Turing Machine in SQL

Ingvi Stígsson

Fabien Coelho

- Turing Machine in SQL
- Teacher and researcher at CRI, MINES ParisTech.
- 5 webpages published 17.-29. August 2013

- http://blog.coelho.net/database/2013/08/17/turing-sql-1/

Agenda

- TM with RECURSIVE and ARRAY
- TM with SQL functions
- TM with window function
- TM with recursive SQL function
- TM in older versions of PostrgreSQL

Turing Machine in PostgreSQL

- PostgreSQL is Turing Complete
  - Cyclic Tag System
    - Andrew Gierth (2011?)
    - http://wiki.postgresql.org/wiki/Cyclic_Tag_System

- How to build a Turing Machine with SQL only?
- What SQL features are required to build a relational Turing Machine?

Turing Machine with Arrays

- WITH RECURSIVE
- INNER JOIN
- ARRAY
- COALESCE

Tables State and Symbol

```
CREATE TABLE State(     -- TM states
  sid INTEGER PRIMARY KEY,     -- 0 is always the initial state
  isFinal BOOLEAN NOT NULL,
  sname TEXT UNIQUE NOT NULL -- just for show
);

CREATE TABLE Symbol( -- TM symbols
  cid INTEGER PRIMARY KEY,  -- 0 is always the blank symbol
  cname TEXT UNIQUE NOT NULL
);
```

Tables Transition and Tape

```
CREATE TABLE Transition( -- TM transition function
  sid INTEGER NOT NULL REFERENCES State,      -- initial state
  symbol INTEGER NOT NULL REFERENCES Symbol, -- & symbol
  UNIQUE(sid, symbol),
  new_state INTEGER NOT NULL REFERENCES State,
  new_symbol INTEGER NOT NULL REFERENCES Symbol,
  move INTEGER NOT NULL CHECK(move=-1 OR move=1)
);

CREATE TABLE Tape( -- TM initial tape contents
  tid INTEGER PRIMARY KEY,
  symbol INTEGER REFERENCES Symbol
);
```

Table Run

```
CREATE TABLE Run(
  rid INTEGER PRIMARY KEY,           -- machine iteration
  sid INTEGER NOT NULL REFERENCES State, -- current state
  tape INTEGER[] NOT NULL,           -- full tape stored as an array
  pos INTEGER NOT NULL               -- current position on tape
);
```

Turing Machine execution

```
WITH RECURSIVE running(rid, sid, tape, pos)  AS (
 -- first store initial tape contents
 SELECT 0, 0, ARRAY(SELECT symbol FROM Tape ORDER BY tid), 1
 UNION   -- then proceed to compute iterations
 SELECT p.rid+1, t.new_state,
         -- build updated tape as an array: prefix || suffix || updated
cell
  p.tape[1:p.pos-1]|| t.new_symbol|| p.tape[p.pos+1:array_length(p.tape,
1)],
         -- move cursor position
        p.pos+t.move
 FROM running AS p
 JOIN State AS s ON (p.sid=s.sid) -- get state details, to know whether to stop
 JOIN Transition AS t ON (t.sid=p.sid AND -- get corresponding state transition
            t.symbol=COALESCE(p.tape[p.pos], 0))   -- coalesce defaults to blank
  WHERE NOT s.isFinal -- stop on a final state
```

Store the computed table

```
-- just store the computed table
INSERT INTO Run
  SELECT * FROM running;
```

Turing Machine with SQL functions

- WITH RECURSIVE
- INNER JOIN
- Table for evolving tape contents
  - INSERT
  - UPDATE

New table: RunningTape

```
CREATE TABLE RunningTape(
  tid SERIAL PRIMARY KEY, -- implicit sequence!
  symbol INTEGER NOT NULL REFERENCES Symbol
);
INSERT INTO RunningTape(symbol) -- initial tape contents
  SELECT symbol FROM Tape ORDER BY tid;
```

Function updRunnigTape

```
-- update tape as a SQL function side effect
CREATE OR REPLACE FUNCTION
  updRunningTape(pos INTEGER, nsymbol INTEGER) RETURNS INTEGER[]
VOLATILE STRICT AS $$
  -- ensure that the tape is long enough, symbol 0 is blank
  INSERT INTO RunningTape(symbol) VALUES(0);
  -- update tape contents
  UPDATE RunningTape SET symbol=nsymbol WHERE tid=pos;
  -- return tape as an array for later display
  SELECT ARRAY(SELECT symbol FROM RunningTape ORDER BY tid)
$$ LANGUAGE SQL;
```

Function getRunningTape

```
-- get table contents from a function...
CREATE OR REPLACE FUNCTION
  getRunningTape(pos INTEGER) RETURNS RunningTape
VOLATILE STRICT AS $$
  SELECT * FROM RunningTape WHERE tid=pos;
$$ LANGUAGE SQL;
```

Turing Machine execution

```
-- iteration, state, tape (for record, not used!), position
WITH RECURSIVE running(rid, sid, tape, pos) AS (
  -- set first iteration at state 0, position 1
  SELECT 0, 0, ARRAY(SELECT symbol FROM RunningTape ORDER BY tid), 1
  UNION   -- compute next iterations
  SELECT pr.rid+1, tr.new_state,
    -- this function relies on side effects to update the tape
    -- the tape is stored as an array just for showing it later
    updRunningTape(pr.pos, tr.new_symbol),
    pr.pos + tr.move
  FROM running AS pr -- previous state
  CROSS JOIN LATERAL getRunningTape(pr.pos) AS tp -- current symbol from tape
  JOIN Transition AS tr ON (tr.sid=pr.sid AND tr.symbol=tp.symbol) -- corresponding
transition
  JOIN State AS st ON (st.sid=tr.sid) -- state information, necessary to know whether to stop
  WHERE NOT st.isFinal -- stop on a final state
)
```

Turing machine with a window function

- WITH RECURSIVE
- OVER (window function)
- CROSS JOIN

Turing Machine execution

```
WITH RECURSIVE running(iter, sid, len, pos, psym, tid, tsym)  AS (
  -- set first iteration at state 0, position 1
    SELECT
      -- first, common part is repeated over and over
      0, 0,
      -- tape length needed to know where to insert blanks
      (SELECT COUNT(*)::INTEGER FROM Tape),
      -- position and next symbol to consider
      1, (SELECT symbol FROM Tape WHERE tid=1),
      -- then the tape contents
      tid, symbol
    FROM Tape
  UNION
```

Turing Machine execution cont.

```
 UNION   -- compute next iteration
 SELECT pr.iter + 1, tr.new_state,
       pr.len,  -- the initial length could also be recomputed with a sub-query
       pr.pos + tr.move,
        -- recover next iteration symbol
        -- this "hack" because 'running' cannot be used twice in the query
   MAX(CASE WHEN pr.pos+tr.move=pr.tid THEN pr.tsym ELSE NULL END) OVER (),
   CASE WHEN hack.keep THEN pr.tid    -- tape index
        ELSE pr.len + pr.iter + 1 END,    -- append a new index
   CASE WHEN hack.keep AND pr.tid=pr.pos THEN tr.new_symbol   -- update symbol
        WHEN hack.keep THEN pr.tsym                           -- or keep previous symbol
        ELSE 0 END                                            -- or append a blank symbol
  FROM running AS pr
  JOIN Transition AS tr ON (pr.sid=tr.sid AND pr.psym=tr.symbol)  -- corresponding transition
  JOIN State AS st ON (tr.sid=st.sid) -- state information, necessary to know whether to stop
  CROSS JOIN (VALUES (TRUE), (FALSE)) AS hack(keep) --hack to append a 0 at the end of tape
    WHERE -- stop on a final state
          NOT st.isFinal
)
```

Turing Machine

With a recursive SQL function

Changed table RunningTape

```
-- create initial tape contents
CREATE TABLE RunningTape(
  tid SERIAL PRIMARY KEY, -- implicit sequence there
  symbol INTEGER NOT NULL REFERENCES Symbol,
  -- previous symbol needed temporarily between an update & its recursion
  psymbol INTEGER REFERENCES Symbol DEFAULT NULL
);

INSERT INTO RunningTape(symbol)
  SELECT symbol FROM Tape ORDER BY tid;
);
```

Turing machine execution

```
- update tape as a recursive SQL function side effect
CREATE OR REPLACE FUNCTION
  recRun(ite INTEGER, sta INTEGER, pos INTEGER) RETURNS
INTEGER
VOLATILE STRICT AS $$
  -- keep a trace for later display
  INSERT INTO Run(rid, sid, pos, tape)
  VALUES (ite, sta, pos,
          ARRAY(SELECT symbol FROM RunningTape ORDER BY
tid));
  -- ensure that the tape is long enough, symbol 0 is
blank
  INSERT INTO RunningTape(symbol) VALUES(0);
```

Turing Machine execution cont.

```
-- update tape contents
  UPDATE RunningTape AS tp
  SET
    symbol = tr.new_symbol, -- update the tape symbol
    psymbol = tr.symbol -- but keep a copy as we need it
again for the recursion
  FROM Transition AS tr
  WHERE tr.sid = sta
    AND tr.symbol = tp.symbol
    AND tp.tid = pos;
```

Turing Machine exectution cont.

```
-- now the recursion
  SELECT
    CASE
      WHEN st.isFinal THEN st.sid -- stop recursion on a final state
      ELSE recRun(ite+1, tr.new_state, pos+tr.move)   -- or do
*recurse*
    END
  FROM Transition AS tr
  JOIN RunningTape AS tp ON (tp.psymbol = tr.symbol) -- use
*previous* symbol
  JOIN State AS st USING (sid)
  WHERE st.sid = sta AND tp.tid = pos;
$$ LANGUAGE SQL;
```

Turing Machine execution cont.

```
SELECT recRun(0, 0, 1); -- start Turing Machine
```

Previous versions of PostgreSQL

- PostgreSQL 8.4 (2009): Yes (without parameter names)
- PostgreSQL 8.2 (2006): Yes
- PostgreSQL 8.1/8.0 (2005): Yes (with changes)
- PostgreSQL 7.4 (2003): Yes (without dollar quoting $$)
- PostgreSQL 7.3 (2002): Yes (without array)

- PostgreSQL 7.2.8 (2005): No