« Visualizing Corporate America

A Brief Tour of Graphd

by Scott Meyer

[Many people ask us about the Freebase backend, so I asked Scott Meyer, who leads our team of graph developers, to talk a bit about "graphd", our in-house solution. This post is long and juicy; click through the "More" tag to read it all. — Skud]

Freebase.com is powered by a tuple store called graphd. Graphd is a C/Unix server which processes commands in a simple template-based query language.

Why Build?

While it has been suggested that a complete rewrite is called for[1] on general principle there were two specific requirements that caused us to build our own database:

1. Schema Last. A collaborative database must, of necessity, support the creation or modification of schema long after data has been entered. While the relational model is quite general, current implementations map tables more-or-less directly into btree-based storage. This structure yields optimal performance but renders applications quite brittle.
2. The conventional table-of-tuples implementation is problematic, even on a modern column store[2]. The starting point, a table of tuples and indexes with compound keys which are permutations of subject-predicate-object is well studied and subject to obvious limitations of index size and self-join performance. Attempting to optimize an existing relational store for this tuple access pattern, while possible, is burdened by compatibility with a relational model that is far more general than we need and a SQL interface in which it is difficult to say what we really mean.

Tuples

Graphd primitives (tuples) are identified by GUIDS which consist of a database id and a primitive id. In a database, primitive ids are assigned sequentially as primitives are written. For example, 9202a8c04000641f8000000000006567, is the guid which corresponds to the one known to you as "Arnold Schwarzenegger." The front part, 9202a8c04000641f8 is the database id and the back part, 6567, is the primitive id. As you might surmise based on the number of intervening zeros, we're quite ambitious. Each graphd primitive consists of:

left
: A guid, the feathered end of a relationship arrow.

right
: A guid, the pointy end of a relationship arrow.

type
: A guid, used in conjunction with left and right to specify the type of a relationship.

scope
: A guid, identifying the creator of a given primitive.

prev
: A guid, identifying the previous guid in this lineage.

value
: A string used to carry literal values, strings, numbers, dates, etc.

And a few other odds and ends.

Any or all of these may be null.

Once written, primitives are read only. Graphd is a log-structured or append-only store. To "modify" a primitive, for example by changing the value, you write a new primitive carrying the modification and use the prev to indicate that it replaces the "modified" primitive. To delete a primitive, you write a new primitive which marks the primitive you wish to delete as being deleted. Deleted or versioned primitives are weeded out during query execution.

In addition to many implementation advantages, a log-structured database makes it easy to run queries "as of" a certain date.

ACID

Graphd supports a subset of the traditional database ACID guarantees: it is optimized for collaborative wiki-style access. Using transactions to mediate an edit war is pointless in the extreme. What is important is capturing and preserving user input such that the truth can ultimately be found. We assume a long cycle of read/write interactions, and so provide no built-in read-write atomicity. Instead, we guarantee only that writes of connected subgraphs of primitives are atomic (and durable). In the sense that it never becomes visible to you, it may be the case that your write collides with someone else's, however, your input is always preserved, and can be found by browsing the modification history, and re-instated if you so desire.

Objects and Identities

With few exceptions (none visible to the user), primitives may be regarded as either nodes (things without a left or right) or as links which always have a left and usually a right. Nodes such as, …6567, are used to represent identities, and carry no other data. Links are used to represent properties of an identity, either a literal value, for example, "height", or a relationship, for example, "employs/employed by." So, looking at Arnold, we see a property named /people/person/height_meters with a value of 1.88. This is represented by a single tuple whose left is …6567, type is …4561f6b, and value is the UTF8 string "1.88".

A relationship is similar but has a right instead of a value. The property such as /type/object/type, which identifies an object as being an instance of a particular type, is represented by a the guid …c. That Arnold is typed as a person is indicated by a primitive whose left is …6567, type is …c, and right is …1237.

At the MQL level, a node and its associated links is regarded as an "object" with fields (properties) described by one or more types. Such objects map naturally into the dictionary based objects supported by dynamic languages such as Python and Perl.

While MQL only exposes nodes as identities, the notion of identity is so fundamental that graphd could reasonably be described as an "identity-oriented" database. Giving every piece of data a fixed identity, is radically different from the relational model which deals only with sets of values and leaves the notion of identity up to the application. Working with identities as a first-class notion is essential if schema is to be flexible. Long before we can agree on the exact shape of the data used to represent a person or a building, we can agree that individual people or buildings exist and that they have certain obvious attributes that we might want to record: height, address, builder, etc.

Queries

Let's ask graphd how tall Arnold is:

```
{
  "query" : {
    "/people/person/height_meters" : null,
    "id" : "/topic/en/arnold_schwarzenegger"
  }
}
```

That, of course, is how we express the question in MQL. MQL looks up the guids for "/people/person/height_meters" and "/topic/en/arnold_schwarzenegger" and produces the following (simplified) graphd query:

```
read
(guid=9202a8c04000641f8000000000006567 result=contents
  (<-left right=null result=(value)
    type=9202a8c04000641f8000000004561f6b)
)
```

The first thing to notice is that all notion of schema has been compiled away. Unlike a conventional rdbms which preserves the notion of type all the way through query evaluation (the row-source tree), we're asking graphd a question which is expressed entirely in a simple vocabulary of constraints on a universal primitive type. Graphd doesn't know anything about /people/person or any other type or even the general structure of types.

Like MQL and unlike SQL, the graph query language is template based: the syntax of the query parallels the structure of the desired result. Each parenthesized expression corresponds to a constrained set of primitives. Nested expressions are related to each other via one of the linkage fields. So, the outermost expression,

```
read
(guid=9202a8c04000641f8000000000006567
  …
)
```

specifies that we'd like to look at exactly one guid, …6567, a node representing Arnold Schwarzenegger. And the inner expression:

```
read
  (<-left right=null
    type=9202a8c04000641f8000000004561f6b)
```

is looking for any primitive (link) which refers to the primitive satisfying the outer expression with its left and has a type of …4561f6b.

It is not unusual for graph queries to have dozens of subclauses. The traditional alternative, expressing each subclause as a join condition: x1.left = y AND x2.left = y AND …, becomes quite opaque after a few dozen iterations.

The basis for the query evaluation is a nested-loop join: we take a candidate satisfying the outermost constraint (in this case there's only one) and then look for candidates matching the inner constraint, right=null type=…4561f6b, which which also relate to the candidate for the outer expression in the specified way (ie. with their left).

While sometimes advantageous, in cases where the entire result set is not needed, an unaided nested-loop join is going to be hopelessly slow. However, there are a variety of optimization techniques that we can use to reduce the size of the candidate set at each node in the query. For now, suffice it to say that such techniques work quite well.

Performance

Aside from being "fast enough" to do what we want, graphd's performance is a bit difficult to evaluate as our query language is currently much less expressive than the SPARql which is the interface assumed by tuple-store benchmarks such as LUBM. That being said, we recently demonstrated sustained throughput of about 200,000 simple queries per minute on a single AMD64 core. Simple queries are things like "show me all people who are authors with names containing 'herman'"

Graphd requires no manual tuning or configuration of indexes. There are no knobs.

On disk, our current graph of 121 million primitives consumes about 12 Gb. That figure includes all index storage.

References

[1] The End of an Architectural Era (It's Time for a Complete rewrite) Michael Stonebraker, Nabil Hachem, Pat Helland, VLDB 2007

[2] Scalable Semantic Web Data Management Using Vertical Partitioning Daniel Abadi, Adam Marcus, Samuel Madden, Kate Hollenbach, VLDB 2007

This entry was posted on Wednesday, April 9th, 2008 at 5:17 pm and is filed under Backend. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

One Response to "A Brief Tour of Graphd"

ian holsman Says:
April 9th, 2008 at 5:28 pm

any plans on open sourcing it? or even selling it as a commercial product?

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment