Graphs, Sheaves, RAM, CPU

Linas Vepstas

2 Sept 2020 draft version 0.1

Abstract

Using connectors-and-sheaves for knowledge representation has distinct implications for RAM usage and CPU cycles. This text sketches and compares memory and CPU usage for three types of systems: a naive graph database, the current OpenCog AtomSpace, which is an in-RAM metagraph database, and a hypothetical connector-based database. The design constraints are such that several different kinds of typical knowledge-representation tasks must be efficient and performant; these include graph traversal (pattern matching), graph rewriting, and parsing (rule application).

This is a work in progress ...

Introduction

Currently, graph databases are popular, and they have a rather distinct performance profile, differing from SQL and noSQL databases. The connectors-and-sheaves concepts sketched in this series of texts provide a different way of representing graphs, and working with them. This representation is not just abstract nonsense, but has actual implications for RAM and CPU consumption. The current prototype experiments with sheaves are layered on top of the OpenCog AtomSpace, which is an in-RAM database for storing generalized hypergraphs or "metagraphs". It has it's own distinct representation, with certain implications for CPU and RAM utilization.

The goal of this text is to look at these three systems, and compare how they use RAM and CPU resources. Describing the AtomSpace is the easiest, because the current implementation has a specific form. Describing a graph database is a bit harder, as we assume a generic, naive design, which actual implementations may or may not follow. Describing the connectionist database presents similar trouble: without a specific implementation and the experience from using it, statements have to be generalized.

Each of these is described in turn, followed by a discussion of algorithms.

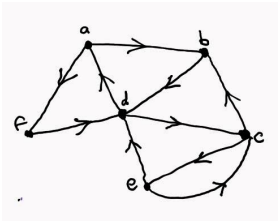
Graph Representations

Formally, a graph is

- A set of vertexes $V = \{v_1, v_2, \cdots, v_M\}$

- A set of edges $E = \{e_1, e_2, \cdots, e_N\}$ where each edge e_k is an ordered pair of vertexes drawn from the set V.

Because edges are ordered pairs, it is conventional to denote them with arrows, having a head and tail. These can be joined together in arbitrary ways. Below is a "typical" directed graph:



In practice, one wishes to associate a label to each vertex, and also some additional attribute data; likewise for the edges. At this time, the JSON format is a very powerful and flexible way of encoding attribute data: it can encode variable format structured data consisting of strings, numbers, arrays and other kinds of data. Thus, for the remainder of this text, it is assumed that attributes are encoded in a JSON format. There are a variety of ways of encoding JSON; this alone is a substantial task, and detailed discussions of this are mostly avoided, except as may be relevant.

In what follows, all data is assumed to live in-RAM; the on-disk representations do not concern us. One reason for this is that a variety of disk management systems exist, and work quite well at abstracting details. The earliest such is perhaps the Berkeley DBM, and the Gnu gdbm. These have been followed by Google's LevelDB and Facebook's extensions RocksDB. It is usually not too hard to take an in-RAM database, and layer it on top of one of these systems to obtain a disk-backed database. Of course, there are numerous ifs-ands-buts, here, these are glossed over.

The vertex table is straight-forward:

vertex id	json-data
1	...
2	...
3	...
...	

The goal of having a vertex id (which is necessarily a "universally unique id" or uid) is that it is required by the edge table. The edge table might have the form

edge id	head-vertex	tail-vertex	json-data
77	1	2	...
88	2	3	...
99	2	4	...

This representation is perhaps too naive. To perform a graph traversal, i.e. to walk from vertex to vertex, following only connecting edges, one needs to know which edges

come in, and which edges go out. Of course, these can be found in the edge table, but searching the edge table is absurd: for an edge table of N edges, such a search takes $O(N)$ time. Thus, we incorporate a special index for edges into the vertex table:

vertex id	incoming	outgoing	json-data
1	{}	{77}	...
2	{77}	{88,99}	...
3	{88}		...
4	{88}		

Note that the incoming and outgoing columns hold sets: any given vertex may appear in more than one edge. They are sets, not lists, as the order is not particularly important. They are not "multisets": any given edge appears at most once in the incoming/outgoing sets. Suitable representations for sets include hash-tables and trees, each with it's own distinct RAM and access-time profile.

A conventional requirement for graph databases is to locate all nodes and vertexes having some particular attribute. This opens a Pandora's box of indexing schemes. The opening of this box is deferred to a later section, but we can take a quick peak: suppose one wants to find all vertexes where the json-data has a field called "favorite song". Vertexes representing buildings and automobiles won't have a "favorite song", vertexes representing people might, but not necessarily. Thus, we need to create an index: a set of all vertexes that have this tag. Every time a vertex is added or removed, this index might have to be updated. Thus, adding indexes in this way incurs a CPU overhead. If there are J indexes, then there is an $O(J)$ CPU overhead for vertex insertion/removal. There is also RAM consumption: an index containing K items requires at least $O(K)$ storage, and possibly $O(K \log K)$.

Hypergraphs

A hypergraph is much like a graph, except that the edges, now called "hyperedges" can contain more than two vertexes. That is, the hyperedge, rather than being an ordered pair of vertexes, is an ordered list of vertexes. The metagraph takes the hypergraph concept one step further: the hyperedge may also contain other hyperedges. A change of terminology is useful: the basic objects are now called "nodes" and "links" instead of "vertexes" and "edges".

Formally, a hypergraph is:

- A set of vertexes $V = \{v_1, v_2, \cdots, v_M\}$
- A set of hyperedges $E = \{e_1, e_2, \cdots, e_N\}$ where each hyperedge e_k is an ordered list of vertexes drawn from the set V. This list may be empty, or have one, or two, or more members.

A metagraph is very nearly the same:

- A set of nodes $V = \{v_1, v_2, \cdots, v_M\}$

- A set of links $E = \{e_1, e_2, \cdots, e_N\}$ where each hyperedge e_k is an ordered list of nodes, or other links. It is convenient to call these "atoms": an atom can be either a node, or a link. Links are thus sets of atoms.

Hypergraph representations

The naive representation for the hypergraph is a straight-forward extension of the edge table. The example encoded in this table is shown in the figure below.

edge id	vertex-list	json-data
e_1	(v_1)	...
e_2	(v_1, v_2)	...
e_3	(v_3, v_4)	...
e_4	(v_3, v_2, v_1)	

The vertex list may be empty, may hold one, or more vertexes. It is necessarily ordered (and thus not a set) and may contain repeated entries (a vertex may appear more than once). In other respects, it is much the same.

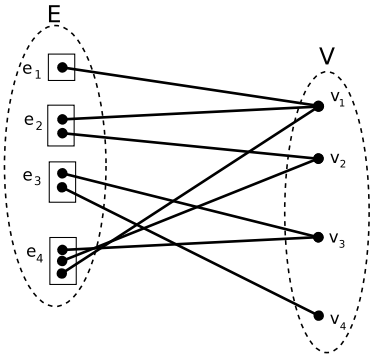
As before, the ability to traverse the hypergraph is a hard requirement. This requires modification to the vertex table. Several choices are possible. One is to add a new column for each positional location:

vertex id	edge-set-0	edge-set-1	edge-set-2	...	json-data
v_1	$\{e_1, e_2\}$	$\{\cdot\}$	$\{e_4\}$		...
v_2	$\{\cdot\}$	$\{e_2, e_4\}$	$\{\cdot\}$		...
v_3	$\{e_3, e_4\}$	$\{\cdot\}$	$\{\cdot\}$		...
v_4	$\{\cdot\}$	$\{e_2\}$	$\{\cdot\}$		

This requires a data-structure that is a list-of-sets, which can be a bit overcomplex and challenging to use. It is easier to just mash all of these into one set; this is all that is needed for hypergraph traversal. If the positional location is needed, then it can always be looked up per-edge. This is neither technically challenging nor CPU-intensive: the arity of hyperedges is typically small, based on real-world mappings to interesting datasets. Thus, the vertex table can look like

vertex id	incoming-set	json-data
v_1	$\{e_1, e_2, e_4\}$	...
v_2	$\{e_2, e_4\}$	...
v_3	$\{e_3, e_4\}$	...
v_4	$\{e_2\}$	

Note that the vertex table looks a lot like the edge table, the only difference being that the vertex-list is an ordered list, while the incoming-set (the edge-set) really is a set. Effectively, this is because a hypergraph is "almost" a bipartite graph, having the form below, with the set U on the left being the set of hyperedges.



The boxes denote the fact that the hyperedges are ordered lists. The E and V ellipses are the hyperedge and vertex tables.

RAM Utilization

One might wish to conclude: "Oh, but a bipartite graph is just a graph, so a graph database is sufficient for all my needs." At some abstract level, this is perhaps true; at the CPU and RAM-consumption level, it is not. So, in this figure, attributed (the json-data) are attached only to the v_k and e_k in the diagram; there is no attribute data attached to the lines in this figure. What's more, the lines in this figure are not directly recorded in any tables; they are implicit only by the structure of the vertex and hyperedge tables.

Counting the memory usage is instructive. Lets assume that the size of the vertex-id and the edge-id are the same – they are pointers or 64-bit ints – so each id requires 1 unit of RAM. Assume that lists are either null-terminated or record a length, so that a list of n items requires $n+1$ units of storage. Lets encode sets as lists, to make counting easy; let J is the average size of the attribute collection. The hyperedges shown in the figure then require 2+3+3+4=12 units of storage, plus 5 more for the hyperedge table itself, and 4J of attribute storage. The vertexes require 4+3+3+2 units of storage, plus 5 for the vertex-table itself, plus 4J more of attributes. Summing this, one obtains 34+8J total RAM consumption. For the general case, one has

$$N_V\,(1+\langle J\rangle+\langle N_I\rangle)+N_E\,(1+\langle J\rangle+\langle N_O\rangle)$$

where

N_V	Number of vertexes
N_E	Number of hyperedges
$\langle J\rangle$	Average size of attributes
$\langle N_I\rangle$	Average size of the incoming set
$\langle N_O\rangle$	Average size of the vertex list

The average size of the incoming set is equal to the average size of the vertex list, so we can approximate $\langle N_I\rangle = \langle N_O\rangle$; the only reason to track these separately is that one may use hash tables or trees for sets, whereas lists require arrays.

The equivalent representation as a graph requires

$$\begin{aligned} (N_V + N_E)(1 + \langle J \rangle) + N_V \langle N_I \rangle + N_E \langle N_O \rangle \quad & \text{for the vertex table} \\ N_V \langle N_I \rangle (3 + \langle J_{nil} \rangle) \quad & \text{for the edge table} \end{aligned}$$

Comparing the two expressions, we see that the vertex table is the same size as the complete hypertable. The ordinary graph representation also requires the overhead of the edge table; here the $\langle J_{nil} \rangle$ is the cost of storing an empty attribute list, and the factor of 3 comes from storing an ordinary edge-id and it's two endpoints.

Storing hypergraphs in graphs

If the only thing that one is storing are hypergraphs, then having a custom hypergraph representation really is smaller than the equivalent bipartite graph: it dispenses with the need for an explicit ordinary-edge table. Does this mean that there's some magic, here? No, not really. Every ordinary graph is a special case of a hypergraph, where the hyper-edge always has arity two. To store a single edge as an ordinary edge, we need $3 + \langle J \rangle$ units of storage: the edge-label, and the two vertexes in the edge. To store a single edge as a hyperedge, we need $4 + \langle J \rangle$ units of storage: the edge-label, the list of vertexes, and the list terminator. Thus, storing an ordinary graph as a hypergraph requires N_E more units of storage. This seems tolerable: for a million-edge graph, and 64-bit pointers, this requires 8MBytes of additional storage. On modern machines, the extra 8MBytes seems not all that large; there's a bit of a penalty in moving from graph storage to hypergraph storage, but it's not that much. Modern cellphones have 8GBytes of RAM...

Moving in the opposite direction is much worse: the penalty is $N_V \langle N_I \rangle (3 + \langle J_{nil} \rangle)$ which is surprisingly large. Assuming that $\langle J_{nil} \rangle = 1$, then a million-vertex graph requires $\langle N_I \rangle$ times 32MBytes of additional storage. For uniformly-distributed graphs, one might have $\langle N_I \rangle$ of 3 to 10; for scale-free graphs of this size, $\langle N_I \rangle$ might be around 14; for square-root-Zipfian graphs (such as Wikipedia topic connectivity, or biological datasets: genome, proteome, reactome datasets) the $\langle N_I \rangle$ would be around 2000[1], so we are looking at overheads in the (multi-)gigabyte range. There is a huge cost of jamming a hypergraph into an ordinary graph store.

[1]The average size of the incoming set is

$$\langle N_I \rangle = \frac{1}{N_V} \int_1^{N_V} n(v)\, dv$$

where $n(v)$ is the number of connections to vertex v and N_V is the total number of vertexes. For a Zipfian distribution, this is

$$\langle N_I \rangle = \frac{1}{N_V} \int_1^{N_V} \frac{N_V}{v} dv = \log N_V$$

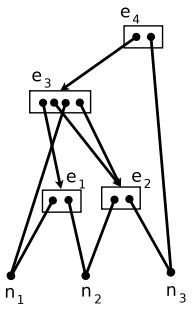
while for a square-root-Zipfian, one has

$$\langle N_I \rangle = \frac{1}{N_V} \int_1^{N_V} \frac{N_V}{\sqrt{v}} dv = 2\sqrt{N_V}$$

This last has particularly dramatic implications for graph stores.

Metagraph representations

The metagraph differs from the hypergraph in that now a hyperedge (link) may contain either another vertex (node) or another link. Visually, this is no longer a bipartite graph, but a polytree, shown below.



The polytree is more-or-less a directed acyclic graph (DAG), the primary difference being that the links are ordered lists, represented as boxes in this diagram.

The AtomSpace

The AtomSpace is an in-RAM generalized hypergraph or “metagraph” database.