT * FROM ...

CREATE VIEW Q-
SELECT * FROM ...

CREATE VIEW S+
SELECT * FROM ...
From Datalog to SQL - 1

Translating Datalog rules into SQL views is simple:

\[
p^+(X,Y) \leftarrow q^+(X,Z), \ s(Z,Y), \ not \ e(X,Y), \ not \ r(Z).
p^+(X,Y) \leftarrow s^+(X,Y)
\]

```
CREATE VIEW P+ AS
SELECT Q+.X, S.Y
FROM Q+, S
WHERE Q+.Z=S.Z AND
    NOT EXISTS (SELECT * FROM E WHERE E.X=Q+.X and E.Y=S.Y) AND
    NOT EXISTS (SELECT * FROM R WHERE R.Z=S.Z)
UNION
SELECT *
FROM S+
```
General scheme:

\[ A \leftarrow L_1, \ldots, L_n, \text{ not } B_1, \ldots, \text{ not } B_m. \]

\[ A \leftarrow \ldots \]

CREATE VIEW pred(A) AS
SELECT vars(A)
FROM pred(L_1), \ldots, pred(L_n)
WHERE joincondition(L_1, \ldots, L_n) and
  NOT EXISTS (SELECT * FROM pred(B_1) WHERE joincondition(L_1, \ldots, L_n, B_1)) AND
  NOT EXISTS (SELECT * FROM pred(B_1) WHERE joincondition(L_1, \ldots, L_n, B_2)) AND
  \ldots
  NOT EXISTS (SELECT * FROM pred(B_1) WHERE joincondition(L_1, \ldots, L_n, B_m)) AND
UNION
SELECT vars(A)
FROM \ldots
There are alternatives to the general scheme, though:

A ← . . ., not $B_1$, ..., not $B_m$.

\[
\text{CREATE VIEW pred(A) AS } \\
\ldots \\
\text{NOT EXISTS (SELECT * FROM pred(B)_1, ..., pred(B)_m) WHERE joincondition(L}_1, ..., L}_n, B}_1)) \text{ OR joincondition(L}_1, ..., L}_n, B}_2)) \text{ OR} \\
\ldots \\
\text{joincondition(L}_1, ..., L}_n, B}_m))
\]

or

\[
\text{CREATE VIEW pred(A) AS } \\
\ldots \\
\text{vars(B}_1) \text{ NOT IN (SELECT * FROM pred(B}_1)) AND vars(B}_2) \text{ NOT IN (SELECT * FROM pred(B}_2)) AND} \\
\ldots \\
\text{vars(B}_m) \text{ NOT IN (SELECT * FROM pred(B}_m)) AND}
\]
Translating SQL views (even without aggregate functions) into Datalog rules is difficult:

```
CREATE VIEW P AS
(SELECT *
FROM Q, (SELECT * FROM ... AS R WHERE Q.X>R.B OR
    NOT EXISTS (SELECT * FROM T WHERE T.X<>Q.X or ) AND
    Q.Y>ALL (SELECT * FROM ...)
INTERSECT/MINUS
(SELECT *
FROM ...
```

Basic SQL expressions can be easily translated, though.
Basic SQL

• There is a „core“ of SQL
  - that can be directly transformed into Datalog and
  - into which all more complicated syntactical expressions can be transformed.

• Which transformations exist? Which syntactical forms in SQL „survive" in Basic SQL?

• **Attention!** In the following discussion we assume that all relations are duplicate-free, i.e., SELECT corresponds to SELECT DISTINCT.
Basic SQL (2)

- The **UNION-operator** is indispensable, but not **OR** in **WHERE-Parts**.

- OR can *always* be equivalently replaced by UNION:

```sql
SELECT r.A
FROM   r
WHERE  r.B=0 OR r.C=0
UNION
( SELECT r.A FROM r
WHERE  r.B=0)
UNION
( SELECT r.A FROM r
WHERE  r.C=0)
```

- In contrast, UNION cannot be simulated with OR if the operators are related to different relations:

```sql
(SELECT r.A FROM r)
UNION
(SELECT s.A FROM s)
```
Basic SQL (3)

- In a similar way it can be shown that:
  - INTERSECT **cannot** be „simulated“ with AND.
  - AND can **always** be expressed using INTERSECT.

- AND can **also** be replaced by INTERSECT, if it occurs as a "Partner" of NOT (because NOT is monadic).

- **Negation of comparisons** can be replaced by inverting the comparator:
Basic SQL (4)

- Following on this argumentation line it is easy to show the following:

<table>
<thead>
<tr>
<th>dispensable:</th>
<th>indispensable:</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>UNION</td>
</tr>
<tr>
<td>AND</td>
<td>MINUS</td>
</tr>
<tr>
<td>NOT</td>
<td>Product in FROM-Part</td>
</tr>
<tr>
<td>EXISTS</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>INTERSECT</td>
<td></td>
</tr>
<tr>
<td>Nesting of blocks</td>
<td></td>
</tr>
</tbody>
</table>

- Note that we still don’t have discussed all SQL concepts. Especially
  - **Null-values**
  - **Aggregate-functions and grouping**

  which would make things much more complicated.

- However, aggregate functions will be discussed at the end of this section.
Basic SQL (5)

- Thus, we could implement everything using UNION, MINUS and the implicit product but that would lead to scattered code.

- Therefore, we allow for AND, (NOT) EXISTS etc. as long as they are used in a Datalog compliant way, e.g.:

```
SELECT r.A
FROM   r,s
WHERE  r.B=S.A AND S.C>0
```

\[ r(A) ← r(A,B), s(B,C), C>0 \]
4.2 SQL2Datalog: Transformation Rules
CREATE VIEW p AS

SELECT A
FROM t
WHERE B>0 AND C=D AND NOT D=E

Table t:
SQL-Style: attributes, no positions
A, B, C, D, E
Datalog-Style: positions, no attributes
Correspondence:

Direct translation into Datalog rule:

p(X) ← t(X, Y, Z, V, W), Y>0, Z=V, not V=W.

More compact rule by expressing (consequences of) equation using the same variable:

p(X) ← t(X, Y, Z, Z, W), Y>0, not Z=W.
CREATE VIEW p AS

\[
\text{SELECT } A \\
\text{FROM } t \\
\text{WHERE } B > 0 \text{ AND } C = D \text{ AND NOT } D = E
\]

TRC: Tuple Relational Calculus

Implicit tuple variable in query:

\[
\text{SELECT } x.A \\
\text{FROM } t \text{ AS } x \\
\text{WHERE } x.B > 0 \text{ AND } x.C = x.D \text{ AND NOT } x.D = x.E
\]

- \(x\) bound to t-tuples one after the other
- Attributes are functions “extracting“ components from current value of \(x\).
- Function application expressed in postfix notation

\[p(X) \leftarrow t(X, Y, Z, V, W), Y > 0, Z = V, \text{ not } V = W.\]

DRC: Domain Relational Calculus

- Variables represent components of the same tuple in \(t\)
- “same tuple in \(t\)“: expressed by common t-literal
CREATE VIEW p AS

SELECT A
FROM r, s
WHERE r.B=s.A

SELECT A
FROM s AS s1, s AS s2
WHERE s1.B=s.A

New tables:
r(A,B,C)
s(A,B)

p(X) ← r(X, Y, Z), s(V,W), Y=V.
p(X) ← r(X, Y, _), s(Y, _).
p(X) ← s(X, Y), s(Y, _).

Same translation idea:

Variant with double occurrence of t and aliasing:

CREATE VIEW p AS

SELECT A
FROM s AS s1, s AS s2
WHERE s1.B=s.A

Compactification by multiple variable occurrence and anonymous variables:
JOIN queries are dispensable in SQL, they are just syntactic alternatives for products queries with conditions in the WHERE part:

Datalog doesn't offer any such special „luxury“ notations: Same translation!
SQL2Datalog (3): Complex UNION-Query

CREATE VIEW p AS

SELECT A
FROM r
WHERE B=C

UNION

SELECT A
FROM s
WHERE B>0

p(X) ← r(X, Y, Y).
p(X) ← s(X, Z, Z>0).

Variant: Y instead of Z in 2nd rule
(Ys are different in different rules!)

p(X) ← r(X, Y, Y).
p(X) ← s(X, Y, Y>0).
CREATE VIEW p AS

SELECT A
FROM t
WHERE B>0 OR C=D

CREATE VIEW p AS

SELECT A
FROM t
WHERE C=D
UNION
SELECT A
FROM t
WHERE B>0

p(X) ← t(X, _, Z, Z, _).
p(X) ← t(X, Y, _, _, _), Y>0.

OR in WHERE parts is dispensable, UNION serves the same purpose!
CREATE VIEW p AS

\[
\text{SELECT } A \\
\text{FROM } r \\
\text{WHERE } B=C
\]

UNION

\[
\text{SELECT } A \\
\text{FROM } s \\
\text{WHERE } B>0
\]

Both tables in FROM part generate product of tables!

UNION is more general than OR as it applies to inhomogeneous input as well (two different tables in sub-queries):

In such a case: No transformation using OR in a single query is possible!

\[
p(X) \leftarrow t(X, \_ , Z, Z, \_). \\
p(X) \leftarrow t(X, Y, \_ , \_ , \_), Y>0.
\]
CREATE VIEW \( p \) AS

\[
\begin{align*}
\text{SELECT} & \quad A \\
\text{FROM} & \quad t \\
\text{WHERE} & \quad B \succ 0 \text{ AND } (C \equiv D \text{ OR } D \equiv E)
\end{align*}
\]

1\textsuperscript{st} step: Un-nesting using laws of Boolean algebra til OR is outermost

\[
\begin{align*}
\text{SELECT} & \quad A \\
\text{FROM} & \quad t \\
\text{WHERE} & \quad (B \succ 0 \text{ AND } C \equiv D) \text{ OR } (B \succ 0 \text{ AND } D \equiv E)
\end{align*}
\]

\[
\begin{align*}
p(X) & \leftarrow t(X, Y, Z, Z, _). \quad Y \succ 0. \\
p(X) & \leftarrow t(X, Y, _, Z, Z), \quad Y \succ 0.
\end{align*}
\]

2\textsuperscript{nd} step: Expressing OR by UNION (as before)

\[
\begin{align*}
\text{SELECT} & \quad A \\
\text{FROM} & \quad t \\
\text{WHERE} & \quad B \succ 0 \text{ AND } C \equiv D
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad A \\
\text{FROM} & \quad t \\
\text{WHERE} & \quad B \succ 0 \text{ AND } D \equiv E
\end{align*}
\]

\[
\begin{align*}
\text{UNION}
\end{align*}
\]
SQL2Datalog (5): MINUS/ NOT EXISTS

CREATE VIEW p AS

SELECT A
FROM r
WHERE B=C

SELECT A
FROM s
WHERE B>0
MINUS
SELECT A
FROM r
WHERE B=C AND NOT EXISTS

SELECT *
FROM s
WHERE B>0 AND
s.A=r.A

p(X) ← r(X, Y, Y), not s′(X)
s′(X) ← s(X, Z), Z>0.

MINUS is dispensable, too, as NOT EXISTS serves the same purpose (and shows clearer which subquery plays „generator role“ and which one „filter role“)

Auxiliary rule needed for embedded subquery in order to guarantee safe negation.
CREATE VIEW $p$ AS

\[
\text{SELECT } A \\
\text{FROM } r \\
\text{WHERE } B = C
\]

INTERSECT

\[
\text{SELECT } A \\
\text{FROM } s \\
\text{WHERE } B > 0
\]

\[
\text{SELECT } A \\
\text{FROM } r \\
\text{WHERE } B = C \text{ AND EXISTS (}
\text{SELECT } * \\
\text{FROM } s \\
\text{WHERE } B > 0 \text{ AND } s.A = r.A)
\]

$p(X) \leftarrow r(X, Y, Y), s'(X)$

$s'(X) \leftarrow s(X, Z), Z > 0.$
As opposed to NOT EXISTS: EXISTS queries can be „folded back“ into a single-level query!

Corresponding effect in Datalog: No auxiliary rule necessary as **not** is missing!
4.3 Incremental Evaluation of Aggregates
Aggregate Functions

- Aggregate functions compute a single result value from a set of input values
  - Often tuples are grouped in advance by some attributes using GROUP BY.
  - Function is then applied to each group, e.g., SUM, COUNT, MAX, AVG, ...
  - For each group exactly one result tuple is computed.

- Re-grouping and re-computation on large amount of data is very expensive.

- Most groups are unaffected by base table modifications though.

⇒ Aggregates views are ideal candidates for being incrementally updated!

- However, updating aggregated views incrementally is not always possible.
Incremental Evaluation of SUM

### Example

<table>
<thead>
<tr>
<th>Employee</th>
<th>Month</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Ann</td>
<td>Jan</td>
<td>500</td>
</tr>
<tr>
<td>Ann</td>
<td>Feb</td>
<td>300</td>
</tr>
<tr>
<td>Jack</td>
<td>Jan</td>
<td>4000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### SQL Query

```
SELECT Employee, SUM(Salary) AS SuSa
FROM tblE
GROUP BY Employee
```

### Results

<table>
<thead>
<tr>
<th>Employee</th>
<th>SuSa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>800</td>
</tr>
<tr>
<td>Jack</td>
<td>4000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Insertion Example

- **Insertion of (Jack, Feb, 2000)**
- **Insertion of (Dave, Jan, 123)**

### Update Example

- **Creation of new group**
  - Dave, Jan, 123
- **Update of existing group**
  - Jack, Feb, 6000
  - Jack, Jan, 10000

<table>
<thead>
<tr>
<th>Employee</th>
<th>SuSa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>800</td>
</tr>
<tr>
<td>Dave</td>
<td>123</td>
</tr>
<tr>
<td>Jack</td>
<td>10000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Incremental Evaluation of SUM

- The SUM-function is applied to each group of tuples with the same employees.

- For insertions we must differentiate between tuples that belong to an already existing group and those that form a new group.

- In the first case we have to perform an UPDATE on tblE

```sql
UPDATE tblE, SUM_E_p
SET tblE.SuSa = tblE.SuSa +
SUM_E_p.Salary
WHERE tblE.Employee=SUM_P_p.Employee
```

while in the second case an INSERT is necessary:

```sql
INSERT INTO tblE
(SELECT Employee, Salary
FROM E_p
GROUP BY Employee)
WHERE NOT EXISTS
(SELECT * FROM tblE WHERE tblE.Employee=SUM_P_p.Employee)
```

- Update should be performed before insert to avoid double updates.
Incremental Evaluation of SUM

<table>
<thead>
<tr>
<th>Employee</th>
<th>Month</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Ann</td>
<td>Jan</td>
<td>500</td>
</tr>
<tr>
<td>Ann</td>
<td>Feb</td>
<td>300</td>
</tr>
<tr>
<td>Jack</td>
<td>Jan</td>
<td>4000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**SELECT** Employee, SUM(Salary) AS SuSa
FROM tblE
GROUP BY Employee

<table>
<thead>
<tr>
<th>Employee</th>
<th>SuSa</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Ann</td>
<td>800</td>
</tr>
<tr>
<td>Jack</td>
<td>4000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Deletion of
(Ann,Feb,300)
(Jack,Jan,4000)

<table>
<thead>
<tr>
<th>Employee</th>
<th>Month</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Ann</td>
<td>Jan</td>
<td>500</td>
</tr>
<tr>
<td>Ann</td>
<td>Feb</td>
<td>300</td>
</tr>
<tr>
<td>Jack</td>
<td>Jan</td>
<td>4000</td>
</tr>
<tr>
<td>Dave</td>
<td>Jan</td>
<td>123</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**UPDATE** existing group
<table>
<thead>
<tr>
<th>Employee</th>
<th>SuSa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>500</td>
</tr>
<tr>
<td>Dave</td>
<td>123</td>
</tr>
<tr>
<td>Jack</td>
<td>4000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Deletion of an old group
<table>
<thead>
<tr>
<th>Employee</th>
<th>SuSa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>500</td>
</tr>
<tr>
<td>Jack</td>
<td>4000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
SUM-Aggregate and Deletions

- For deletions we must differentiate between empty and non-empty groups.
- In the first case we have to perform an UPDATE on tblE:

  ```sql
  UPDATE tblE, SUM_E_m
  SET tblE.SuSa = tblE.SuSa - SUM_E_m.Salary
  WHERE tblE.Employee=SUM_P_m.Employee
  ```

  while in the second case a DELETE is necessary:

  ```sql
  DELETE FROM tblE
  WHERE counter = ( SELECT count(*)
                     FROM tblE_m
                     WHERE tblE_m.g1=tblP.g1... ) = 0
  ```

- Counter is an auxiliary attribute calculated using the COUNT(*)-Function and represents the size of a group.
- This time deletions must be performed before updates to avoid redundant updates.
Incremental Evaluation of Aggregates

• In a similar way all „standard“-functions in SQL can be incrementally calculated (possibly using auxiliary attributes):

\[
\text{SUM}(P_{\text{new}}) = \text{SUM}(P) + \text{SUM}(P_p) - \text{SUM}(P_m)
\]

\[
\text{COUNT}(P_{\text{new}}) = \text{COUNT}(P) + \text{COUNT}(P_p) - \text{COUNT}(P_m)
\]

• **AVG** is more complicated and requires auxiliary attributes

\[
\text{AVG}(P_{\text{new}}) = \text{AVG}(P) + \frac{\text{SUM}(P_p) - \text{COUNT}(P_p) \cdot \text{AVG}(P)}{\text{COUNT}(P) + \text{COUNT}(P_p)}
\]

\[
\text{AVG}(P_{\text{new}}) = \frac{\text{SUM}(P) + \text{SUM}(P_p)}{\text{COUNT}(P) + \text{COUNT}(P_p)}
\]

• Standard deviation and variance in SQL will not be discussed here.
Incremental Evaluation of Aggregates

- Given insertions $P_{p}$ w.r.t. $P$ new MAX and MIN values can be easily computed by $\text{MAX}(P, P_{p})$ respectively $\text{MAX}(P, P_{p})$ where

$$\text{MAX}(P, P_{p}) = \begin{cases} \text{MAX}(P) & \text{if } \text{MAX}(P) > \text{MAX}(P_{p}) \\ \text{MAX}(P_{p}) & \text{otherwise} \end{cases}$$

- Given deletions $P_{m}$ with respect to relation $P$, new MAX and MIN values can be computed incrementally using an auxiliary table:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Ann</td>
<td>500</td>
</tr>
<tr>
<td>$P_{m}$</td>
<td>300</td>
</tr>
<tr>
<td>Ben</td>
<td>4000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
<th>Succ_Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Ann</td>
<td>500</td>
<td>300</td>
</tr>
<tr>
<td>Joe</td>
<td>300</td>
<td>212</td>
</tr>
<tr>
<td>Ben</td>
<td>4000</td>
<td>500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

next maximal value (analogously for MIN)
Sliding Windows

- In event monitoring systems we often deal with sliding windows:
  - Deletions and insertions go hand in hand
  - Since all tuples are usually timestamped they automatically represent true updates.

<table>
<thead>
<tr>
<th></th>
<th>Window $i$</th>
<th></th>
<th>Window $i+1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec</td>
<td>232</td>
<td>Dec</td>
<td>232</td>
</tr>
<tr>
<td>Jan</td>
<td>1000</td>
<td>Jan</td>
<td>1000</td>
</tr>
<tr>
<td>Feb</td>
<td>3000</td>
<td>Feb</td>
<td>3000</td>
</tr>
<tr>
<td>Mar</td>
<td>1424</td>
<td>Mar</td>
<td>1424</td>
</tr>
<tr>
<td>Apr</td>
<td>4711</td>
<td>Apr</td>
<td>4711</td>
</tr>
<tr>
<td>May</td>
<td>10815</td>
<td>May</td>
<td>10815</td>
</tr>
<tr>
<td>Jun</td>
<td>1234</td>
<td>Jun</td>
<td>1234</td>
</tr>
<tr>
<td>Jul</td>
<td>200</td>
<td>Jul</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

deletions $P_m$

insertions $P_p$
5. Summary / Remarks for the Examination
Examination Special - Topics

- Language Datalog: dependency graph, rule categories (hierarchical, recursive, (un-)stratifiable), iterated fixpoint computations
- Datalog semantics for unstratifiable rules: Herbrand base/universe, Doubled program approach + fixpoint computation
- Magic Sets: Transformation + properties
- Application of RACS/CSE for breaking introduced recursive cycles
- Determination of propagation rules (true, safe, potential)
- Magic Updates (specializing state relations)
- Specialized SQL: Propagation rules in Datalog ⇒ SQL and SQL ⇒ Datalog