Temporal Data Management in PostgreSQL: Past, Present, and Future

by Jeff Davis
Aster Data / Teradata, Inc.

Original Problem

- Needed to implement a table “audit log”
  - Historical record of table
- But also needed to be efficiently queryable
  - See data as of a specific time
    - “snapshot”

Simple, right?

- The problem was simple.
- The solution was not.
- Queries were awkward and performed poorly.

Awkward Queries

- Get the “<”, “<=” signs right.
- Be careful of NULLs!
  - Often used to represent “infinity”, but they don't act that way with comparison ops.
- Representing single points of time or empty periods of time awkward

Poor Performance

```
EXPLAIN SELECT * FROM mytable
 WHERE ts_from      <= '2010-01-01' AND
       '2010-01-01' <  ts_to;

Bitmap Heap Scan on mytable
   Recheck Cond: ...
   Filter: ...
   ->  Bitmap Index Scan on mytable_from_idx
         Index Cond: …
```

(Or, perhaps a BitmapAnd if the planner guesses correctly; but still not a great plan.)

And it gets worse

- As the queries become more complex, the problems get worse.
- Planner knows even less, and is more likely to be wildly off in cardinality estimates

How to improve the situation

- Period data type (extension since 2007)
- Exclusion Constraints (9.0)
- Range Types (9.2)
- Range Keys / Range Foreign Keys (future)
- Range Merge Join (future)
- Simple Table historical log (future)
- Multi-Range (future)

PERIOD data type (extension since 2007)

- https://github.com/jeff-davis/PostgreSQL-Temporal
- Implements single data type with a definite beginning and a definite end, e.g. “[2010-01-01, 2010-02-01)”
- Bounds can be inclusive or exclusive
- Indexable using a spatial index that can search for “overlaps”, “contains” and other queries efficiently.
- Simplifies queries and makes them more efficient
- But it will be superseded by Range Types in 9.2!

Exclusion Constraints (9.0)

- Solves the “schedule conflict” problem
- Like UNIQUE constraints, but more flexible

Exclusion Constraints (9.0)

- UNIQUE (loosely) means: any row that ***is equal to*** this one conflicts, and both cannot exist simultaneously.
- Exclusion constraints allow you to use other conditions, like ***overlaps with***
- Therefore, you can prohibit schedule conflicts with a declarative constraint!

Exclusion Constraints (9.0)

```
-- example shown in 9.2 to take advantage
-- of Range Types and Extensions

CREATE EXTENSION btree_gist;
CREATE TABLE conf_room_reservation (
  room    TEXT,
  speaker TEXT,
  during  TSTZRANGE,
  EXCLUDE USING gist
    (room WITH =, during WITH &&),
  EXCLUDE USING gist
    (speaker WITH =, during WITH &&)
);
```

Exclusion Constraints (9.0)

```
INSERT INTO conf_room_reservation VALUES
  ('Room123', 'Speaker1',
   '[2010-01-01 14:30, 2010-01-01 15:30)');

-- succeeds

INSERT INTO conf_room_reservation VALUES
  ('Room123', 'Speaker2',
   '[2010-01-01 15:15, 2010-01-01 16:30)');

-- conflict!
ERROR:  conflicting key value violates
exclusion constraint ...
```

Exclusion Constraints (9.0)

- Simplest way to avoid schedule conflicts
- Performs the best
- Less error-prone than triggers
- Declarative

Exclusion Constraints (9.0)

- Avoid trying to improvise a solution with triggers, etc.
  - Many pitfalls!
- Exclusion constraints much better.

Range Types (9.2)

- Generalization of PERIOD data type extension
- Ranges of any ordered data type
- “TSTZRANGE” (range of TIMSTAMPTZ) supersedes PERIOD

Range Types (9.2)

- Offers many more data types:
  - TSTZRANGE
  - DATERANGE
  - TSRANGE
  - ...
  - Non-temporal (e.g. INT4RANGE, …)
- Ability to create more data types easily
  - CREATE TYPE … AS RANGE (...)

Range Types (9.2)

```
CREATE TABLE hotel_reservation AS (
…,
during DATERANGE,
…
);

CREATE TABLE conf_room_reservation AS (
…,
during TSTZRANGE,
…
);

-- and remember to specify exclusion
-- constraints, of course
```

Range Keys / Range Foreign Keys (future)

- Part of range types, just not done yet
- “Range Key” would be like declaring a column unique, but with range semantics
- Syntax sugar for an Exclusion Constraint where ranges use “overlaps” and non-ranges use ordinary equality

Range Keys / Range Foreign Keys (future)

- “Range Foreign Key” would be like a foreign key, but with range semantics
- ranges in referencing table must be “contained in” ranges in referenced table
- Referenced table must have a range key
- Can sort of be done with triggers now, but this would be easier and more complete

Range Merge Join (future)

- Joins on “overlaps” rather than “equals”
- Useful for matching up two events that partially overlap, or happen within some threshold of each other

Range Merge Join (future)

```
SELECT
  customer_id,
  bill(rate,
       range_intersect(u.during, r.during)
      ) AS bill
FROM billing_rate r, billing_usage u
WHERE r.during && u.during;
```

Range Merge Join (future)

- Right now, that can only be executed with nested loop join
- Make it faster!

Simple Historical Table Log (future)

- Simple DDL to create a “historical” version of the table
  - Keep old records with a special “during” column to hold the time range that the row existed
  - Trigger makes it automatic
- Automatically include current records (with end time infinity) when selecting from the historical table
  - Kind of like inheritance

Simple Historical Table Log (future)

```
ALTER TABLE mytable ADD HISTORY;

-- See version of mytable as of 2010-01-01
SELECT * FROM mytable_history
  WHERE during @> '2010-01-01';
```

Multi-Range (future)

- Extend range types to allow multiple disjoint ranges inside a single value
- Mathematical closure of ranges over range_union() and other functions
- In other words, range_union() wouldn't have to throw an error if it can't produce a single output range
  - Can hold the information necessary for further operations

Conclusion

- Many of the critical capabilities are available today
- Will perform well
- But complex cases are still problematic and I'm still working on solutions
- More hackers welcome!