Graphs, Metagraphs, RAM, CPU

Linas Vepstas

2 Sept 2020 Version 1.0

Abstract

This text reviews the concepts of a graph store, starting from the fundamental question of how to efficiently represent a graph in RAM (that is, in storage). Starting with a naive conception of a graph database, it arrives at hypergraphs and metagraphs as simpler and more efficient representations for graphical data.

Even stronger claims are made: metagraph databases are easier to use than graph databases. The structure and contents of a metagraph database is easier to understand. Algorithms that act on a metagraph database are simpler, more compact, and easier to write, than corresponding algorithms for graph databases. This is really quite remarkable, and seems to come "for free" with just a minor change in perspective.

Once one arrives at the concept of a metagraph (and arrives at it quite naturally, through minor modifications of an ordinary graph store), there is then a rather remarkable and easy slide down-hill that culminates at the OpenCog AtomSpace and Atomese as a (near-)optimal design point. The metagraph representation is remarkably flexible and powerful tool for representing data (representing knowledge) and working with (manipulating, rewriting) that data.

This is part of a sequence of texts on sheaves, although it does not explicitly mention these. It is a mandatory pre-requisite for understanding the further efficiencies and flexibility that sheaves provide.

Introduction

Currently, graph databases are popular, and they have a rather distinct performance profile, differing from SQL and noSQL databases. At a simplistic level, the OpenCog AtomSpace is a kind of a graph database. More correctly, it is a generalized-hypergraph or "metagraph" database. This design has certain implications for RAM and CPU usage. This text argues that it has superior properties to ordinary graph databases. It arrives at this conclusion by starting with the most basic, foundational description of graph databases, and then defines hypergraphs and metagraphs as minor variants on the underlying data structures.

This text is organized into several parts:

- A description of graphs, and how they can be represented in memory,
- Modifications to the representation that result in hypergraphs and metagraphs, including a quick analysis for memory usage

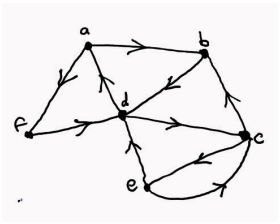
- A discussion of the concept of "indexing", especially in how it contrasts with the concept as generally understood in SQL/noSQL databases,
- A quick introduction to partial indexes (a full discussion is impossible without first tackling pattern matching *aka* querying, done in a later text).
- A short review of how meta-trees are used to represent knowledge.

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:



Attributes

In practice, one wishes to associate a label to each vertex, and also some additional attribute data; likewise for the edges. There are two fundamental choices available for storing attributes: merged schema+data, and disjoint schema–data. An example of the first is JSON. Each block of data to be stored is preceded by its name. Additional markup, such as quotes and square brackets, indicate structure such as text-strings and arrays. An example of disjoint schema–data are tables. The name and data type appears only in the column heading; individual rows in the table do not need to repeat the schema. Clearly, the tabular format offers a huge advantage in terms of memory usage. Conversely, tables are highly inflexible when new columns or new schema need to be added. There is no sensible way to take one row of a table, and have it use a different schema than the other rows. It doesn't even make sense to talk about rows in this way; they aren't rows any more.

Thus, in a graph store, one has these two choices for storing attributes, both for vertexes and for edges. One might even contemplate a mixture of both; after all, a JSON blob is isomorphic to a table with only a single row. The remainder of this text will make little or no assumptions about the storage format of the attributes. The details of attribute handling has little or no impact on the primary topics here, with possibly one exception: indexing. This is reviewed in a distinct section later on.

RAM vs. Disk

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, which provide motivations to roll-your-own; these will be ignored in this text.

Representing graphs in RAM

Storing a set of vertexes in RAM is straight-forward. Since it is a set, one can use either a hash-table, a b-tree, or even an array or list. For the discussion here, the precise format is not directly relevant, and so a tabular format will be used to illustrate the ideas. Again, the table rows might actually reside in hash-tables, b-trees, or wherever.

The vertex table is straight-forward:

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

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

edge id	head-vertex	tail-vertex	attr-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, it is natural to incorporate a special index for edges into the vertex table:

vertex id	incoming	outgoing	attr-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.

Prelude to indexing

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 attr-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, there is a need for 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, or a mixture. They are arranged to be acyclic (to form a directed acyclic graph).

It is convenient to give the name "atoms" to something that is 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 table below provides an example that is shown in the following figure.

hyper-edge id	vertex-list	attr-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 the list). In other respects, this edge table is quite similar to the edge table for ordinary graphs.

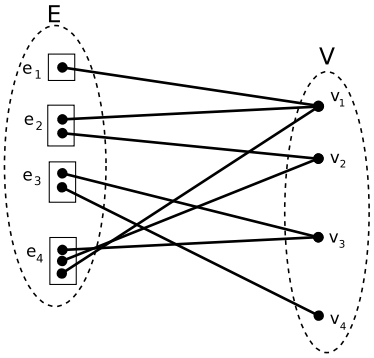
As before, the ability to traverse the hypergraph is a hard requirement. This requires modification to the vertex table. Several choices are possible. One is to add a new column for each positional location:

vertex id	edge-set-0	edge-set-1	edge-set-2	...	attr-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-hyperedge. This is neither technically challenging nor CPU-intensive: the arity of hyperedges is typically small, based on real-world mappings with interesting datasets. Thus, the vertex table can look like

vertex id	incoming-set	attr-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 E 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, attributes (the attr-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 in 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 $\langle J \rangle$ be 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 $4\langle J \rangle$ of attribute storage. The vertexes require 4+3+3+2 units of storage, plus 5 for the vertex-table itself, plus $4\langle J \rangle$ more of attributes. Summing this, one obtains $34 + 8\langle J \rangle$ 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. This makes the RAM usage slightly different for the two.

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 this to the expression for the hypertable, we see that the vertex table is the same size as the entire 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. Graph databases can store hypergraphs, but incur a RAM penalty for doing so.

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 hyperedge always has arity two. To use an ordinary graph store to record a single edge, we need $3 + \langle J \rangle$ units of storage: the edge-label, and the two vertexes in the edge. To use a hyperedge store to record a single (ordinary) edge, 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

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

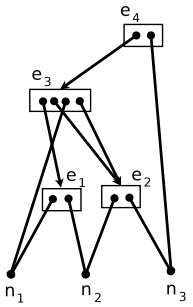
The scale factor A is data-dependent. For Wikipedia page views, $A \approx 20$, see https://en.wikipedia.org/wiki/Wikipedia:Does_Wikipedia_traffic_obey_Zipf%27s_law%3F for graphs and discussion. For genomics, see https://github.com/linas/biome-distribution/blob/master/paper/biome-distributions.pdf where A is in the range of 0.1 to 0.3, depending on the dataset. These last estimates are a

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 has the shape of a directed acyclic graph (DAG), such as the one shown below.



The primary difference between the above, and a DAG is that the links are ordered lists, represented as boxes in this diagram. For lack of a better name, this can be called a "metatree". The metatree can be converted to a DAG in two different ways. One way is to collapse the boxes to single points. The other way is to dissolve the boxes entirely, and replace a single arrow from point-to-box by many arrows, from point to each of the box elements.

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

node id	incoming-set	attr-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 set.

link id	outgoing-list	incoming-set	attr-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)	$\{\cdot\}$	

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.

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. We can conclude that it is quite efficient to store a graph in a metagraph, but that storing a metagraph in an ordinary graph database pays a large penalty.

Indexing

More interesting is the structure of indexes, or rather, the alternatives one has for index representation. The whole point of using a graph database, as opposed to an SQL or key-value 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 will be 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

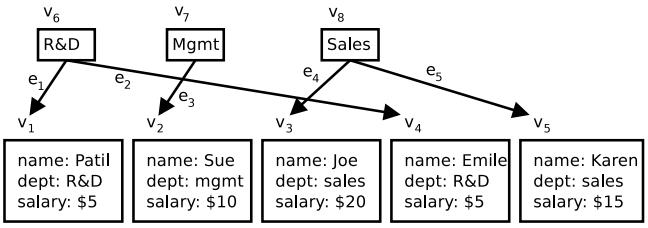
Suppose one wishes to find all nodes with some specific attribute. Naively, this requires walking over all nodes, examining the attached attribute structure, extracting a named field from the attributes, and examining the value of that field. This is a task that SQL databases excel at - for example, "*SELECT name,salary FROM employees WHERE department='sales';*".[3] A graph database is not needed for this task. Nonetheless, this is a plausible task, even for a graph database. 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

[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 key-value databases, then simply reverse the directions of all arrows to obtain the equivalent discussion.

[3] The author would like to apologize for this seemingly non-sexy example. It is the stereotypical example from database textbooks, and harks back to the 1960's, when working out the fine details of management science actually was a sexy research topic, and helped power the economic ascent of the Western world. It's importance should not be under-estimated: Ancient Rome was an agrarian civilization built on concepts of hierarchical organization; organizational hierarchies will continue to describe reality, including AGI. Org-charts are boring but important.

all records having that value. The `SELECT` is then straight-forward: given D, 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 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 act of indexing creates these vertexes and edges.



The size of this structure is again $O(N)$ for N employees, assuming every employee is indexed. More precisely, it is $O(N) + O(N_D)$ where N_D is the number of departments.

This diagram exposes some unusual possibilities: 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, other than perhaps some conceptual clarity. Under the covers, the SQL databases effectively have 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 at any time, as they are ultimately graphical in nature. There's even a bit of an advantage: 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 between the SQL and the graph database. Naively, both require $O(N)$ for N employees, plus $O(N_D)$ for N_D 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!

The explicit hypergraph representation does cost more than SQL. An SQL index can be a b-tree or hash table; the only thing that the b-tree/hash table needs to store is the row ID. For a hypergraph, we have imposed the additional requirement that hypergraphs must be rapidly traversible. This forces the storage of the incoming set in addition to the outgoing set. Hypergraph stores necessarily use more RAM than equivalent SQL tables. But recall why we did this: rapid graph traversal. Graph traversal in SQL is easy for trivial graphs, but becomes profoundly challenging for anything more complex. Mashing up "*SELECT INTO*" with "*JOIN*" is tough.

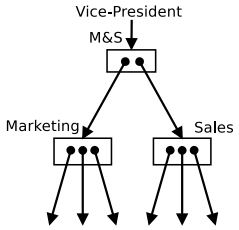
CPU usage considerations are harder to dissect. To avoid discussions of network overhead in client-server architectures, its easier, here, to limit discussions to databases that run in the same address space as the application. Thus, for SQL bench-marking, one might look at SQLite, which runs embedded, rather than Postgres, which requires network interfaces. Queries usually begin life as text-strings, for example, "*SELECT name,salary FROM employees WHERE department='sales';*" was a text-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 extracting fields for each employee. If vertexes themselves are indexed (as they should be), then locating the vertex "*sales*" is either $O(1)$ for hash tables, or $O(\log N_D)$ 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 and metagraphs

The power of partial indexes together with metagraphs begins to reveal itself when one considers query and search query optimization. This text is long enough, so this will be deferred to a later chapter.

Partial indexes reveal their utility in another way. Sticking with the management example from above, consider extending and looking at organizational structures (org charts).

Conventionally, corporations, political and military organizations are organized hierarchically, with divisions reporting to executives, departments rolling up into divisions, and so on. This is precisely the structure of a metatree. It is tempting to gloss this, and say that the org chart is a tree, or perhaps a DAG. It is not! It is a metatree, and confusion arises because a metatree can be collapsed to a DAG in several different ways. So, consider a division chief who manages a line item. One can draw the org-chart several ways: by drawing an edge from a manager to each (named) employee that they manage, or from the manager to a functional box labeled with the function. Employees are then grouped inside these functional boxes. This is shown below.



This is manifestly in the shape of a metatree. It can be collapsed down to an ordinary directed tree in several ways, left to the imagination of the reader. The point is that the natural structure of an org chart is not a naive tree; it contains a bit more complexity than that, and is far more readily represented with a metatree.[4]

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

Normalization

The implication for RAM usage is similar to that of "database normalization". In a naive, un-normalized table format, one might store, for each employee, the employee name, the department, the 2nd line, the division and the name of the company. 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 separate table, also of two columns, encoding the directed tree reporting structure. This offers a huge space savings. For N_D departments/divisions, this second table is $O(2N_D)$ and clearly, $O(5N) \gg O(2N) + O(2N_D)$.

Look-ups in a normalized database 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 before. 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 =*

[4]This is hardly the only way to represent an org-chart with a meta-tree. One could put the department titles into boxes of their own, as well as perhaps the names of the actual people, adding even some dashed-line cross-functional reporting structures. The point here is that it is not "just a tree".

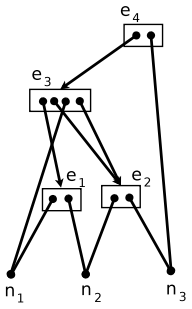
orgchart.dept AND orgchart.division = 'marketing & sales';".[5] This SQL snippet is oversimplified by quite a bit, but it does convey the general spirit of the thing. It is attempting to specify a graph-walk without explicitly acknowledging that there is a graph hidden under the covers.

The key message here is that metagraphs retain the key benefits of table normalization, while making the graphical nature of indexing explicit. They do even more: they effectively "automate" table normalization. To some fairly large degree, you no longer have to explicitly think about table normalization. It "just happens naturally", under the covers. This is not because there is some super-clever algo running under the covers, performing magic normalization. It is instead purely a byproduct of changing ones perspective about data.

Comparing metagraphs to graph stores, one sees a different improvement. By discarding the edge table (that the graph store demands), and the associated edge attributes, one gets the representational compactness of indexes, without paying a high price for them. The price one does pay (the incoming set) enables something quite dramatic: an easy graph walk, which is anything-but-easy in a traditional SQL database.[6]

Metatrees and String Representations

The proper way of representing a meta-tree as a text string can be mildly confusing. Articulating this carefully is rewarding. Consider again the generic example:



This can be written as $(e_4 : (e_3 : (e_1 : n_1, n_2), (e_2 : n_2, n_3), n_1, (e_2 : n_2, n_3)), n_3)$ or perhaps, with indentation, but without parenthesis, so as to improve readability (so,

[5] Its nasty, because we have to "join" different rows in the org chart table. SQL does not offer any basic primitives for joining different rows together; this requires a good bit of creativity on the part of the DBA.

[6] OK, sure, it becomes "easy" if you are willing to write PL/SQL, or, if your database supports it, then embedded Python. Otherwise, you have to descend into C/C++ (or your other favorite programming language of choice), and once you are "programming", it is no longer "easy". A properly designed graph query system makes graph walks "easy". And, to take one quick pot-shot: GraphQL is not properly designed. It is effectively a query anti-pattern. It took what is nice about SQL, but then utterly failed to take into account anything and everything that this text is trying to explain. It is not for nothing that the OpenCog AtomSpace differs so dramatically from everything else out there.

Python-style, *i.e.* with "significant whitespace"):

$$
\begin{array}{llll}
e_4: & e_3: & e_1: & n_1 \\
 & & & n_2 \\
 & & e_2: & n_2 \\
 & & & n_3 \\
 & & n_1 & \\
 & & e_2: & n_2 \\
 & & & n_3 \\
 & n_3 & &
\end{array}
$$

To understand this last, indentation matters. Yuck. This is still hard to read. Perhaps JSON will do:

```
{
   Link: {
      Link: {
            Link: {
                  Node: "one",
                  Node: "two"
               },
            Link: {
                  Node: "two",
                  Node: "three"
               },
            Node: "one",
            Link: {
                  Node: "two",
                  Node: "three"
               }
         },
      Node: "three"
   }
}
```

Much better, but still a bit awkward. Perhaps all the links can be replaced with square brackets, so an array of arrays? Doing this will convert the JSON into funny-looking s-expression. This will be presented shortly, below; in the meantime, it is left to the reader's imagination.

A strange thing has happened here: the nodes n_1, n_2 and n_3 appear in multiple places, yet they are supposed to be "the same thing". Likewise for e_2, which appears twice, but is meant to be the same e_2 both times.

Consider representing the metatree with JSON (or something similar, *e.g.* YAML) The duplication presents a difficulty for JSON. Ordinary JSON does not support object references; there is no way to say the multiple e_2's and the n_k's are "the same thing". There is an IETF draft standard for references, but it is not widely used. Thus, although metatrees can be represented with JSON, some care must be taken when parsing them: one must find all repeated objects and understand them to be universally unique. That is, one must replace all repeated objects by universally unique references.

UUID's

It is very tempting, at this point, to write "just use universally unique identifiers" (UUID's). This is good enough for local address spaces; if nothing else, then an ordinary C/C++ pointer is a UUID to the object. But, when writing a text string, what UUID shall one use? Much worse, UUID's cause major, fundamental problems when considering network-distributed storage; this is the problem of UUID collision. One solution to avoid UUID collision is to have a single centralized atomic issuer of ID's that can guarantee uniqueness. This introduces a single, centralized bottleneck. Another solution is to use cryptographic hashes. Each meta-tree string can be hashed down to a number. To avoid collisions due to the birthday paradox, the hashes have to be quite large. For 1 million distinct atoms, a 64-bit hash and crossed fingers should be enough; uncrossing the fingers requires at least a 96-bit hash. For a trillion atoms, a 128-bit hash is just barely enough and a 192-bit hash is preferred. These eat up RAM (as compared to pointers) and the computation of cryptographic hashes requires significant CPU overhead.

Merely adding references to JSON is not enough to solve the problem, either. There may be millions or billions of meta-trees; forcing the user correctly employ references across all of them, without error is just not reasonable. Worse: new meta-trees could be added weeks or months later. Using references would force the user to maintain a table of references over long time periods. How? Where shall this table be stored? How much RAM will it require? References can be convenient, but they don't solve the fundamental representational problem for metatrees.

Insertion and Deletion

When a metatree is added to a metagraph, a scan must be made to determine if the tree, or any subtree already occurs in the metagraph. How might we know this? By looking it up! This does require indexing on the metagraph. Using indexes as described in the previous section introduces a bit of a chicken-and-egg problem. Thus, at the metagraph store level, a naked b-tree or hash table is required. Here, (non-cryptographic) hashes are useful, as they can be made small, and can be used to speed up compares in a b-tree, or used in a hash table, which will resolves any collisions.

Compared to an SQL database, this disambiguation adds to the cost of insertion and deletion into a metagraph store. In SQL, tables have primary keys; these are the primary indexes for each row. They are always made explicit; a table schema is declared with one of the columns designated as "*PRIMARY KEY*". In a metagraph, each distinct node and link is ... distinct, and unique. No explicit declaration is needed; uniqueness is implicit in the definition. This is kind-of nice; why mess with primary keys, if one doesn't have to? One less thing to think about, and not unrelated to the "automatic" nature of table normalization with metagraphs. Metagraphs provide automatic key maintenance. Nice!

Compared to a graph database ... well, how, exactly is one supposed to say that "this vertex is the same as that vertex" in a graph store? This is non-trivial; it requires either references or some other technique; if one is not careful, one finds oneself performing queries while inputting data. Again, we seem to have discovered, quite accidentally,

with no explicit intent, that hypergraphs offer an elegant property that graph databases lack.

Types and Atomese

Lets return to the string expression $(e_4 : (e_3 : (e_1 : n_1, n_2), (e_2 : n_2, n_3), n_1, (e_2 : n_2, n_3)), n_3)$. Since nodes and links (atoms) are unique, the subscripts on the hyperedges are not required. Nor is the use of the colon. The above is representable with the S-expression

```
(Link
   (Link
      (Link
         (Node "one")
         (Node "two"))
      (Link
         (Node "two")
         (Node "three"))
      (Node "one")
      (Link
         (Node "two")
         (Node "three")))
   (Node "three"))
```

Just as with JSON, the same considerations about repeated objects apply.

Types

Perhaps the most interesting aspect of the s-expression format is the first explicit appearance of types. Here, there are just two types: the nodes, and the links. The nodes have to be given a name, so as to disambiguate them from one-another. The links do not: the metatree structure is enough to disambiguate one link from another. Again: nodes and links are naturally universally unique (or conversely, every node having the same name, every link having the same structure is indistinguishable.)[7] When written in this form, the doubly-typed nature of the metatree is calling out for generalization. The types are sitting there, almost begging: "do something with me".

Multiple types are not strictly needed for storage: indeed, the two tables, one for nodes, and one for links, is sufficient to tell them apart. Actually generalizing the written form above to multiple types does require additional storage. Is it worth it?

In comp sci, in practice, types are incredibly useful. If they are not given explicit, distinguished treatment in the metatree, they will appear, without question, in the attribute section. Hang on, whats a type?

[7] This is eerily similar to atomic physics. All electrons are indistinguishable. All protons are indistinguishable. When combined into atoms, however, this changes. An element of one type (say, carbon) is clearly different than another (say, oxygen), but all carbon atoms are indistinguishable... unless they appear in an organic molecule. That is, it is the relationship of elementary particles to one-another that gives structure to the universe. It is not the particles themselves.

By "types" it is meant both the types of computer science, and of type theory. In ordinary programming, ints, floats and strings are types (these are the "primitive types"), and so are object-oriented classes (these are the "compound types" or "product types"). Function calls have a type signature too, and ML, CaML, Haskell and F# are all about programming with types. Unlike these standard, well-established languages, there is no particular need to limit oneself to a proscribed set of primitive types with a metagraph. One can have as many different primitive types as desired. One can have user-defined types.

The simplest example of a user-defined type is, for example, a *CatLink*. What's a *CatLink*? Well, cats - house cats, are those things with paws and whiskers, and have an entire subdivision of the internet devoted to them. What do you do with a *CatLink*? Well, anything you want: use it to link to a photo URL. Use it in some phylogenetic tree. Describe the properties of a cat: fur, tail, *etc*. Generic knowledge representation.

```
; "The cat sat on the mat", as a typed metatree.
(SittingLink
   (ObjectNode "cat")
   (ObjectNode "mat"))
```

Of course, a good collection *IsALink*s, *HasALink*s and *PartOfLink*s, together with some reasonable verbs might be able to achieve the same goals ... but those are ... more types. The point here is that there is no need to limit oneself to ints, floats and strings, and the object-oriented classes one can build out of these primitives. There is no need to be constrained by the philosophy of functional programming language design.

Wait, what? Programming language design?

Atomese

Metagraphs fit well with type theory: it is easy, effectively trivial to represent product types, sum types and function signatures with metagraphs. This, in turn, opens the door to a very unusual property of metagraphs: they can be interpreted as programs. To illustrate, consider the case of an abstract syntax tree. Conventionally, one says that an expression, such as "*if (a > b) return a+b;*" can be represented by an abstract syntax tree. This is a tree where each internal node of the tree is one of the operators: plus, greater-than, if(). Leaves are variables or constants.

For the case of (typed) metatrees, the representation is "obvious":

```
(IfLink
   (GreaterThanLink
      (VariableNode "a")
      (VariableNode "b"))
   (PlusLink
      (VariableNode "a")
      (VariableNode "b")))
```

This not only represents the expression "*if (a > b) return a+b;*" but it can also be made active: when supplied with values for the variables, the comparison and the

addition can be explicitly performed.

This is a key property that (typed) metatrees have, and that ordinary (typed or untyped) graphs do not. Ordinary graphs are not, in general, DAGs: they can contain cycles. Erecting an abstract syntax tree structure on an ordinary graph store is not natural; one must impose some representational constraints. Now, of course, one could limit oneself to using only DAG's, but any sane, normal designer of any graph store would strenuously object to adding executable graphs as a fundamental property of the graph store itself. The execution of expressions is "orthogonal" to the storage of graphs. These tasks have "orthogonal concerns" or "non-overlapping concerns". Strangely, metatrees erase that orthogonality.

ProLog

Perhaps this seems dubious. The core issue is most easily exposed by looking at something like ProLog. It's a programming language, of course. It makes abstract syntax trees fairly explicit: it is hard to understand how prolog works, unless one learns to think in terms of trees. Certainly, the "cut operator" in prolog is one of the early stumbling blocks for programmers learning prolog. It's hard to see what it does, until one realizes that it is literally cutting branches off of a tree. At the same time, chapter one, the very first chapter of any book on prolog programming is rife with examples of knowledge representation. One will find something like this:

```
father_child(tom, erica).
father_child(mike, tom).
sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).
```

The first two statements are assertions of fact, the third is a inference rule. They have a very obvious representation as trees, and, more precisely as meta-trees. Yet, strangely, prolog programs are encoded in text files, and *not* as a collection of entries in a graph database!

This last observation becomes truly bizarre, if one imagines some non-trivial knowledge representation problem. Imagine keeping census data in prolog. Maintaining a text file with dozens of lines of code for millions of people is absurd. Applying transformations or graph rewrites is impossible for text files. Consider the inference "*sibling*" above. It can be taken as a run-time directive, but it can also be taken as a graph rewrite rule: find all graphs having two terms, "*parent_child(Z, X)*" and "*parent_child(Z, Y)*" and create a new term "*sibling(X, Y)*". Such a rewrite is not all that hard in a database; in SQL, some appropriate combination of "*SELECT INTO*" and "*JOIN*". [8] Looking at it this way, prolog "wants" to live in a database. Yet it doesn't.

So, yes, superficially, databases and execution are orthogonal concerns. But if one begins to look at what people actually do, in practice, with SQL, and how, in practice, they design code for object-relational databases, that orthogonality gets a bit fuzzy. It's downright cloudy by the time one is writing PL/SQL statements. Coming from the opposite direction, as prolog does, makes the "separation of concerns" even cloudier.

[8] The only stumbling block being the need to reference the "parents" table twice during the join. Hmm... what was that bit, about explicit graph-walking, again?

Intermediate Languages

These aren't even the only examples. Inside of compilers, one finds "intermediate languages". For Microsoft, this is the CIL or Common Intermediate Language. For Gnu GCC, it is GIMPLE. These are somewhat like assembly code, but abstract, and not specific to CPU hardware. They encode abstract syntax trees, and thus sit above the bytecode layer. Looking carefully, one will observe that "compiler optimization" actually consists of a very small database of the currently active, non-retired abstract syntax trees, and that optimization is a collection of re-write rules (in the sense of the prolog rewrite rule, above) being applied to the trees in the active database.

Of course, what happens inside a compiler is very narrow, and very carefully crafted to suite the needs of the compiler, and nothing more. This is a high art that has been honed over many decades. Intermediate languages are almost never written to disk, except as text strings. Yet the lessons they teach can be taken as generic: creating graphs, and then transforming them, via graph re-writing, is a generically useful operation. Representing knowledge as trees, and specifically, as type meta-trees, offers a huge representational efficiency over plain graphs or SQL tables or key-value stores. The efficiency is both computational (RAM usage, CPU cycles) and expressive: writing inferences in prolog really is a lot easier than writing SQL statements.

Atomese

The above seems to be saying "typed metatrees are a programming language". This is true, and in the OpenCog AtomSpace, it actually is a language, called "Atomese". But what kind of language is it? Well, if one abstains from providing some large collection of Link and Node types, then the answer is "any kind at all". The metatrees provide just enough structure, just enough groundwork, to create your own language. This is vaguely comparable to lex and yacc: these are just some basic tools that allow competent users to design custom, user-defined languages. Likewise here: metatrees offer the toolset needed to design a custom knowledge representation system. Perhaps offering a built-in *IfLink*, *GreaterThanLink*, *PlusLink* and *VariableNode* is a good idea, and perhaps not. Nothing compels a metagraph database system to offer these; at the same time, it makes it easy to offer these. To some large degree, it is sufficient to offer a typed metagraph storage system, and nothing more.

The OpenCog AtomSpace almost does this, but it also tacks on more than 100 different predefined Atom types. This is effectively a historical accident, having to do with how it evolved. A strong case could be made for splitting out this vast assortment of predefined types into their own module. This is easier said than done; the AtomSpace continues to be a research platform for the concepts described here.

Conclusion

Several important claims were made here about metagraphs and metagraph stores. These are:

- Metagraphs are a simple and relatively minor generalization of graphs.

- Metagraphs are more representationally compact than graphs. They are more efficient at representing data.
- Table normalization, normally an intellectually demanding task for relational database design, comes "for free", when one works with metagraphs.
- Specifying metagraphs as text strings is easier than specifying graphs as strings.
- Typed metatrees can be abstract syntax trees; ordinary graphs cannot.
- Typed metagraphs provide a powerful under-girding for symbolic computing, not just for representing knowledge, but also carrying out the computations themselves.

This text started out as an attempt to describe the RAM and CPU usage properties of metagraphs, and unwittingly turned into a strong statement about the general utility of metagraphs as a foundational system. As such, it provides a strong statement about many of the design decisions that went into the OpenCog AtomSpace, which is the primary research vehicle for these ideas. Even so, it is not the culmination of the journey. The series of chapters in this directory are about sheaves. These provide a general connectionist approach to data representation, and the sheaf approach is different from metagraphs, having distinct representational properties (including CPU and RAM usage). Understanding metagraphs is a key gateway to the sheaves project. Fortunately, the metagraph approach, taken by the current AtomSpace is fairly mature, having more than a decade of implementation and use experience behind it. This text gives a flavor of where it has arrived.

References

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