TEMPORAL DATA ROADMAP

Paul A. Jungwirth

PGConf.dev 2026

21 May 2026

SQL:2011

- ☒ PRIMARY KEY + UNIQUE WITHOUT OVERLAPS
- ☒ FOREIGN KEY PERIOD
- ☒ UPDATE/DELETE FOR PORTION OF
- ☐ CASCADE/SET NULL/SET DEFAULT
- ☐ PERIOD FOR valid_at (valid_from, valid_til)
- ☐ SYSTEM TIME

BEYOND

- ☐ Arbitrary Types
- ☐ FDWs
- ☐ Relational Ops: semijoin, antijoin, outer join, setops, aggregates
- ☐ `MERGE`
- ☐ Optimizations
- ☐ `mdrange`
- ☐ Temporal DDL

ARBITRARY TYPES

PKS AND UKS

```
/* Discover the opclass's overlap operator: */
GetOperatorFromCompareType(
  opclass, InvalidOid, COMPARE_OVERLAP,
  &opid, &strat);
```

ARBITRARY TYPES

PKS AND UKS

ARBITRARY TYPES

PKS AND UKS

```
CONSTRAINT legacy_pk EXCLUDE
  (id WITH =, valid_at WITH &&)

CONSTRAINT shiny_pk PRIMARY KEY
  (id, valid_at WITHOUT OVERLAPS)
```

ARBITRARY TYPES

PKS AND UKS

```
CONSTRAINT legacy_pk EXCLUDE
  (id WITH =, valid_at WITH &&)

CONSTRAINT shiny_pk PRIMARY KEY
  (id, valid_at WITHOUT OVERLAPS)
```

```
(5, 'empty')
(5, 'empty')
```

ARBITRARY TYPES

PKS AND UKS

```
switch (typtype)
{
  case TYPTYPE_RANGE:
    r = DatumGetRangeTypeP(attval);
    isempty = RangeIsEmpty(r);
    break;
  case TYPTYPE_MULTIRANGE:
    mr = DatumGetMultirangeTypeP(attval);
    isempty = MultirangeIsEmpty(mr);
    break;
  default:
    elog(ERROR, "WITHOUT OVERLAPS column \"%s\" is not a range
       NameStr(attname));
}
```

ARBITRARY TYPES

FOREIGN KEYS

```
 1 typedef struct RI_ConstraintInfo
 2 {
 3   ...
 4
 5   /* anyrange <@ anyrange */
 6   Oid   period_contained_by_oper;
 7
 8   /* fkattr <@ range_agg(pkattr) */
 9   Oid   agged_period_contained_by_oper;
10
11    /* anyrange * anyrange */
12   Oid   period_intersect_oper;
13 }
```

ARBITRARY TYPES

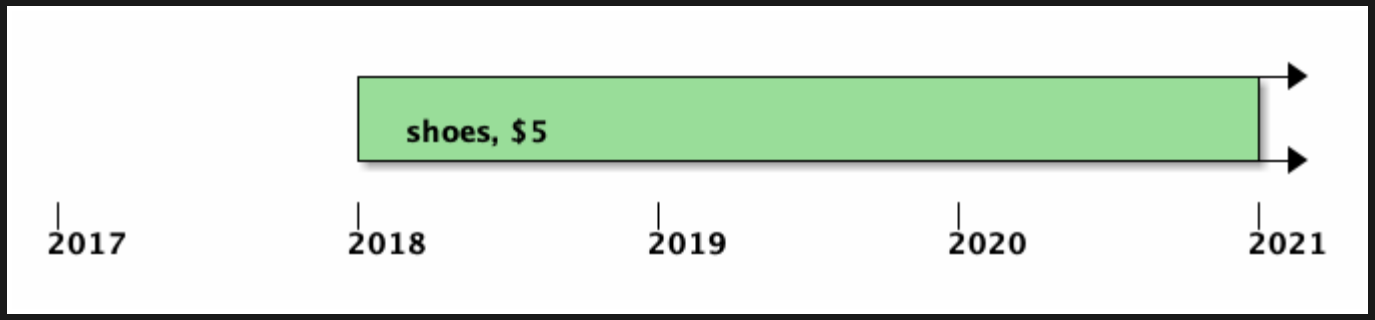
FOREIGN KEYS

```
 1 typedef struct RI_ConstraintInfo
 2 {
 3   ...
 4
 5   /* anyrange <@ anyrange */
 6   Oid   period_contained_by_oper;
 7
 8   /* fkattr <@ range_agg(pkattr) */
 9   Oid   agged_period_contained_by_oper;
10
11    /* anyrange * anyrange */
12   Oid   period_intersect_oper;
13 }
```

ARIBTRARY TYPES

FOR PORTION OF

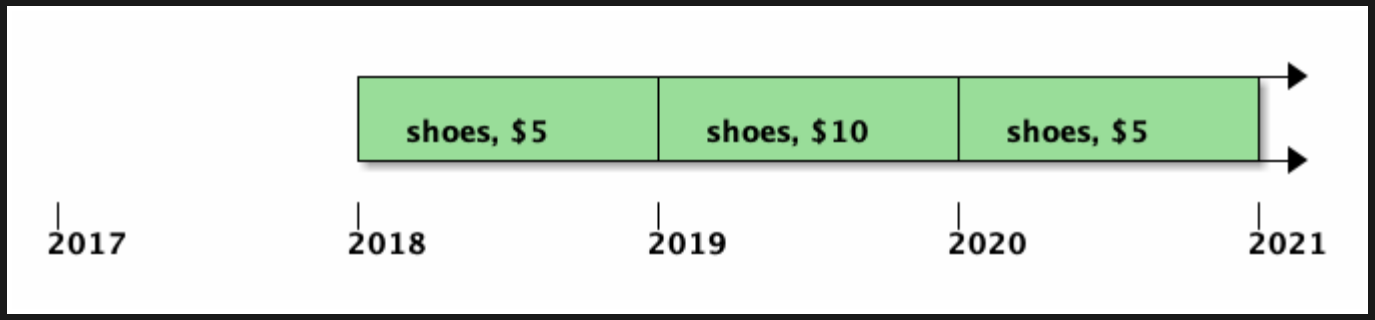
```
UPDATE t FOR PORTION OF valid at
   FROM '2019' TO '2020'
   SET price = 10;
```



ARIBTRARY TYPES

FOR PORTION OF

```
UPDATE t FOR PORTION OF valid at
   FROM '2019' TO '2020'
   SET price = 10;
```



ARIBTRARY TYPES

FOR PORTION OF

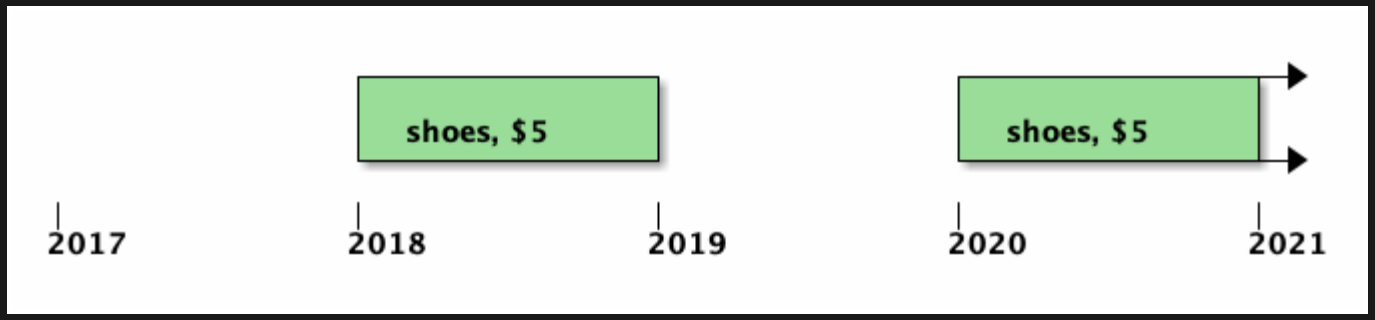
```
UPDATE t FOR PORTION OF valid at
   FROM '2019' TO '2020'
   SET price = 10;
```


```
new = old * target
```

ARIBTRARY TYPES

FOR PORTION OF

```
UPDATE t FOR PORTION OF valid at
   FROM '2019' TO '2020'
   SET price = 10;
```




```
leftovers = range_minus_multi(old, targ
```

ARIBTRARY TYPES

FOR PORTION OF

```
if (get_opclass_opfamily_and_input_type(opclass, &opfamily, &o
  switch (opcintype)
  {
    case ANYRANGEOID:
      result->withoutPortionProc = F_RANGE_MINUS_MULTI;
      break;
    case ANYMULTIRANGEOID:
      result->withoutPortionProc = F_MULTIRANGE_MINUS_MULTI;
      break;
    default:
      elog(ERROR, "unexpected opcintype: %u", opcintype);
  }
else
  elog(ERROR, "unexpected opclass: %u", opclass);
```

FDWs

```
/* We don't support FOR PORTION OF FDW queries. */
if (targetrel->rd_rel->relkind == RELKIND_FOREIGN_TABLE)
  ereport(ERROR,
    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
     errmsg("foreign tables don't support FOR PORTION OF")));
```

JOINS

```
1 SELECT  *,
2         -- intersection:
3         a.valid_at * b.valid_at AS valid_at
4 FROM    a
5 JOIN    b
6 ON      a.id = b.id
7         -- overlaps:
8 AND     a.valid_at && b.valid_at;
```

JOINS

```sql
SELECT  *,
        -- intersection:
        a.valid_at * b.valid_at AS valid_at
FROM    a
JOIN    b
ON      a.id = b.id
        -- overlaps:
AND     a.valid_at && b.valid_at;
```

SEMIJOIN

```
1 SELECT  *,
2         a.valid_at * b.valid_at AS valid_at
3 FROM    a
4 WHERE   EXISTS (
5           SELECT  1
6           FROM    b
7           WHERE   a.id = b.id
8           AND     a.valid_at && b.valid_at
9         );
```

SEMIJOIN

```sql
SELECT  *,
        a.valid_at * b.valid_at AS valid_at
FROM    a
WHERE   EXISTS (
          SELECT  1
          FROM    b
          WHERE   a.id = b.id
          AND     a.valid_at && b.valid_at
        );
```

SEMIJOIN

```sql
SELECT  *,
        a.valid_at * b.valid_at AS valid_at
FROM    a
WHERE   EXISTS (
          SELECT  1
          FROM    b
          WHERE   a.id = b.id
          AND     a.valid_at && b.valid_at
        );
```

SEMIJOIN

```
1 SELECT  *,
2         a.valid_at * b.valid_at AS valid_at
3 FROM    a
4 WHERE   EXISTS (
5           SELECT  1
6           FROM    b
7           WHERE   a.id = b.id
8           AND     a.valid_at && b.valid_at
9         );
```

SEMIJOIN

```
1 SELECT  a.id,
2         UNNEST(multirange(a.valid_at) * j.valid_at) AS valic
3 FROM    a
4 JOIN (
5   SELECT  b.id, range_agg(b.valid_at) AS valid_at
6   FROM    b
7   GROUP BY b.id
8 ) AS j
9 ON a.id = j.id AND a.valid_at && j.valid_at;
```

SEMIJOIN

```sql
SELECT  a.id,
        UNNEST(multirange(a.valid_at) * j.valid_at) AS valid
FROM    a
JOIN (
  SELECT  b.id, range_agg(b.valid_at) AS valid_at
  FROM    b
  GROUP BY b.id
) AS j
ON a.id = j.id AND a.valid_at && j.valid_at;
```

SEMIJOIN

```sql
SELECT  a.id,
        UNNEST(multirange(a.valid_at) * j.valid_at) AS valid
FROM    a
JOIN (
  SELECT  b.id, range_agg(b.valid_at) AS valid_at
  FROM    b
  GROUP BY b.id
) AS j
ON a.id = j.id AND a.valid_at && j.valid_at;
```

ANTIJOIN

```
 1 SELECT  a.id,
 2         UNNEST(CASE WHEN j.valid_at IS NULL THEN multirange
 3                     ELSE multirange(a.valid_at) - j.valid_a
 4 FROM    a
 5 LEFT JOIN (
 6   SELECT  b.id, range_agg(b.valid_at) AS valid_at
 7   FROM    b
 8   GROUP BY b.id
 9 ) AS j
10 ON a.id = j.id AND a.valid_at && j.valid_at
11 WHERE   NOT isempty(a.valid_at);
```

ANTIJOIN

```
 1 SELECT  a.id,
 2         UNNEST(CASE WHEN j.valid_at IS NULL THEN multirange
 3                     ELSE multirange(a.valid_at) - j.valid_a
 4 FROM    a
 5 LEFT JOIN (
 6   SELECT  b.id, range_agg(b.valid_at) AS valid_at
 7   FROM    b
 8   GROUP BY b.id
 9 ) AS j
10 ON a.id = j.id AND a.valid_at && j.valid_at
11 WHERE   NOT isempty(a.valid_at);
```

ANTIJOIN

```
 1 SELECT  a.id,
 2         UNNEST(CASE WHEN j.valid_at IS NULL THEN multirange
 3                     ELSE multirange(a.valid_at) - j.valid_a
 4 FROM    a
 5 LEFT JOIN (
 6   SELECT  b.id, range_agg(b.valid_at) AS valid_at
 7   FROM    b
 8   GROUP BY b.id
 9 ) AS j
10 ON a.id = j.id AND a.valid_at && j.valid_at
11 WHERE   NOT isempty(a.valid_at);
```

OUTER JOIN

```
 1 SELECT  a.*, b.*,
 2         UNNEST(multirange(a.valid_at) * multirange(b.valid_
 3 FROM    a
 4 JOIN    b
 5 ON      a.id = b.id AND a.valid_at && b.valid_at
 6 UNION ALL
 7 SELECT  a.*, (NULL::b).*,
 8         UNNEST(
 9           CASE WHEN j.valid_at IS NULL
10                THEN multirange(a.valid_at)
11                ELSE multirange(a.valid_at) - j.valid_at END
12         )
13 FROM    a
14 LEFT JOIN (
15   SELECT  b.id, range_agg(b.valid_at) AS valid_at
```

OUTER JOIN

```
 1 SELECT  a.*, b.*,
 2         UNNEST(multirange(a.valid_at) * multirange(b.valid_
 3 FROM    a
 4 JOIN    b
 5 ON      a.id = b.id AND a.valid_at && b.valid_at
 6 UNION ALL
 7 SELECT  a.*, (NULL::b).*,
 8         UNNEST(
 9           CASE WHEN j.valid_at IS NULL
10                THEN multirange(a.valid_at)
11                ELSE multirange(a.valid_at) - j.valid_at END
12         )
13 FROM    a
14 LEFT JOIN (
15   SELECT  b.id, range_agg(b.valid_at) AS valid_at
```

OUTER JOIN

```
 7 SELECT  a.*, (NULL::b).*,
 8         UNNEST(
 9           CASE WHEN j.valid_at IS NULL
10                THEN multirange(a.valid_at)
11                ELSE multirange(a.valid_at) - j.valid_at END
12         )
13 FROM    a
14 LEFT JOIN (
15   SELECT  b.id, range_agg(b.valid_at) AS valid_at
16   FROM    b
17   GROUP BY b.id
18 ) AS j
19 ON      a.id = j.id AND a.valid_at && j.valid_at
20 ORDER BY 1, 5
21 ;
```

ENCAPSULATING

```
1 SELECT  a, b
2 FROM    temporal_semijoin(
3           'a', 'id', 'valid_at',
4           'b', 'a_id', 'valid_at'
5         ) AS j(a a, b b)
6 WHERE   a.id = 5;
```

ENCAPSULATING

```
1 SELECT  a, b
2 FROM    temporal_semijoin(
3           'a', 'id', 'valid_at',
4           'b', 'a_id', 'valid_at'
5         ) AS j(a a, b b)
6 WHERE   a.id = 5;
```

ENCAPSULATING

```
1 SELECT  a, b
2 FROM    temporal_semijoin(
3           'a', 'id', 'valid_at',
4           'b', 'a_id', 'valid_at'
5         ) AS j(a a, b b)
6 WHERE   a.id = 5;
```

INLINING

```
typedef struct SupportRequestInlineInFrom
{
  NodeTag     type;

  /* Planner's infrastructure */
  PlannerInfo *root;
  /* Function call to be simplified */
  RangeTblFunction *rtfunc;
  /* Function definition from pg_proc */
  HeapTuple   proc;
} SupportRequestInlineInFrom;
```

temporal_semijoin_sql

```
appendStringInfo(&q,
  "SELECT %2$s, UNNEST(multirange(%2$s.%3$s) * %4$s.%5$s) AS %
  "FROM %1$s\n"
  "JOIN (\n"
  "  SELECT ",
  left_nsp_rel_q, left_rel_q, left_valid_col_q,
  subquery_alias, right_valid_col_q, result_valid_col_q);
```

ENCAPSULATING

```
SELECT  a, b
FROM    temporal_semijoin(
          'a', 'id', 'valid_at',
          'b', 'a_id', 'valid_at'
        ) AS j(a a, b b)
WHERE   a.id = 5;
```

ENCAPSULATING

```
SELECT  a, b
FROM    a
JOIN    b
ON      temporal_semijoin(a, b, a.id, a.valid_at, b.a_id, b.va
WHERE   a.id = 5;
```

CUSTOM SCAN

```
typedef struct CustomScan
{
    Scan        scan;
    uint32      flags;
    List       *custom_plans;
    List       *custom_exprs;
    List       *custom_private;
    List       *custom_scan_tlist;
    Bitmapset  *custom_relids;
    const struct CustomScanMethods *methods;
} CustomScan;
```

IN CORE?

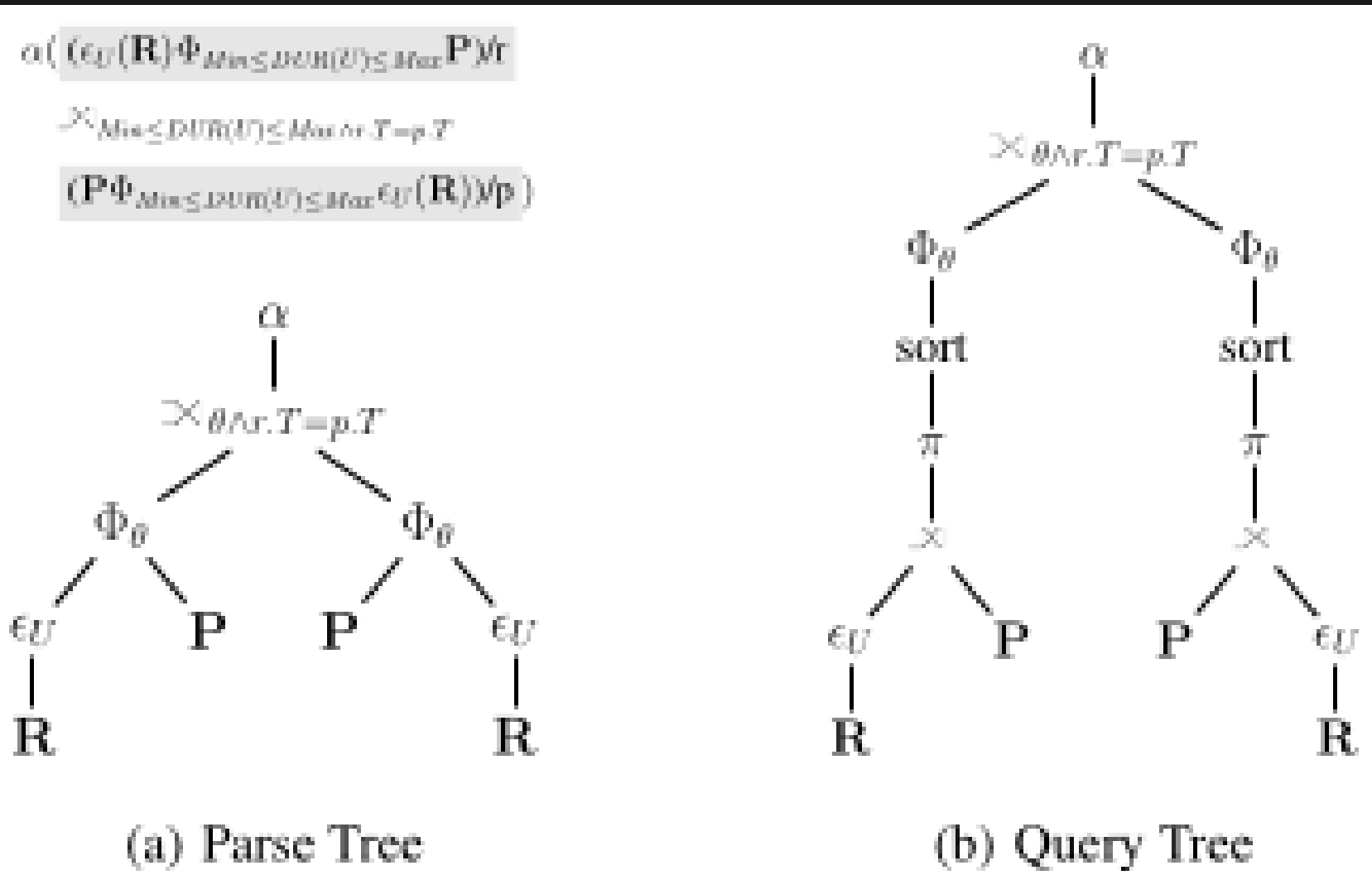


Figure 12: Parse Tree and Query Tree.

SYNTAX: valid_at?

```
1 SELECT  a.*, combined_valid_at
2 FROM    a
3 LEFT JOIN b AS b (PERIOD combined_valid_at)
4 USING (id, PERIOD valid_at)
```

SYNTAX: valid_at?

```
1 SELECT  a.*, combined_valid_at
2 FROM    a
3 LEFT JOIN b AS b (PERIOD combined_valid_at)
4 USING (id, PERIOD valid_at)
```

SYNTAX: valid_at?

```
1 SELECT  a.*, combined_valid_at
2 FROM    a
3 LEFT JOIN b AS b (PERIOD combined_valid_at)
4 USING (id, PERIOD valid_at)
```

SYNTAX: valid_at?

```
1 SELECT  a.*, combined_valid_at
2 FROM    a
3 LEFT JOIN b AS b (PERIOD combined_valid_at)
4 USING (id, PERIOD valid_at)
```

RELATIONAL ALGEBRA

> *To your question on equivalence rules: Most traditional rules hold also for temporal operators, but there is a crucial difference. This difference regards whenever we access timestamps, so the* ***distributivity of selection****. Whenever we have a temporal join, we have two input tables with each one timestamp, but the output has only one which will be the intersection of these two timestamps. This means we modified/removed them and thus we cannot distribute selection over these timestamps, but we can over all other columns.*

—Anton Dignös, 2025-11-26

RELATIONAL ALGEBRA

$$Q \times (R - S) \neq (Q \times R) - (Q \times S)$$

```
LHS — (temporal-cartesian-product Q (temporal-except R S)):
+------+----------+------+-----------+-----------+
| q-id | valid-at | r-id | valid-at  | valid-at  |
+------+----------+------+-----------+-----------+
| q1   | (0 . 20) | r1   | (0 . 5)   | (0 . 5)   |
| q1   | (0 . 20) | r1   | (10 . 20) | (10 . 20) |
+------+----------+------+-----------+-----------+
RHS — (temporal-except (temporal-cartesian-product Q R)
                       (temporal-cartesian-product Q S)):
+------+----------+------+----------+----------+
| q-id | valid-at | r-id | valid-at | valid-at |
+------+----------+------+----------+----------+
| q1   | (0 . 20) | r1   | (0 . 20) | (0 . 20) |
+------+----------+------+----------+----------+
```

SETOPS

```sql
-- temporal UNION:
SELECT  id, UNNEST(range_agg(valid_at)) AS valid_at
FROM  (
  SELECT * FROM a
  UNION
  SELECT * FROM b
) x
GROUP BY id
ORDER BY id, valid_at;
```

AGGREGATES

processed together are identical.

3) *Scaling of attribute values.* We use the values of attributes B, U, and T to scale the budget values. Note that each tuple in x_2 includes the information necessary for scaling. For instance, the first tuple in x_2 records that the 5K budget for 5 months is to be scaled to a 3 months budget.

4) *Evaluation of the corresponding nontemporal operator.* Instead of a temporal aggregation over the original argument relation, we perform a nontemporal aggregation over the relation with adjusted intervals and scaled budgets. The adjusted interval attribute T is used for grouping like other attributes.

P_2	CS	6K	[2014-4, 2014-7)	[2014-6, 2014-7)
P_3	MA	2K	[2014-1, 2014-3)	[2014-1, 2014-3)

3) ↓

$x_3 = \pi_{P,D,scaleU(B,T,U)/B,T}(x_2)$

P	D	B	T
P_1	CS	3K	[2014-1, 2014-4)
P_1	CS	2K	[2014-4, 2014-6)
P_2	CS	4K	[2014-4, 2014-6)
P_2	CS	2K	[2014-6, 2014-7)
P_3	MA	2K	[2014-1, 2014-3)

4) ↓

$z = {}_{T,D}\vartheta_{SUM(B)/B}(x_3)$

	D	B	T
z_1	CS	3K	[2014-1, 2014-4)
z_2	CS	6K	[2014-4, 2014-6)
z_3	CS	2K	[2014-6, 2014-7)
z_4	MA	2K	[2014-1, 2014-3)

MERGE

```
MERGE INTO products
  FOR PORTION OF valid_at FROM t1 TO t2
  WHEN NOT MATCHED THEN INSERT ...
  WHEN MATCHED THEN UPDATE SET ...
;
```

OPTIMIZATIONS

- Use btree
  - Skip scan?
- Improve gist
  - Binary search
- Range Merge Join
- Recognize snapshot queries
  - `n_distinct`
  - `GROUP BY`
- Self-Join Elimination
- More Algebra

GIST BINARY SEARCH

To avoid having to check all entries in a node sequentially when traversing it, either for search, insert or delete operations, a GiST implementation should provide an interface to specialize the intra-node layout of entries. This way, the nodes can store and compress predicates in any way that is efficient for the particular key domain and set of supported queries. For example, a GiST implementation of a B-tree would use an ordered sequence in order to be able to do binary search.

RANGE MERGE JOIN

- 2012: Jeff Davis, "9.3 Pre-Proposal"
- 2017: Jeff Davis, "Range Merge Join v1"
- 2020: Thomas Mannhart, BSc Thesis
- 2021: Thomas Mannhart, "Patch: Range Merge Join"
- 2021: Anton Dignös et al, ""Leveraging range joins for the computation of overlap joins"

RANGE MERGE JOIN

The problem is this: the algorithm for a single key demands that the input is sorted by (lower(r) NULLS FIRST, upper(r) NULLS LAST). That's easy, because that's also the sort order for the range operator class, so everything just works.

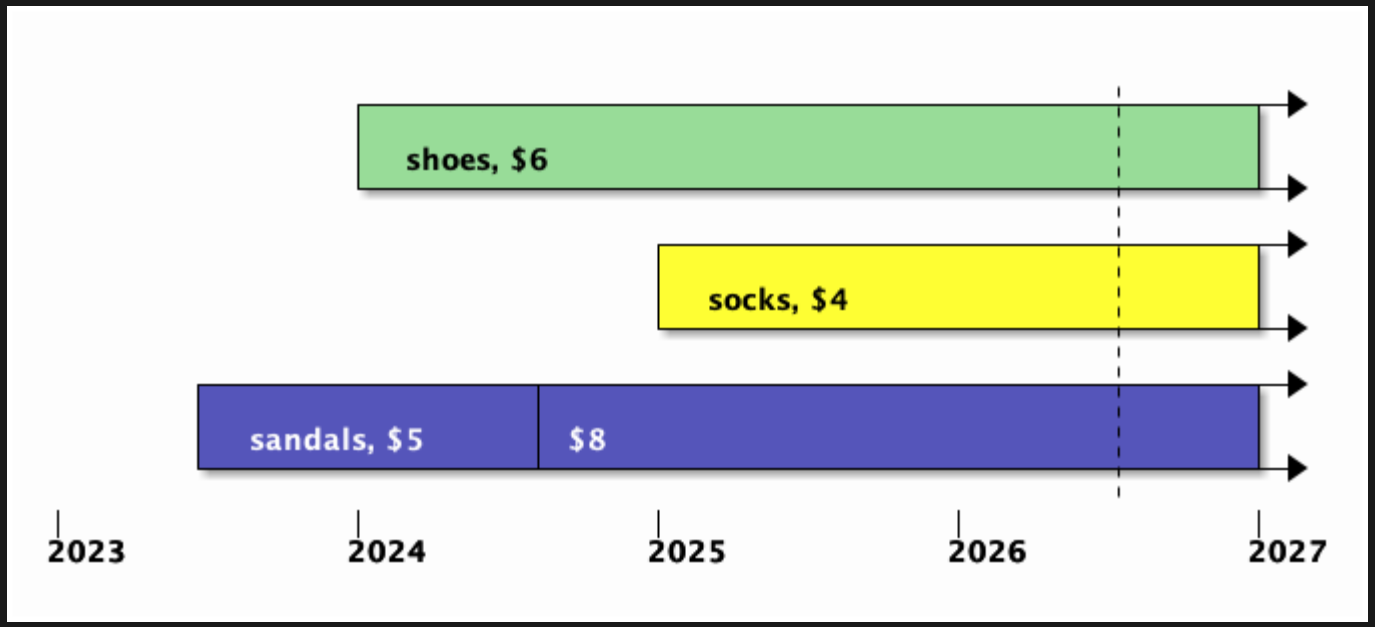
For multiple keys, the order of the input is:

```
lower(r1) NULLS FIRST, lower(r2) NULLS FIRST,
upper(r1) NULLS LAST, upper(r2) NULLS LAST
```

But that can't match up with any opclass, because an opclass can only order one attribute at a time. In this case, the lower bound of r2 is more significant than the upper bound of r1.

—Jeff Davis, 2017-09-17

SNAPSHOT QUERIES



SNAPSHOT QUERIES

```
 1 static struct StatsArgInfo attarginfo[] =
 2 {
 3    [ATTRELSCHEMA_ARG] = {"schemaname", TEXTOID},
 4    [ATTRELNAME_ARG] = {"relname", TEXTOID},
 5    [ATTNAME_ARG] = {"attname", TEXTOID},
 6    [ATTNUM_ARG] = {"attnum", INT2OID},
 7    [INHERITED_ARG] = {"inherited", BOOLOID},
 8    [NULL_FRAC_ARG] = {"null_frac", FLOAT4OID},
 9    [AVG_WIDTH_ARG] = {"avg_width", INT4OID},
10    [N_DISTINCT_ARG] = {"n_distinct", FLOAT4OID},
11    [MOST_COMMON_VALS_ARG] = {"most_common_vals", TEXTOID},
12    [MOST_COMMON_FREQS_ARG] = {"most_common_freqs", FLOAT4ARR
13    [HISTOGRAM_BOUNDS_ARG] = {"histogram_bounds", TEXTOID},
14    [CORRELATION_ARG] = {"correlation", FLOAT4OID},
15    [MOST_COMMON_ELEMS_ARG] = {"most_common_elems", TEXTOID}
```

SNAPSHOT QUERIES

```sql
SELECT  p.id, p.name, COUNT(v.id)
FROM    products p
JOIN    variants v
  ON    v.product_id = p.id
  AND   v.valid_at @> NOW()
WHERE   p.valid_at @> NOW()
GROUP BY p.id
```

SELF-JOIN ELIMINATION

Hi Hackers,

relation_has_unique_index_for() checks whether join clause equality operators belong to the index's opfamily via mergeopfamilies. Since mergeopfamilies only lists btree opfamilies, this check always fails for GiST-backed unique indexes such as those created by PRIMARY KEY with WITHOUT OVERLAPS, preventing self-join elimination.

Fix by falling back to op_in_opfamily() when the mergeopfamilies check fails. The clause is already known to be a mergejoinable equality, so confirming the operator is registered in the index's opfamily is sufficient to prove that the index's uniqueness guarantee applies.

Attached a patch to fix this and added corresponding tests.

—Satyanarayana Narlapuram, 2026-04-21

MORE ALGEBRA

(a * b) @> t = (a @> t AND b @> t)

```
SELECT  c.valid_at
FROM    (
  SELECT  a.valid_at * b.valid_at AS valid_at
  FROM    a JOIN b
  ON      a.valid_at && b.valid_at) AS c
WHERE   c.valid_at @> current_date;
```

MDRANGE

```
CREATE TYPE mdrange AS (
  INPUT = mdrange_in,
  OUTPUT = mdrange_out,
  ...
);
```

MDRANGE

```
CREATE PSEUDOTYPE mdrange AS (
  ??? = ???,
  ??? = ???,
  ...
);
```

DDL CHANGES

SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution

John F. Roddick
School of Computer and Information Science,
University of South Australia,
The Levels, SA 5095, South Australia

Abstract

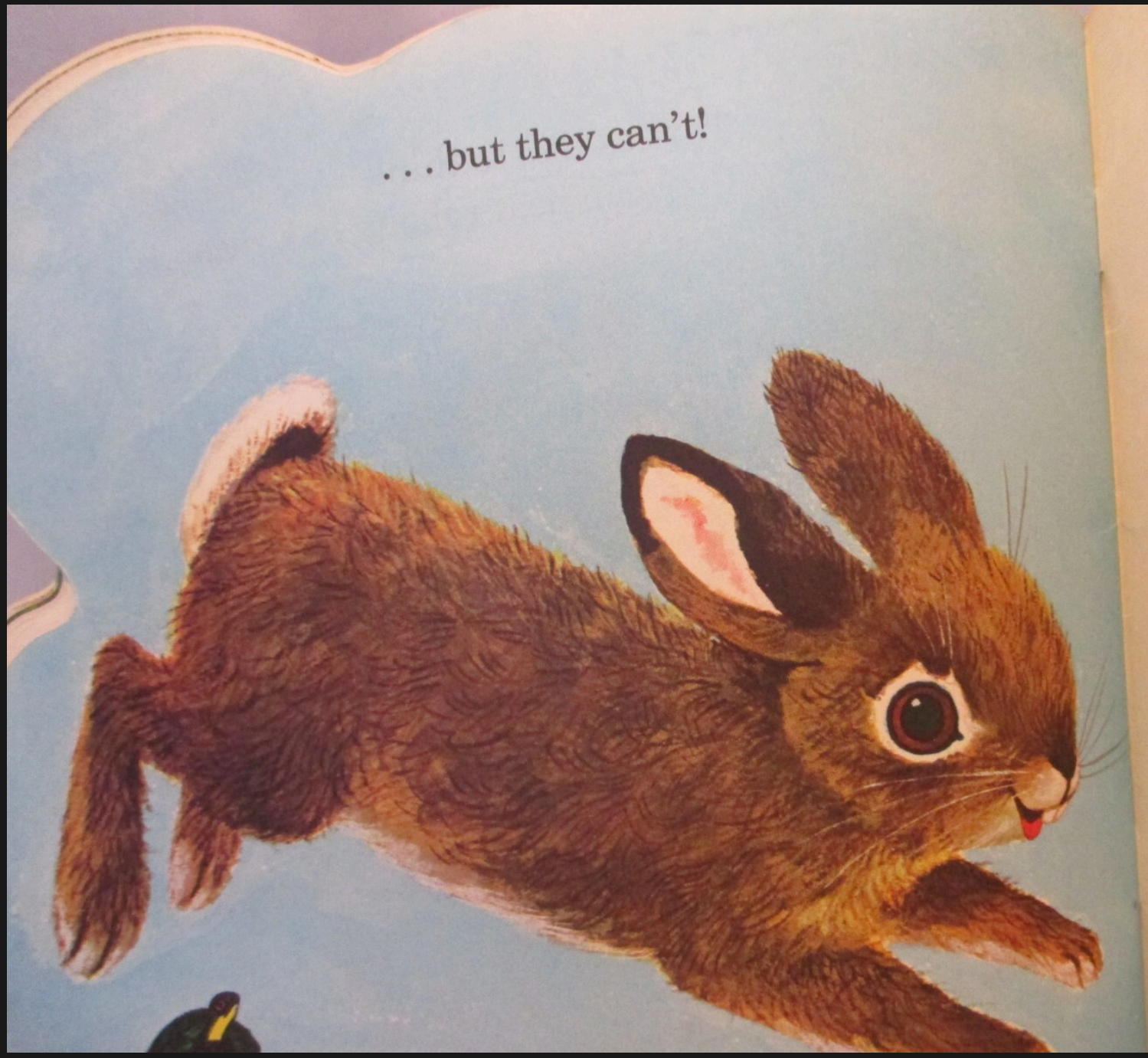
The incorporation of a knowledge of time within database systems allows for temporally related information to be modelled more naturally and consistently. Adding this support to the meta-database further enhances its semantic capability and allows elaborate interrogation of data. This paper presents SQL/SE, an SQL extension capable of handling schema evolution in relational database systems.

structure and/or more appropriate output. For instance, one of the facilities afforded by schema evolution support is the ability, through a (sufficiently powerful) query language, to reissue queries made at a previous point in time with the assurance of identical results. Such assurances are not possible within temporal databases unless a static schema is maintained.

The ANSI and ISO standards committees have proposed an extension to the SQL standard to provide for limited temporal support[17]. The

DDL CHANGES

. . . but they can't!



REFERENCES

- John F. Roddick. "SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution." SIGMOD RECORD, 1992. https://dl.acm.org/doi/epdf/10.1145/140979.140985
- Alexander Tuzhilin and James Clifford. "A Temporal Relational Algebra as a Basis for Temporal Relational Completeness." VLDB, 1990. https://www.vldb.org/conf/1990/P013.PDF
- Richard Snodgrass. "An Overview of TQuel." *Temporal Databases: Theory, Design, and Implementation*, 1993.
- Marcel Kornacker, C. Mohan, Joseph M. Hellerstein. "Concurrency and recovery in generalized search trees." SIGMOD, 1997. https://dl.acm.org/doi/10.1145/253260.253272
- Anton Dignös, Michael H. Böhlen, Johann Gamper. "Temporal Alignment." SIGMOD, 2012. https://files.ifi.uzh.ch/boehlen/Papers/modf174-dignoes.pdf
- Jeff Davis. "9.3 Pre-Proposal: Range Merge Join." pgsql-hackers, 2012. https://www.postgresql.org/message-id/1334554850.10878.52.camel%40jdavis
- Ali Piroozi. "Equivalence Rules" pgsql-hackers, 2014. https://www.postgresql.org/message-id/CAMi-Eo1Wrxft=0ZsvKkZRYkoFVGRnbhbjL6a=rUf58xOGuj7DA@mail.gmail.com
- https://github.com/postgres/postgres/blob/master/src/backend/optimizer/README
- Anton Dignös, Michael Hanspeter, Johann Gamper, Christian S. Jenson. "Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries." ACM Transactions on Database Systems, 2016. https://dl.acm.org/doi/10.1145/2967608
- Anton Dignös. "Temporal query processing with range types." pgsql-hackers, 2016. https://www.postgresql.org/message-id/CALNdv1h7TUP24Nro53KecvWB2kwA67p+PByDuP6_1GeESTFgSA@mail.gmail.com
- Jeff Davis. "Range Merge Join v1." pgsql-hackers, 2017. https://www.postgresql.org/message-id/CAMp0ubfwAFFW3O_NgKqpRPmm56M4weTEXjprb2gP_NrDaEC4Eg%40mail.gmail.com
- Thomas Mannhart, Anton Dignös. "A General-purpose Range Join Algorithm for PostgreSQL." BSc thesis, 2020. https://tpg.inf.unibz.it/downloads/rmj-report.pdf
- Danila Piatov, Sven Helmer, Anton Dignös, Fabio Persia. "Cache-efficient sweeping-based interval joins for extended Allen relation predicates." VLDB 2021. https://link.springer.com/content/pdf/10.1007/s00778-020-00650-5.pdf
- Thomas Mannhart. "Patch: Range Merge Join." pgsql-hackers, 2021. https://www.postgresql.org/message-id/CAMWfgiAsJgVrkbrsv740Y2%3D2duO4rVYRhaD08EhFqBuJFmBH1A%40mail.gmail.com
- Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, Peter Moser. "Leveraging range joins for the computation of overlap joins." VLDB, 2021. https://link.springer.com/content/pdf/10.1007/s00778-021-00692-3.pdf
- Paul Jungwirth. Temporal Ops Postgres Extension. https://github.com/pjungwir/temporal_ops
- Boris Novikov. "PostgreSQL Temporal Aggregates: SUM, AVG & COUNT Across Time." Red Gate, 2024. https://www.red-gate.com/simple-talk/databases/postgresql/making-temporal-databases-work-part-2-computing-aggregates-across-temporal-versions/
- Satyanarayana Narlapuram. "Allow SJE to recognize GiST-backed temporal primary keys." pgsql-hackers, 2026. https://www.postgresql.org/message-id/CAHg%2BQDeXwdOzrmb-sSATK4whbyhOgzyCGN%2BbY%3DYXU9qOzJaWSg%40mail.gmail.com
- Paul Jungwirth. Racket Library for Relational Operators. https://github.com/pjungwir/relsim
- Paul Jungwirth. These slides. https://github.com/pjungwir/temporal-roadmap

THANKS

https://github.com/pjungwir/temporal-roadmap