Why is Solr so much faster than Postgres?

I recently switched from Postgres to Solr and saw a ~50x speed up in our queries. The queries we run involve multiple ranges, and our data is vehicle listings. For example: "Find all vehicles with mileage < 50,000, $5,000 < price < $10,000, make=Mazda..."

I created indices on all the relevant columns in Postgres, so it should be a pretty fair comparison. Looking at the query plan in Postgres though it was still just using a single index and then scanning (I assume because it couldn't make use of all the different indices).

As I understand it, Postgres and Solr use vaguely similar data structures (B-trees), and they both cache data in-memory. So I'm wondering where such a large performance difference comes from.

What differences in architecture would explain this?

performance postgresql solr lucene rdbms

edited Apr 7 '12 at 11:32

jpountz
7,212 15 29

asked Apr 7 '12 at 8:40

cberner
1,234 2 8 28

3 Did you use Postgres full text search or simple `LIKE` queries? – a_horse_with_no_name Apr 7 '12 at 9:36

The relational model was never intended to perform well on ugly queries like `give me all the people who were born on a wednesday and owned a red car between 2003 and 2005` . There are search engines (such as Lucene) for that. Indexes do help sometimes, memory settings always help. – wildplasser Apr 7 '12 at 12:44

1 I am a bit confused by your question, so I ask here: dba.stackexchange.com/questions/34014/… Will/can Solr/Lucene searches be faster than PostgreSQL even if no full-text search is involved? – alfonx Feb 2 '13 at 13:07

If there was a loaded question - this is it. I bet you haven't even considered using any partial indexes. – vonPetrushev Jun 24 '13 at 16:50

What version of PostgreSQL was that? If all columns were indexed, I'm surprised it didn't use bitmap index scans, unless you were using a very old version of PostgreSQL or your statistics were not up-to-date. When it uses such a plan it scans multiple indexes for the ranges of values, building up a bitmap of row locations, then applies the boolean logic to the bitmaps and retrieves only the matching rows, in location order (to optimize disk access). – kgrittn Aug 26 '13 at 20:43

5 Answers

First, Solr doesn't use B-trees. A Lucene (the underlying library used by Solr) index is made of a read-only segments. For each segment, Lucene maintains a term dictionary, which consists of the list of terms that appear in the segment, lexicographically sorted. Looking up a term in this term dictionary is made using a binary search, so the cost of a single-term lookup is `O(log(t))` where t is the number of terms. On the contrary, using the index of a standard RDBMS costs `O(log(d))` where d is the number of documents. When many documents share the same value for some field, this can be a big win.

Moreover, Lucene committer Uwe Schindler added support for very performant numeric range queries a few years ago. For every value of a numeric field, Lucene stores several values with different precisions. This allows Lucene to run range queries very efficiently. Since your use-case seems to leverage numeric range queries a lot, this may explain why Solr is so much faster. (For more information, read the javadocs which are very interesting and give links to relevant research papers.)

But Solr can only do this because it doesn't have all the constraints that a RDBMS has. For example, Solr is very bad at updating a single document at a time (it prefers batch updates).

answered Apr 7 '12 at 11:31

jpountz
7,212 15 29

2 Great answer (the first paragraph) +1. – Yavar Apr 7 '12 at 11:35

2 Though I have one thing to say here. When you say "Lucene maintains a term dictionary, which consists of the list of terms that appear in the segment, lexicographically sorted.". So if the terms are lexicographically sorted & it takes o(log t) time (Binary Search) that means the terms are stored in an array? Is that correct (I mean as you rejected a balanced tree structure for storing indexes and hash table cant store straightaway sorted indexes so we are just left with a 2D array (with term, pointer to postings list as its elements) structure on which binary search is carried on) – Yavar Apr 7 '12 at 11:51

3 Another thing to add here is even if the Lucene Index is not a BTree, it is however an inverted index (like most of search engine). Your answer though came as something new to me. As I was expecting a Balanced Tree structure to store terms (In that case too search would be log (t) and each node will also contain pointer to posting lists. With Balance Trees we would even be able to maintain lexicographically sorted terms. – Yavar Apr 7 '12 at 11:56

3 There is no need for a balance tree, since the data never needs to be updated. When data is added to a Lucene index, a new segment is created. This segment has precedence over the previous segments. When there are too many segments, a MergeScheduler chooses the segments to merge according to a MergePolicy (these are the names of the classes in Lucene). – jpountz Apr 7 '12 at 12:06

4 When a new SegmentReader (an IndexReader over a single segment) is open, it loads by default every (n*16)th term of the index in a Java array in memory (16 is the `indexDivisor`). Then the lookup is performed by a binary search in memory over this array, then one disk seek and at most 15 term comparisons on disk. So the overall cost is `O(log(t/16)) + O(1) + O(15) = O(log(t))`. – jpountz Apr 7 '12 at 12:07

You didn't really say much about what you did to tune your PostgreSQL instance or your queries. It's not unusual to see a 50x speed up on a PostgreSQL query through tuning and/or restating your query in a format which optimizes better.

Just this week there was a report at work which someone had written using Java and multiple queries in a way which, based on how far it had gotten in four hours, was going to take roughly a month to complete. (It needed to hit five different tables, each with hundreds of millions of rows.) I rewrote it using several CTEs and a window function so that it ran in less than ten minutes and generated the desired results straight out of the query. That's a 4400x speed up.

Perhaps the best answer to your question has nothing to do with the technical details of how searches can be *performed* in each product, but more to do with *ease of use* for your particular use case. Clearly you were able to find the fast way to search with Solr with less trouble than PostgreSQL, and it may not come down to anything more than that.

I am including a short example of how text searches for multiple criteria might be done in PostgreSQL, and how a few little tweaks can make a large performance difference. To keep it quick and simple I'm just running *War and Peace* in text form into a test database, with each "document" being a single text line. Similar techniques can be used for arbitrary fields using the `hstore` type or `JSON` columns, if the data must be loosely defined. Where there are separate columns with their own indexes, the benefits to using indexes tend to be much bigger.

```
-- Create the table.
-- In reality, I would probably make tsv NOT NULL,
-- but I'm keeping the example simple...
CREATE TABLE war_and_peace
  (
    lineno serial PRIMARY KEY,
    linetext text NOT NULL,
    tsv tsvector
  );

-- Load from downloaded data into database.
COPY war_and_peace (linetext)
  FROM '/home/kgrittn/Downloads/war-and-peace.txt';

-- "Digest" data to lexemes.
UPDATE war_and_peace
  SET tsv = to_tsvector('english', linetext);

-- Index the lexemes using GiST.
-- To use GIN just replace "gist" below with "gin".
CREATE INDEX war_and_peace_tsv
  ON war_and_peace
  USING gist (tsv);

-- Make sure the database has statistics.
VACUUM ANALYZE war_and_peace;
```

Once set up for indexing, I show a few searches with row counts and timings with both types of

indexes:

```
-- Find lines with "gentlemen".
EXPLAIN ANALYZE
SELECT * FROM war_and_peace
  WHERE tsv @@ to_tsquery('english', 'gentlemen');
```

84 rows, gist: 2.006 ms, gin: 0.194 ms

```
-- Find lines with "ladies".
EXPLAIN ANALYZE
SELECT * FROM war_and_peace
  WHERE tsv @@ to_tsquery('english', 'ladies');
```

184 rows, gist: 3.549 ms, gin: 0.328 ms

```
-- Find lines with "ladies" and "gentlemen".
EXPLAIN ANALYZE
SELECT * FROM war_and_peace
  WHERE tsv @@ to_tsquery('english', 'ladies & gentlemen');
```

1 row, gist: 0.971 ms, gin: 0.104 ms

Now, since the GIN index was about 10 times faster than the GiST index you might wonder why anyone would use GiST for indexing text data. The answer is that GiST is generally faster to maintain. So if your text data is highly volatile the GiST index might win on overall load, while the GIN index would win if you are only interested in search time or for a read-mostly workload.

Without the index the above queries take anywhere from 17.943 ms to 23.397 ms since they must scan the entire table and check for a match on each row.

The GIN indexed search for rows with both "ladies" and "gentlemen" is over 172 times faster than a table scan in exactly the same database. Obviously the benefits of indexing would be more dramatic with bigger documents than were used for this test.

The setup is, of course, a one-time thing. With a trigger to maintain the `tsv` column, any changes made would instantly be searchable without redoing any of the setup.

With a slow PostgreSQL query, if you show the table structure (including indexes), the problem query, and the output from running `EXPLAIN ANALYZE` of your query, someone can almost always spot the problem and suggest how to get it to run faster.

edited Aug 26 '13 at 20:57

answered Apr 7 '12 at 14:55

kgrittn

6,473 11 32

This biggest difference is that a Lucene/Solr index is like a single-table database without any support for relational queries (JOINs). Remember that an index is usually only there to support search and not to be the primary source of the data. So your database may be in "third normal form" but the index will be completely be de-normalized and contain mostly just the data needed to be searched.

Another possible reason is generally databases suffer from internal fragmentation, they need to perform too much semi-random I/O tasks on huge requests.

What that means is, for example, considering the index architecture of a databases, the query leads to the indexes which in turn lead to the data. If the data to recover is widely spread, the result will take long and that seems to be what happens in databases.

answered Apr 7 '12 at 10:19

Yavar

5,964 1 16 44

Solr is designed primarily for searching data, not for storage. This enables it to discard much of the functionality required from an RDMS. So it (or rather lucene) concentrates on purely indexing data.

As you've no doubt discovered, Solr enables the ability to both search and retrieve data from it's index. It's the latter (optional) capability that leads to the natural question... "Can I use Solr as a database?"

The answer is a qualified yes, and I refer you to the following:

- Solr or database?
- Using Solr search index as a database - is this "wrong"?
- For the guardian solr is the new database

My personal opinion is that Solr is best thought of as a searchable cache between my application and the data mastered in my database. That way I get the best of both worlds.

answered Apr 7 '12 at 9:30

Mark O'Connor
39.9k 3 46 93

Please read this and this.

Solr (Lucene) creates an inverted index which is where retrieving data gets quite faster. I read that PostgreSQL also has similar facility but not sure if you had used that.

The performance differences that you observed can also be accounted to "what is being searched for ?", "what are the user queries ?"

edited Apr 7 '12 at 9:36

answered Apr 7 '12 at 8:52

Tejas Patil
4,015 1 6 23

Thanks! Those were very interesting. I was hoping for something a bit more technical though. Like an architecture overview of Solr, or something like that. – cberner Apr 7 '12 at 9:16

@Tejas: Even databases can create inverted indexes. What's stopping them to create inverted indexes? – Yavar Apr 7 '12 at 10:21

Yavar: I didnt say that databases cannot create inverted indexes. In fact in 2nd line I pointed to link about PostgreSQL using GIN-inverted index. There is another type: GiST(Generalized Search Tree)-based index in PostgreSQL which SLOWER than GIN. The actual index type used by @cberner will be one factor for the low performance of PostgreSQL. – Tejas Patil Apr 7 '12 at 12:28
