Graphs, Sheaves, RAM, CPU

Linas Vepstas

2 Sept 2020 draft version 0.2

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.

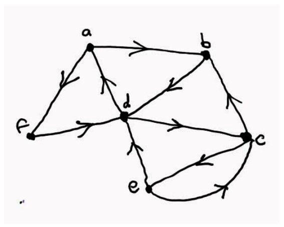
Graph Representations

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-and-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.

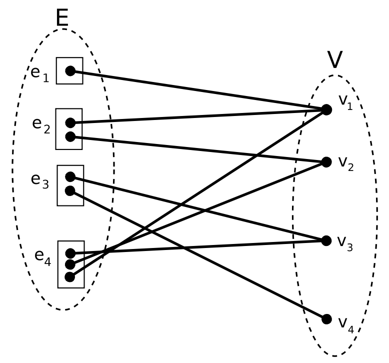
As before, the ability to traverse the hypergraph is a hard requirement. This requires modification to the vertex table. Several choices are possible. One is to add a new column for each positional location:

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 over-complex 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

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

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, and vice-versa

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 page views, or biological datasets: genome, proteome, reactome datasets) the $\langle N_I\rangle$ would be around 200[1], so

[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

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

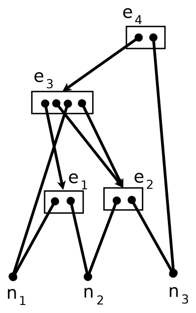
The scale factor A is data-dependent. For Wikipedia page views, $A\approx 20$, see https://en.wikipedia.org/wiki/Wikipedia:Does_Wikipedia_traffic_obey_Zipf%27s_law%3F for graphs and discussion. For genomics, see https://github.com/linas/biome-distribution/blob/master/paper/biome-distributions.pdf where A is in the range of 0.1 to 0.3, depending on the dataset. These last estimates are a bit glib, as the specifics of the datasets are quite subtle. Still, one may conclude that these considerations have quite dramatic implications for graph stores.

we are looking at overheads in the gigabyte range. There is a huge cost of jamming a hypergraph into an ordinary graph store.

XXX TODO This is making some rather strong claims about RAM usage, and really needs to be quadruple-checked and strengthened. It's a bit breezy and casual, as written. XXX TODO.

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, such as the one 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 node table is effectively the same as the vertex table for the hypergraph, before. For this graph, it is

node id	incoming-set	json-data
n_1	$\{e_1, e_3\}$	...
n_2	$\{e_1, e_2\}$	...
n_3	$\{e_2, e_4\}$	...

The link table now requires both an outgoing-atom list, and an incoming-link list.

link id	outgoing-list	incoming-set	json-data
e_1	(n_1, n_2)	$\{e_3\}$	...
e_2	(n_1, n_3)	$\{e_3\}$	...
e_3	(e_1, e_2, n_1, e_2)	$\{e_4\}$	...
e_4	(e_3, n_3)	$\{\cdot\}$	

Notice how this link-table resembles the vertex-table of an ordinary graph store: it has columns for both incoming and outgoing "sets"; the outgoing-set, however, is not a set but an ordered list. In terms of designing storage, the naive graph tables, the hypergraph tables, and the metagraph tables seem to have much in common. This is, however, perhaps a bit deceptive, as performance considerations dictate the finer aspects of the design.

Clearly, the metagraph, having the general shape of a DAG, can be wedged into an ordinary graph store. Conversely, an ordinary graph is merely a metagraph that never goes more than one-level deep, and whose links always have arity-two. Either format is adequate for representing the other. The metagraph, much like the hypergraph, has no need to explicitly declare the arrows in the tree; they are not stored, nor do they have attributes. The RAM-usage considerations are much like those for the hypergraph. More interesting is the structure of indexes, or rather, the alternatives one has for index representation.

Indexing

The whole point of using a graph database, as opposed to an SQL or noSQL database, is that the graph structure encodes something important about the problem, something that cannot be easily achieved by doing table joins or key-value look-ups.[2] However, just as with table-based databases, there are certain types of queries that are used a lot, and speeding these up through indexing is a key ability. How might this work, in practice? Let's examine some queries, and see how they might work.

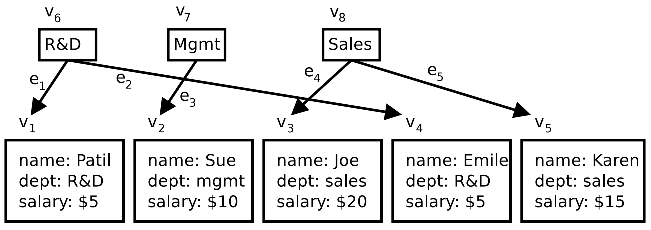
Single attribute queries

Suppose one wishes to find all nodes with some specific attribute. Naively, this requires walking over all nodes, and then examining the attached attribute structure (presumed to be JSON-like in form) and then extracting a named field from the attributes, and examining the value of that field. This is a task that SQL databases excel at - for example, "*SELECT name,salary FROM employees WHERE department='sales';*". A graph database is not needed for this task. Nonetheless, this is a plausible task. The traditional solution would be "CREATE INDEX ON employees(department);" which results in the creation of ordered pairs $(D,\{R\})$ with D the name of the department, and $\{R\}$ the set of all records having that value. The *SELECT* is then straight-forward: it need only return $\{R\}$. Note that the size of this index is $O(N)$ where N is the number of employees. This is necessarily so: one cannot build an index smaller than the number of employees: every employee must be in some department, as, conventionally, table-driven databases don't have null entries in rows. (Well, in practice, they often do; but now imagine the task of finding all records with a null value in some column...) Table-based information also has some representational difficulties: imagine the case of an employee with dashed-line reporting to multiple departments.

How might one accomplish indexing like this in a graph database? The simplest, most naive answer is to create new, "privileged" vertexes, one vertex per department. They are "privileged", in that the associated attributes record one and only one value: the name of the department. Basically, the vertexes are labeled, thus escaping the overhead of crawling through a collection of attributes to find one in particular. One

[2]In the following, only SQL databases will be discussed. The noSQL databases are effectively identical, from our perspective, as they are categorical opposites: that is, all arrows reversed. This was explicitly articulated in a famous paper by Meijer and Bierman.[1] Thus, in the discussions below, if you are more familiar with noSQL, then simply reverse the directions of all arrows to obtain the equivalent discussion.

also created an unlabeled, attribute-free edge, from the department name back to the full employee record. Finding all employees in "*sales*" is now trivial: one can trivially find the vertex "*sales*", and trace all edges to the full record. The contents of the graph database, after indexing, is illustrated below. Before indexing, the vertexes v_6, v_7, v_8 and edges e_1, e_2, e_3, e_4, e_5 simply did not exist.



The size of this structure is again $O(N)$ for N employees, assuming every employee is indexed. It has some rather unusual properties: not every employee has to be indexed! It is possible to create only one vertex, "*sales*", and hook up edges to only that one. Effectively, one has a partial-index, with correspondingly less RAM usage! Of course, with some cleverness, an experienced DBA can achieve the same effect: "CREATE INDEX ON employees(department) WHERE department=sales;" and this is not a big deal, so, here, at least, graphs do not offer any particular advantage. Under the covers, the SQL databases effectively has more-or-less the same format, although their graph-based nature is *ad hoc*, as there are no explicit graph-walking directives in SQL.

The key point here is that, in a properly-designed graph database, there is no generic need for "indexes" *per se*, they can be conjured into being as they are ultimately graphical in nature. In the graph database, the graph structure of the index is explicit, and can be walked.

Space and Time

Comparing RAM-usage, at first glance, there is no particular difference. Naively, both require $O(N)$ for N employees, plus $O(M)$ for M different departments. Looking more carefully, there are also the edges e_1, e_2, e_3, e_4, e_5. In the SQL case, these edges were implicit in the index: after all, the index was a collection of ordered pairs $(D, \{R\})$: the edges run from D to $\{R\}$. In the graph representation, these edges become explicit: that is, they appear in an explicit table, with attached attributes, even if the attributes are null. Shades of hypergraphs! Why, this was exactly the *same* situation as with the hypergraph! Squinting more carefully, the indexed employee table is nothing other than a bipartite graph! Thus, one can effectively say: the indexes in an SQL database are *de facto* hypergraphs under the covers, even though no one ever explicitly says so. The bipartite nature of the graph makes this explicit. Surprise!

CPU usage considerations are harder to dissect. To avoid discussions of network overhead in client-server architectures, its easier, here, to limit discussions to databases that run in the same address space as the application. Thus, for SQL bench-marking, one might look at SQLite, which runs embedded, rather than Postgres, which requires network interfaces. Queries usually begin life as strings, for example, "*SELECT name,salary FROM employees WHERE department='sales';*" was a string that had to be parsed to figure out "what to do". Let's assume that this cost has been amortized, and that there is a way to get a handle to a query that has already been analyzed. Query run-time execution is then a matter of finding the vertex "*sales*", tracing the edges to each of the employees, and completing the work by analyzing each employee. If vertexes themselves are indexed (as they should be), then locating the vertex "*sales*" is either $O(1)$ for hash tables, or $O(\log V)$ for trees. In the hypergraph representation, finding the set $\{R\}$ of employees in "*sales*" comes for free. The dominant cost is almost surely the analysis needed to extract the desired information from the record attributes.

Partial indexes

The power of partial indexes together with metagraphs begins to reveal itself when one considers mapping an organizational chart onto the above example. Conventionally, corporations, political and military organizations are organized hierarchically, with divisions reporting to executives, departments rolling up into divisions, and so on. This in the form of a DAG, and, in complex organizations, the DAG has the form of a polytree, as illustrated for the metagraph data structure.

The conceptual jump here is then: rather than stopping with a single-level hypergraph, where there are "tables", and then there are "indexes" that are "on top of tables", one can go further: indexes of indexes: namely the polytree or metagraph.

The implication for storage is similar to that of "database normalization". In a naive table format, one might store, for each employee, the employee name, the department, the 2nd line, the division and the name of the company, the latter being needed if one was handling multiple companies. This is a bit silly in terms of storage: 5 columns are needed; for N employees, this requires $O(5N)$ storage. One "normalizes" by storing only the employee-department relationship with a table of $O(2N)$ in size, and the remainder of the org chart in a table, also of two columns, showing the reporting structure. This offers a huge space savings. Look-ups proceed through table joins: to find all employees in a division, one looks up what 2nd lines report to the division, what departments report to the second line, and what employees report to the departments. The indexing proceeds just as described above, and the table joins are an *ad hoc* graph walk. The SQL for this is a bit nasty, but still effectively human-readable: "*SELECT employees.name FROM employees, orgchart WHERE employees.department=orgchart.dept AND orgchart.division='marketing & sales';*". To be correct, this example is oversimplified by quite a bit, but it does convey the general spirit of the thing. It is attempting to specify a graph-walk without explicitly acknowledging that there is a graph hidden under the covers.

The key message here is that metagraphs retain the key benefits of table normalization, while making the graphical nature of indexing explicit. This is an actual improvement over graph stores: by discarding the edge table, and the associated edge attributes, one gets representational compactness of indexes, without paying a high price for them.

Queries and pattern matching

Key points:

- polytrees can be thought of as either s-expressions or as json.
- Walks are explicit: one moves up or down the trees e.g. the car, cdr of lisp, but also the tinkerpop/gremlin primitives for graph-walking.
- Tree walks are a prime example of a recursive algorithm, requiring a stack machine to execute, and describable by a context-free grammar.
- This means tree walks can be automated. Rather than designing a search query where the user explicitly states the walk/path to be performed (tinkerpop/gremlin), a context-free grammar can be used to describe the search, which is then executed by the stack machine. This is what "pattern matching" actually is.

Connectors and Sheaves

Only now do we get to the main act...

- Specifying (context-free) grammars in a natural way.
- Parsing as jigsaw-puzzle assembly
- RAM and CPU costs of storing jigsaw pieces

References

[1] Erik Meijer and Gavin Bierman. A co-relational model of data for large shared data banks. *ACM Databases*, 9, 2011.