Agenda

- What kind of temporal data do we need?
- What data types does PostgreSQL offer?
- Temporality Representations
  - Time Travel, Transaction Tables, Serial Numbers
What kind of temporal data do we need?

- Databases store facts about objects and events
- Interesting times include
  - When an event took place
  - When the event was recorded
  - When someone was charged for the event
More Interesting Times

- When you start recognizing income on the event
- When you end recognizing income on the event
- When an object state begins
- When an object state ends
PostgreSQL Data Types

- **Date**
  Problem: Pre-assumes evaluation of cutoff between days!

- **Time with/without timezone**
  Problem: Comparisons of Date+Time turn into hideous SQL

- **Timestamp**
  Combines Date + Time
PostgreSQL Data Types

- Timestamp with time zone
  Allows collecting time in ‘local times’ and recognizing that

- Interval
  Difference between two times/timestamps
  Very useful for indicating duration of time ranges
Operators

- `time/timestamp/date +|-= interval = time/timestamp/date`
- `timestamp - timestamp = interval`
  (likewise for the others)
- `timestamp <, <=, >, >= timestamp`
- `A BETWEEN B AND C`
  `A >= B and A <= C`
Variations on “when is it???”

- NOW(), transaction_timestamp, current_timestamp all providing start of transaction
- statement_timestamp
- clock_timestamp
- transaction commit timestamp - not available!
Commit Timestamp

- Useful representation: Tables record (serverID, ctid)
- At COMMIT time, if the transaction has used this, then insert (serverID, ctid, clock_timestamp) into timestamp table
- Eliminates Slony-I “SYNC” thread & simplifies queries
- Helpful for multimaster replication strategies
- Adds a table full of timestamps that needs cleansing :-(

PGTemporal

- PgFoundry project implementing (timestamp,timestamp) type + all logical operations
- First aspect: Supports inclusive & exclusive periods
  - [ From, To ], ( From, To ), [ From, To ), ( From, To ]
  - [ and ] indicate “inclusive” periods beginning and ending at the specified moment
  - ( and ) indicate exclusive periods excluding endpoints
Inclusion & Exclusion

- Commonly, [From, To) is the ideal representation
  - Today’s data easily characterized as [2009-05-22, 2009-05-23)
  - This month’s period: [2009-05-01, 2009-06-01)
- Note that SQL “BETWEEN” is equivalent to [From, To]
A Veritable Panoply of Operators

- length(p), first(p), last(p), prior(p), next(p)
- contains(p, t), contains(p1, p2), contained_by(t, p),
  contained_by(p1,p2), overlaps(p1,p2), adjacent(p1,p2),
  overleft(p1,p2), overright(p1,p2), is_empty(p), equals
  (p1,p2), nequals(p1,p2), before(p1,p2), after(p1,p2)
- period(t), period(t1,t2), empty_period()
- period_intersect(p1,p2), period_union(p1,p2), minus
  (p1,p2)
Core???

- Should PGTemporal be in core?
- What would be needed for it to head in?
Classical SQL Temporality

- Developing Time-Oriented Database Applications in SQL - Richard Snodgrass, available freely as PDF
- Uses periods much as in PGTemporal
- Standard SQL does not support periods, alas!
- Considerable attention to handling insertion of past/future history
Foreign Key Challenges

- Nontemporal tables: No temporality, No problem!
- Referencing table is temporal, referenced table isn’t: No problem!
- Referenced table is temporal Troublesome!
  - Referential integrity may be violated simply via passage of time
  - Referenced & referencing tables may vary independently!
PostgreSQL Time Travel

- Take a stateful table
- Add triggers to capture (From, To) timestamps on INSERT, UPDATE, DELETE
- Sadly, this breaks if you require referential integrity constraints pointing to this table :-(

Time Travel Actions

- On INSERT
  - \((\text{NEW.From}, \text{NEW.To}) = (\text{NOW}(), \text{NULL})\)

- On DELETE
  - \((\text{OLD.From}, \text{OLD.To}) = (\text{PrevValue}, \text{NOW}())\)

- On UPDATE
  - Transforms into DELETE old, INSERT new
Pulling Specific State

- Current state:
  select * from table where endtime is NULL

- State at a particular time: Set Returning Function
  select * from table_at_time(ts)
  - Pulls tuples effective at that time

  - starttime <= ts

  - endtime is null or endtime >= ts
Explicit Temporal Tables

- Accept that it’s temporal to begin with
- Not just a way to get “history for free”
- Enables Science Fiction: Declaring future state!
  - At 9am next Wednesday, state will change
  - Eliminates need for “batch jobs”
- May need to pre-record future-dated events!
Science Fiction....
Problems

- Foreign key references into temporal tables are problematic
  - Overlap?
  - Reference disappearing?
- Fixing problems requires “fabricating a historical story” not just “fixing the state”
Temporality via Tx References

- create table transactions (  
  tx_id integer primary key default nextval('tx_seq'),  
  whodunnit integer not null references users(user_id),  
  and_when timestamptz not null default NOW());

- create table slightly_temporal_object (  
  object_id serial primary key,  
  tx_id integer not null default currval('tx_seq')  
  references transactions(tx_id));
Getting More Temporal - I

- Add ON UPDATE trigger that updates tx_id to currval ('tx_seq')
More Temporal: History!

- Create a “past history” table
  - Similar schema, but drop all data validation
  - Add end_tx
  - UPDATE/DELETE throw obsolete tuples into the “past history table”
- Data validation dropped because validation can change over time
Serial Number Temporality

- Used in DNS
  - Sets of updates grouped together temporally
  - A “bump of serial number” indicates common publishing at a common point in time
<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>Zone</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns1.abc.org</td>
<td>10.2.3.1</td>
<td>org</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ns1.abc.org</td>
<td>10.2.3.2</td>
<td>org</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ns2.abc.org</td>
<td>10.2.2.1</td>
<td>org</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ns3.abc.org</td>
<td>10.9.1.2</td>
<td>org</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ns1.abc.org</td>
<td>10.2.3.1</td>
<td>info</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>ns2.abc.org</td>
<td>10.2.3.2</td>
<td>info</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>ns2.abc.org</td>
<td>10.9.1.2</td>
<td>info</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>ns3.abc.org</td>
<td>141.2.3.4</td>
<td>info</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
Zone Representation Merits

- It’s fast. We extract multimillion record zones in minutes
- Arbitrary ability to roll back…
- Nicely supports DNS AXFR/IXFR operations
- Each serial # represents a sort of “Logical Commit”
Further Merits of this

- Rename “zone” to “module” and this is nice for configuration
- We already know it supports large amounts of data efficiently
- Configuration is smaller (we hope!)
Demerits of zone-like structure

- No way to specify a point of time in the future
- Serial numbers are intended to just keep rolling along
- HOWEVER....
- With complex apps & configuration, fancier temporality looks like a misfeature
Conclusions

- 3 ways to represent temporal information
  - Timestamps, Transaction IDs, Serial numbers
- PostgreSQL changes possible
  - Should PGtemporal be added to “core”?
  - Should we try to have temporal foreign key functionality in core?