Big Data Infrastructures - Homework 2: SQL Review Part 2

Markus Roth

October 10, 2016

1 A: Indexes

1.1 Create Table

```
CREATE TABLE Position (
   positionId BIGSERIAL PRIMARY KEY,
   timestamp TIMESTAMP NOT NULL,
   lat NUMERIC NOT NULL,
   lon NUMERIC NOT NULL,
   alt NUMERIC NOT NULL
);
```

2 Populate Table

```
import psycopg2
import os
import random
import string
import datetime
import time


def cursorize(func):
   def func_wrapper(conn, *args):
      cursor = conn.cursor()
      func(cursor, *args)
      conn.commit()
      cursor.close()
   return func_wrapper

def strTimeProp(start, end, format, prop):
   """
   from:
       http://stackoverflow.com/questions/553303/generate-a-random-date-between-two-other-dates
   """
```

```
    stime = time.mktime(time.strptime(start, format))
    etime = time.mktime(time.strptime(end, format))

    ptime = stime + prop * (etime - stime)

    return time.strftime(format, time.localtime(ptime))

def randomDate(start, end, prop):
    """
    from:
        http://stackoverflow.com/questions/553303/generate-a-random-date-between-two-other-dates
    """
    return strTimeProp(start, end, '%Y-%m-%d %H:%M:%S', prop)


@cursorize
def create_position(cursor):
    timestamp = randomDate("2016-01-01 00:00:00", "2017-01-01 00:00:00",
        random.random())
    lat = (random.random() * 180) - 90 #value between -90 and 90
    lon = (random.random() * 360) - 180 #value between -180 and 180
    alt = random.random() * 10000
    cursor.execute(str.format("INSERT INTO Position(timestamp, lat, lon,
        alt) VALUES ('{}', {}, {}, {});", timestamp, lat, lon, alt))


if __name__ == "__main__":
    os.system("./create_schema.sh")
    conn = psycopg2.connect("dbname=homework2 user=markus
        password=markus")

    for _ in range(100000):
       create_position(conn)

    conn.close()
```

2.1 Choose Rectangular Area

```
SELECT *
FROM Position p
WHERE p.lat >= 60
AND p.lat <= 90
AND p.lon >= 20
AND p.lon <= 44
```

2.2 Describe Execution Plan

```
 Seq Scan on "position" p (cost=0.00..2944.00 rows=1154 width=45)
     (actual time=0.025..18.994 rows=1088 loops=1)
```

```
  Filter: ((lat >= 60::numeric) AND (lat <= 90::numeric) AND (lon >=
      20::numeric) AND (lon <= 44::numeric))
  Rows Removed by Filter: 98912
Planning time: 0.315 ms
Execution time: 20.259 ms
```

Seq scan on "position p" means that the database engine must start reading at the beginning of the table and check each and every row of the table. If the row fullfills the conditions, it will be part of the result set, otherwise it won't be.

The execution time of the query is around 20 ms.

2.3 Create Indexes

```
CREATE INDEX lat_index ON Position USING BTREE(lat);
CREATE INDEX lon_index ON Position USING BTREE(lon);
```

2.4 Re-Check Execution Plan

```
Bitmap Heap Scan on "position" p (cost=176.87..1257.19 rows=1155
    width=45) (actual time=1.188..5.596 rows=1088 loops=1)
  Recheck Cond: ((lon >= 20::numeric) AND (lon <= 44::numeric))
  Filter: ((lat >= 60::numeric) AND (lat <= 90::numeric))
  Rows Removed by Filter: 5631
  Heap Blocks: exact=944
  -> Bitmap Index Scan on lon_index (cost=0.00..176.58 rows=6816
      width=0) (actual time=1.085..1.085 rows=6719 loops=1)
       Index Cond: ((lon >= 20::numeric) AND (lon <= 44::numeric))
Planning time: 0.272 ms
Execution time: 6.555 ms
```

The new execution first uses a Bitmap Heap Scan on "position p", where it filters for the lat condition. After that, it uses a Bitmap Index Scan over the lon_index on the lon condition.

The execution time of the query is now around 7 ms, quite an improvement!

2.5 Bitmap Index Scan

A Bitmap Index Scan optimizes an index scan to only load disk pages once each. This is good if there is some data locality, but not necessarity good if the data is completely randomly stored [1].

3 B: Relational Queries

3.1 i

[1] urlhttp://stackoverflow.com/questions/6592626/what-is-a-bitmap-heap-scan-in-a-query-plan

```
SELECT name
FROM Author a
NATURAL JOIN Writes
GROUP BY name, paperId
HAVING count(paperId) = 1;
```

3.2 ii

```
SELECT DISTINCT a1.email
FROM Author a1
NATURAL JOIN Writes w1
NATURAL JOIN Submits s1
WHERE NOT EXISTS (
    SELECT *
    FROM Author a2
    NATURAL JOIN Writes w2
    WHERE w2.paperId = w1.paperId
    AND a1.authorId != a2.authorId
)
```

3.3 iii

```
SELECT affiliation
FROM Author
NATURAL JOIN Writes
GROUP BY affiliation
HAVING count(paperid) >= ALL (
    SELECT count(paperid)
    FROM Author
    NATURAL JOIN Writes
    GROUP BY affiliation
)
```

3.4 iv

```
SELECT name, count(paperId)
FROM Author
NATURAL JOIN Writes
NATURAL JOIN Submits
WHERE isAccepted = TRUE
GROUP BY authorid
HAVING count(paperId) >= ALL (
    SELECT count(paperId)
    FROM Author
    NATURAL JOIN Writes
    NATURAL JOIN Submits
    WHERE isAccepted = TRUE
```

```
    GROUP BY authorid
)
```

3.5 v

```
SELECT authorId
FROM Writes w5
JOIN Writes w6 ON w5.paperId = w6.paperId
WHERE w6.authorid IN ALL (
    SELECT w1.authorId
    FROM Writes w1
    JOIN Writes w2 ON w1.paperId = w2.paperId
    WHERE w1.authorId != w2.authorId
    AND w1.authorId = w5.authorId
)
OR w6.authorId NOT IN ANY (
    SELECT w3.authorId
    FROM Writes w3
    JOIN Writes w4 ON w3.paperId = w4.paperId
    WHERE w3.authorId != w4.authorId
    AND w3.authorId = w5.authorId
)
```