Counting Triangles

by swalkauskas@vertica.com on September 21st, 2011 • in Hadoop, social graph analysis

Previous Post Next Post

by Stephen Walkauskas

Recently I've heard from or read about people who use Hadoop because their analytic jobs can't achieve the same level of performance in a database. In one case, a professor I visited said his group uses Hadoop to count triangles "because a database doesn't perform the necessary joins efficiently."

Perhaps I'm being dense but I don't understand why a database doesn't efficiently support these use-cases. In fact, I have a hard time believing they wouldn't perform better in a columnar, MPP database like Vertica – where memory and storage are laid out and accessed efficiently, query jobs are automatically tuned by the optimizer, and expression execution is vectorized at run-time. There are additional benefits when several, similar jobs are run or data is updated and the same job is re-run multiple times. Of course, performance isn't everything; ease-of-use and maintainability are important factors that Vertica excels at as well.

Since the "gauntlet was thrown down", to steal a line from Good Will Hunting, I decided to take up the challenge of computing the number of triangles in a graph (and include the solutions in GitHub so others can experiment – more on this at the end of the post).

Problem Description

A triangle exists when a vertex has two adjacent vertexes that are also adjacent to each other. Using friendship as an example: If two of your friends are also friends with each other, then the three of you form a friendship triangle. How nice. Obviously this concept is useful for understanding social networks and graph analysis in general (e.g. it can be used to compute the clustering coefficient of a graph).

Let's assume we have an undirected graph with reciprocal edges, so there's always a pair of edges ({e1,e2} and {e2,e1}). We'll use the following input for illustration (reciprocal edge is elided to condense the information):

source	destination
Ben	Chuck
Ben	Stephen
Chuck	Stephen
Chuck	Rajat
Rajat	Stephen
Andrew	Ben
Andrew	Matt
Matt	Pachu
Chuck	Lyric

A little ascii art to diagram the graph might help.

Get Started Wi

Commu

Evaluate

New Cus

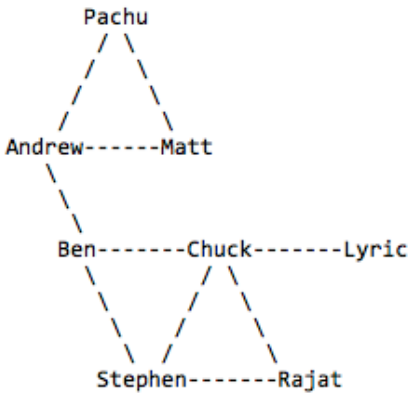
Subscribe to V

Most Pop

Cardlytics Powe
Counting Triang
The Power of Pr
Announcing the
Introducing Vert

Archives

March 2013
February 2013
January 2013
December 2012
November 2012
October 2012
September 2012
August 2012
July 2012
June 2012
April 2012
March 2012
October 2011
September 2011
August 2011



I know you can quickly count the number of triangles. I'm very proud of you but imagine there are hundreds of millions of vertexes and 10s of billions of edges. How long would it take you to diagram that graph? And how much longer to count all of the triangles? And what if your 2 year old daughter barges in counting "one, two, three, four, …" and throws off your count?

Below we present a few practical solutions for large scale graphs and evaluate their performance.

The Hadoop Solution

Let's consider first the Hadoop approach to solving this problem. The MapReduce (MR) framework implemented in Hadoop allows us to distribute work over many computers to get the count faster. The solution we describe here is a simplified version of the work at Yahoo Research. You can download our solution here.

Overview

The solution involves a sequence of 3 MR jobs. The first job constructs all of the triads in the graph. A *triad* is formed by a pair of edges sharing a vertex, called its *apex*. It doesn't matter which vertex we choose as the apex of the triad, so for our purposes we'll pick the "lowest" vertex (e.g. friends could be ordered alphabetically by their names). The Yahoo paper makes a more intelligent choice of "lowest" – the vertex with the smallest degree. However that requires an initial pass of the data (and more work on my part) so I skipped that optimization and did so consistently for all solutions to ensure fairness.

These triads and the original edges are emitted as rows by the first MR job, with a field added to distinguish the two. Note that the output of the first job can be quite large, especially in a dense graph. Such output is consumed by the second MR job, which partitions the rows by either the unclosed edge, if the row is a triad, or the original edge. A partition has *n* triangles if it contains an original edge and *n* triads. A third, trivial MR job counts the triangles produced by the second job, to produce the final result.

Details

Let's look at each MR job in detail. The map part of the first job generates key-value pairs for each triad such that the apex is the key and the value is the edge. In our small example the map job would emit the following rows.

key	value
Andrew	Andrew, Matt
Andrew	Andrew, Pachu
Andrew	Andrew, Ben
Matt	Matt, Pachu
Ben	Ben, Chuck
Ben	Ben, Stephen
Chuck	Chuck, Rajat
Chuck	Chuck, Lyric

June 2011
February 2011
January 2011
November 2010
October 2010
September 2010
August 2010
June 2010
May 2010
April 2010
January 2010
December 2009
December 2008
July 2008
January 2008
October 2007
September 2007

Categories

bbbt (1)
big data (35)
Bulldozer (2)
column store (16
Community (1)
compression (4)
Customers (1)
data loading (2)
Data Scientists (
Engineering (8)
External Tables
Fault Tolerance
Hadoop (11)
HDFS (4)
HP Discover (1)
HP IT (2)
in-database ana
interns (7)
merge (1)
Moneyball (2)
MyVertica (1)
pattern matching
R programming

Chuck	Chuck, Stephen
Rajat	Rajat, Stephen

For each apex-partition, the reduce job emits the original edges and all of the corresponding triads (there are ?(j-1) -> j=1 to d triads per partition, where d is the degree of the vertex at the apex). For each original edge, the key is the edge itself and the value is “edge”. For each triad, the key is the unclosed edge. In other words, the edge needed to complete the triangle. The value is “triad.” The actual code used “0″ for the edge value and “1″ for the triad value for run-time efficiency.

The rows corresponding to the triads emitted by this reduce job in our simple example are described below in the “key” and “value” columns (the original edges are also emitted by the reduce job but elided below for brevity). For presentation purposes we added a third column “triad content”. That column is not produced by the actual reduce job.

key	value	triad content
Ben, Matt	triad	{Andrew, Ben}, {Andrew, Matt}
Ben, Pachu	triad	{Andrew, Ben}, {Andrew, Pachu}
Matt, Pachu	triad	{Andrew, Matt}, {Andrew, Pachu}
Chuck, Stephen	triad	{Ben, Chuck}, {Ben, Stephen}
Lyric, Rajat	triad	{Chuck, Lyric}, {Chuck, Rajat}
Lyric, Stephen	triad	{Chuck, Lyric}, {Chuck, Stephen}
Rajat, Stephen	triad	{Chuck, Rajat}, {Chuck, Stephen}

The input to the next reduce job is partitioned such that the unclosed edge of each triad is in the same partition as its corresponding original edge, if any. The reduce job just needs to check for the existence of an original edge in that partition (i.e., a row with value set to “edge”). If it finds one, all of the triads in the partition are closed as triangles. The reduce job sums up all of the closed triads and on finalize emits a count. A trivial final MR job aggregates the counts from the previous job.

There we’ve used MapReduce to count the number of triangles in a graph. The approach isn’t trivial but it’s not horribly complex either. And if it runs too slowly we can add more hardware, each machine does less work and we get our answer faster.

Experiences with Hadoop

I have to admit it took me much longer than I estimated to implement the Hadoop solution. Part of the reason being I’m new to the API, which is exacerbated by the fact that there are currently two APIs, one of them deprecated, the other incomplete, forcing use of portions of the deprecated API. Specifically, the examples I started with were unfortunately based on the deprecated API and when I ported to the newer one I ran into several silly but somewhat time consuming issues (like mapred’s version of *Reducer.reduce* takes an *Iterator* but mapreduce’s version takes an *Iterable* – they look similar to the human eye but the compiler knows that a method that takes an *Iterator* should not be overridden by one that takes an *Iterable*). Learning curve aside there was a fair chunk of code to write. The simple version is >200 lines. In a more complex version I added a secondary sort to the MR job that computes triads. Doing so introduced several dozen lines of code (most of it brain dead stuff like implementing a *Comparable* interface). Granted a lot of the code is cookie cutter or trivial but it still needs to be written (or cut-n-pasted and edited). In contrast, to add a secondary sort column in SQL is a mere few characters of extra code.

The PIG Solution

Rajat Venkatesh, a colleague of mine, said he could convert the algorithm to a relatively small PIG script and he wagered a lunch that the PIG script would outperform my code. He whipped up what was eventually a 10 statement PIG script that accomplished the task. When we get to the performance comparison we’ll find out who got a free lunch.

security (1)
social graph ana
sorted data (5)
SQL (5)
third-party tools
Uncategorized (
updates & delet
use cases (6)
User-Defined Lo
vertica (42)
Vertica 6 (3)
Vertica 6.1 (3)
Vertica Commur
Vertica Custome
Vertica OEM (1)
Video Webinars
VLDB (2)
workload analyz

```
set default_parallel N;
set mapreduce.job.maps M;
EDGES         = load 'input/few-edges.txt' using PigStorage(' ') as (source:long, dest:long);
CANON_EDGES_1 = filter EDGES by source < dest;
CANON_EDGES_2 = filter EDGES by source < dest;
TRIAD_JOIN    = join CANON_EDGES_1 by dest, CANON_EDGES_2 by source;
OPEN_EDGES    = foreach TRIAD_JOIN generate CANON_EDGES_1::source, CANON_EDGES_2::dest;
TRIANGLE_JOIN = join CANON_EDGES_1 by (source,dest), OPEN_EDGES by (CANON_EDGES_1::source, CANON_EDGES_2::dest);
TRIANGLES     = foreach TRIANGLE_JOIN generate 1 as a:int;
CONST_GROUP   = group TRIANGLES ALL parallel 1;
FINAL_COUNT   = foreach CONST_GROUP generate COUNT(TRIANGLES);

dump FINAL_COUNT;
```

Here's the PIG solution, much simpler than coding MR jobs by hand. We used PIG 0.8.1. We made several passes over the script to optimize it, following the PIG Cookbook. For example, we rearranged the join order and put the larger table last (I'm probably not giving too much away by mentioning that Vertica's optimizer uses a cost model which properly chooses join order). We also tried several values for default_parallel and mapreduce.job.maps (and we changed the corresponding parameter in mapred-site.xml as well, just to be safe). We did not enable lzo compression for two reasons. First, considering the hardware used for the experiment (large RAM – plenty of file system cache, high throughput network), the CPU tax incurred by compression was more likely to hurt performance than help in this case. Second, one website listed 7 steps to get the compression working but the 2nd step had several steps itself, so I gave up on it.

The Vertica Solution

Can you count the number of triangles in a graph using a database? Of course. First create an "edges" table and load the graph. Vertica can automate the decision about how to organize storage for the table – something called projections specify important characteristics of physical storage such as sort order, segmentation, encoding and compression. In this case we simply tell Vertica how to distribute data among nodes in our cluster (Vertica calls this segmenting). Alternatively the Vertica Database Designer can be used to automate projection design. The following statements create our table and load data.

```
create table edges (source int not null, dest int not null) segmented by hash(source,dest) all nodes;
copy edges from :file direct delimiter ' ';
```

We've got the data loaded and stored in an efficient way. If we need to run more jobs with the same data later we won't incur the load cost again. Likewise, if we need to modify the data we only incur work proportional to the change. Now we just need a horribly complex hack to count triangles. Take a deep breath, stretch, get a cup of coffee, basically do what you have to do to prepare your brain for this challenge. Ok, ready? Here it is:

```
select count(*)
  from edges e1
  join edges e2 on e1.dest = e2.source and e1.source < e2.source
  join edges e3 on e2.dest = e3.source and e3.dest = e1.source and e2.source < e3.source;
```

Good, you didn't run away screaming in horror. If we ignore the less than predicates the query is simply finding all triplets that form a cycle, v1 -> v2 -> v3 -> v1. The less than predicates ensure we don't count the same triangle multiple times (remember our edges are reciprocal and we only need to consider triangles with the "lowest" vertex at the apex).

That's it! A single, 4-liner query. Of course you're interested in what the Vertica database does under the covers and how its performance, disk utilization and scalability compare with those of Hadoop and PIG.

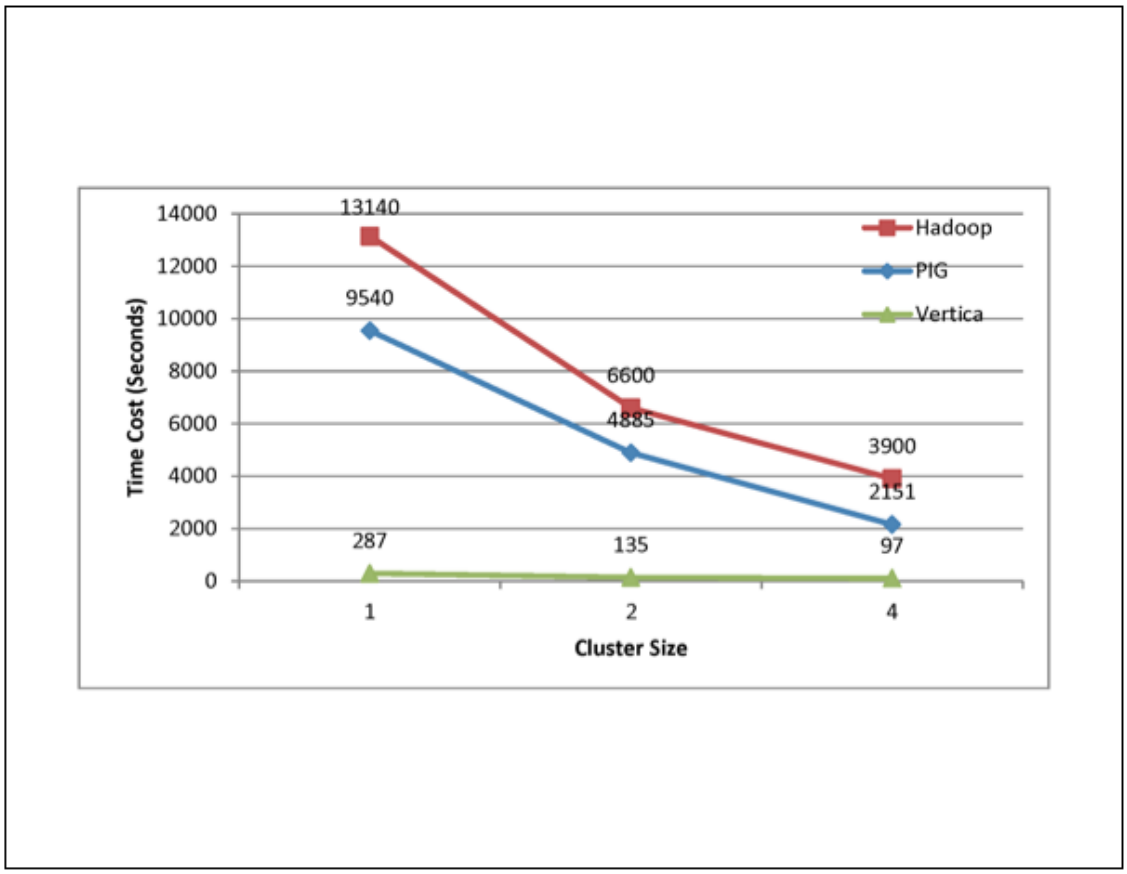
Performance Study

The publicly available LiveJournal social network graph (http://snap.stanford.edu/data/soc-LiveJournal1.html) was used to test performance. It was selected because of its public availability, its modest size permitted relatively quick experiments. The modified edges file (in the original file not every edge is reciprocated) contained 86,220,856

edges, about 1.3GB in raw size. We used HDFS dfs.replication=2 (replication=1 performed worse – fewer map jobs were run, almost regardless of the mapreduce.job.maps value). Experiments were run on between 1 and 4 machines each with 96GB of RAM, 12 cores and 10GBit interconnect.

Run-Time Cost

All solutions are manually tuned to obtain the best performance numbers. For the Hadoop and PIG solutions, the number of mappers and reducers as well as the code itself were tweaked to optimize performance. For the Vertica solution, out-of-the-box Vertica is configured to support multiple users; default expectation is 24 concurrent queries for the hardware used. This configuration was tweaked to further increase pipeline parallelism (equivalent configuration settings will be on by default in an upcoming release). The following chart compares the best performance numbers for each solution.



PIG beat my Hadoop program, so my colleague who wrote the PIG script earned his free lunch. One major factor is PIG's superior join performance – its uses hash join. In comparison, the Hadoop solution employs a join method very close to sort merge join.

Vertica's performance wasn't even close to that of Hadoop – thankfully. It was much much better. In fact Vertica ate PIG's and Hadoop's lunch – its best time is 22x faster than PIG's and 40x faster than the Hadoop program (even without configuration tweaks Vertica beats optimized Hadoop and PIG programs by more than a factor of 9x in comparable tests).

Here are a few key factors in Vertica's performance advantage:

- Fully pipelined execution in Vertica, compared to a sequence of MR jobs in the Hadoop and PIG solutions, which incurs significant extra I/O. We quantify the differences in how the disk is used among the solutions below in the "disk usage" study.
- Vectorization of expression execution, and the use of just-in-time code generation in the Vertica engine
- More efficient memory layout, compared to the frequent Java heap memory allocation and deallocation in Hadoop / PIG

Overall, Hadoop and PIG are free in software, but hardware is not included. With a 22x speed-up, Vertica's performance advantage effectively equates to a 95% discount on hardware. Think about that. You'd need 1000

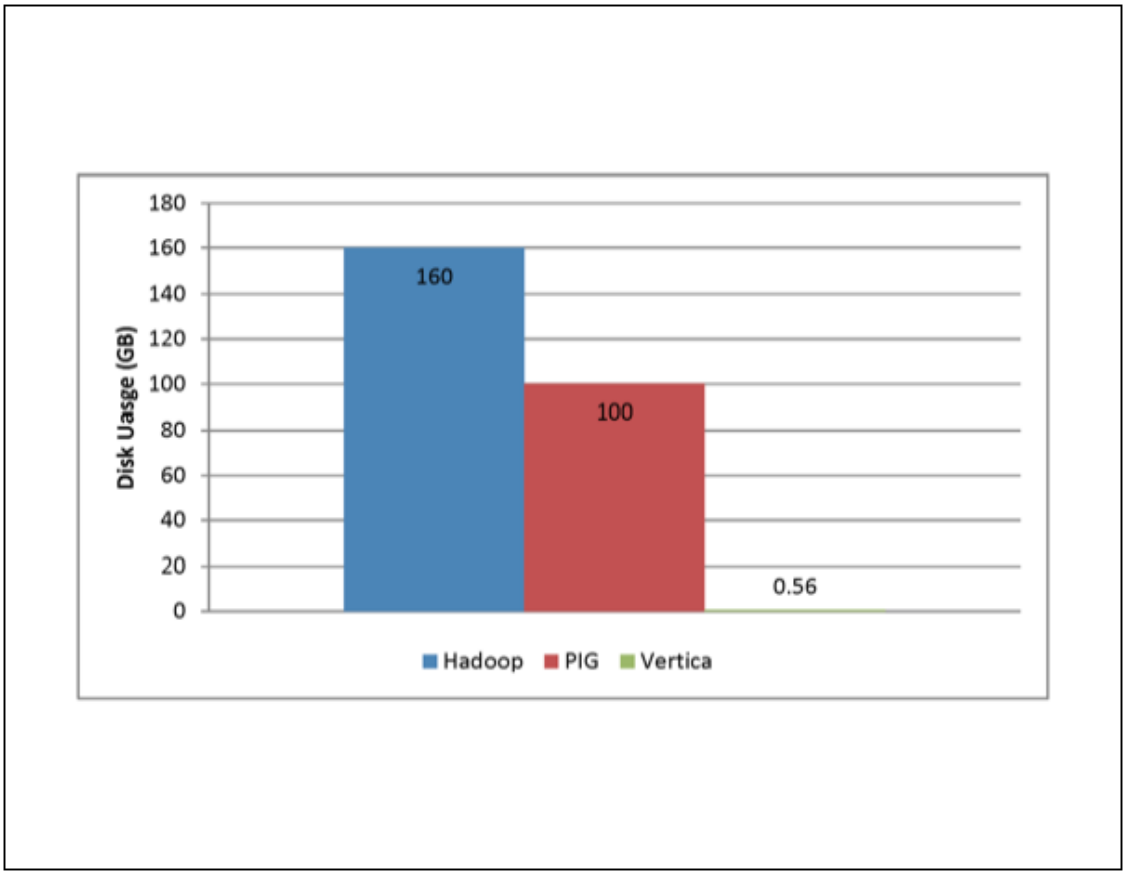
nodes to run the PIG job to equal the performance of just 48 Vertica nodes, which is a rack and a half of the Vertica appliance.

Finally consider what happens when the use case shifts from counting all of the triangles in a graph to counting (or listing) just the triangles that include a particular vertex. Vertica's projections (those things that define the physical storage layout) can be optimized such that looking up all of the edges with a particular vertex is essentially an index search (and once found the associated edges are co-located on disk – an important detail which anyone who knows the relative cost of a seek versus a scan will appreciate). This very quickly whittles e1 and e3 down to relatively few rows which can participate in a merge join with e2. All in all a relatively inexpensive operation. On the other hand PIG and Hadoop must process all of the edges to satisfy such a query.

Disk Usage

For the input data set of 1.3GB, it takes 560MB to store it in Vertica's compressed storage. In comparison, storing it in HDFS consumes more space than the raw data size.

At run-time, here is the peak disk usage among all 3 solutions in a 4-node cluster (remember lzo was not enabled for Hadoop and PIG – turning it on would reduce disk usage but likely hurt performance).



Given the huge differences in disk usage and thus I/O work, along with other advantages outlined above it should come as no surprise that the Vertica solution is much faster.

Join Optimization

As we mentioned earlier, the Hadoop solution does not optimize for join performance. Both Vertica and PIG were able to take advantage of a relatively small edges table that fit in memory (100s of billions or more edges can fit in memory when distributed over 10s or 100s of machines), with a hash join implementation.

For PIG, the join ordering needs to be explicitly specified. Getting this ordering wrong may carry a significant performance penalty. In our study, the PIG solution with the wrong join ordering is 1.5x slower. The penalty is likely even higher with a larger data set, where the extra disk I/O incurred in join processing can no longer be masked by sufficient RAM. To further complicate the matter, the optimal join ordering may depend on the input data set (e.g.

whether the input graph is dense or not). It is infeasible for users to manually tweak the join ordering before submitting each PIG job.

In comparison, the Vertica columnar optimizer takes care of join ordering as well as many other factors crucial to optimizing for the job run-time.

The Right Tool for the Job

Many people get significant value out of Hadoop and PIG, including a number of Vertica's customers who use these tools to work with unstructured or semi-structured data – typically before loading that data into Vertica. The question is which tool is best suited to solve your problem. With User Defined Functions, Aggregates, Load, et cetera available or coming soon to Vertica the lines are becoming blurred but when it comes to performance the choice is crystal clear.

In the case of triangle counting as we presented above, the Vertica solution enjoys the following advantages over Hadoop and PIG:

- Ease of programming and maintenance, in terms of both ensuring correctness (The Vertica SQL solution is simpler) and achieving high performance (The Vertica optimizer chooses the best execution plan)
- Compressed storage
- Orders of magnitude faster query performance

Do Try this at Home

It is a relatively safe experiment (unlike slicing a grape in half and putting it in the microwave – don't try that one at home). We've uploaded all three solutions to GitHub. Feel free to run your own experiments and improve on our work. As it stands the project includes a build.xml file which runs the Hadoop and PIG solutions in standalone mode – the project README file describes these targets and more in detail. With a little more work one can configure a Hadoop cluster and run the experiments in distributed mode, which is how we ran the experiments described above.

It's a little more difficult to run the tests if you are not currently a Vertica customer, but we do have a free trial version of the Vertica Analytics Platform software.

Acknowledgements

Many thanks to Rajat Venkatesh for writing the PIG script (though I already thanked him with a lunch) and Mingsheng Hong for his suggestions, ideas and edits.

Previous Post Next Post