Graphs, Metagraphs, RAM, CPU

Linas Vepštas*

2 Sept 2020 Version 1.0
22 Nov 2021 Version 2.0
5 Dec 2021 Version 2.1

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 minor modifications to the definition of a graph. The resulting structure proves to be simpler and more efficient for representing graphical data.

This starts a domino chain of claims. Metagraphs are more flexible than either graph stores or SQL-style table stores or JSON-stores for storing data. Metagraphs have a natural query language that is more powerful and easier to use than SQL-inspired query languages, mostly because the table-join concept is obviated and replaced by isomorphism.

It is easy to specify a term rewriting system with metagraphs; it is a trite extension of having a query system. Both forward and reverse queries are natural, and so metagraphs provide a solid foundation for rule-based systems.

Metagraphs are naturally typed, in the sense of type theory. Metagraphs are easily reified, and so the type system is itself trivially expressible as metagraphs.

Metagraphs can be used to specify a programming system/language, but that language is low-level, and not really suitable for humans. It is ideal for algorithmic (machine) manipulation, analogous to intermediate languages inside of compilers.

Most of these claims are experimental results, rather than assertions or theoretical results. The experimental platform is the OpenCog AtomSpace and Atomese language.

This is part of a sequence of texts on sheaves (*a la* sheaf theory). It is a prerequisite for understanding the practical foundations on which sheaves can be built.

Introduction

Currently, graph databases are popular, as they have a rather distinct performance profile, differing from both 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

*linasvepstas@gmail.com

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.

Graphs offer an interesting storage format for many reasons. Coupled to those reasons is a need for graph traversal, and another need for graph query (solving the subgraph isomorphism problem). Thus, the questions examined here include how one might best be able to traverse graphs, and how to best perform subgraph matching. By "best", it is meant algorithms that find a good balance between speed and size.

This text is organized into the following sections:

- An naive definition of graphs.
- An aside about attaching attributes to the edges and vertexes of a graph. This is effectively a quick summary of traditional knowledge representation formats, specifically, tables, JSON and s-expressions.
- How graphs can be represented in memory (RAM), and how much storage is used to represent a graph.
- Modifications to the representation that result in hypergraphs and metagraphs, including a n analysis of memory usage. A formal definition of a metatree, the core component of a metagraph. A key result is that metatrees are more compact than graphs for most data representation problems. The first hints that they are also more natural and easier to work with start to show.
- A discussion of the concept of "indexing", and how indexes are used to find data. Contrasts are drawn between indexing in graphs, and indexing in SQL databases. (The noSQL databases are given short shrift, mostly because of a famous result showing that they are actually an "opposite category" best termed coSQL.[1]) A key observation is that metagraphs unavoidably have a certain kind of index built into them; they come equipped with a (partial) index.
- A discussion of partial indexes, and their role in database normalization. An argument is made that metatrees are self-normalizing. This feels perhaps uncomfortable, given the vast resources that database textbooks expend on teaching normalization to students.
- Representing metatrees as strings, with specific attention focused on the fact that subtrees are necessarily unique, and how this uniqueness can be managed in a practical way. This includes a rejection of UUID's as a technique for assigning unique labels to metatrees.
- A discussion of insertion, deletion and graph mutability. A primary result is that metatrees are necessarily immutable, as otherwise a host of useful and desirable assumptions are shattered. This is a positive result: immutability is a marvelous property for data structures to have, when they are being accessed by multi-threaded systems. A secondary result is that a metagraph database looks exactly like a single metatree, but a mutable one. Thus, there are two kinds of metatrees: the immutable ones are called "data" and the mutable ones are called

"databases". Since they are both metatrees, they can be layered, nested, hierarchically stacked however desired.

- A review of query languages and graph traversal. A primary result is that all SQL-inspired query languages inherit the table-based viewpoint of SQL. This in turn causes one to think in terms of table joins, which, in a graph setting, looks like a graph walk, chaining from one edge to the next. This is explicitly seen on systems such as GraphQL, SparQL, Neo4J or grakn.ai (TypeDB), which explicitly present graph queries in terms edge walks.

- An example of a metagraph query language, which hops outside of the *QL paradigm of edge walks (aka table joins), into a query mode that more closely resembles pattern matching. Conventional pattern matching is implemented in terms of a state machine, and so it is not enough for a proper graph query. A true graph walk requires a stack machine; it requires a recursive walk of a graph.

- A short discussion of query analysis and query planning for a metagraph query language.

- A presentation of inverted queries. These are queries that are "answers" in search of a "question". That is, they are queries that consist entirely of constant terms, that can be matched to forward-queries having variables in them. Inverted queries are the bread-and-butter of chat-bots and rule engines. The key observation here is that metatrees represent inverted queries just as readily and naturally as forward queries. That is because a partial index on a metatree can be understood as a kind of trie. Metatrees are therefore able to unify both database concepts, and rule engine concepts with the same framework.

- A discussion of the natural interpretation of metatrees. The experimental result here, after a decade of use, is that metatrees correspond to type-theoretical types. They reify easily and naturally, and have an obvious suite of type constructors. As you might guess from the tone of voice here, the type constructors are themselves just metatrees. The concept of sheaves is touched upon very quickly here, as the sheaf elements are just "jigsaw puzzle pieces" and the jigsaw connectors are types.

- A discussion of execution, evaluation, fexprs, macros, $vau, term rewriting systems and intermediate languages. Obviously, abstract syntax trees are a special case of trees, and trees are a special case of metatrees. Abstract syntax trees achieve two things: they encode executable programs, and they also encode syntactic structure that can be re-written by homotopic transformations. As such, they bridge across some old ideas in lisp (fexprs, $vau) and newer ideas in knowledge representation (prolog).

- A discussion of human-oriented programming languages vs. machine-oriented programming languages. Metatrees and metagraphs are themselves too low a level to be suitable as a programming language which human beings would want

to use on a daily basis. Instead, they have more in common with compiler intermediate languages, which are term rewriting systems used by machines (compilers) to perform transformations on data.

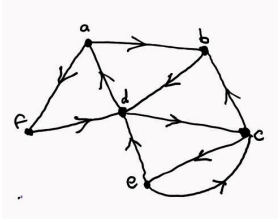
As a conclusion, it is noted that the OpenCog AtomSpace has been the experimental platform on which all of the above results have been obtained, and embodies the most of the ideas presented here.[1]

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 three fundamental choices available for storing attributes: merged schema+data, disjoint schema–data and s-expressions. An examples 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 for the table; individual rows in the table do not need to repeat the schema. Clearly, for tables with more than a few rows, the tabular format offers a huge advantage in terms of memory usage. Conversely, having many tables with just one or two rows each quickly becomes a table management problem; conventional systems are not designed to hold a million tables of one row each.

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

[1] See `https://wiki.opencog.org/w/AtomSpace` and `https://github.com/opencog/atomspace`

than the other rows. It doesn't even make sense to talk about rows in this way; if one row has a different schema than another, they aren't rows of the same table any more.

An example of an s-expression store is a key-value store. Here, the first word is taken to be the key; subsequent words are taken to be a list of values associated with that key. The idea of s-expressions emphasizes that the key-value store can be hierarchically structured. An example is the Unix file-system structure: each directory can have files, but it can also have subdirectories, *ad infinitum*. In this sense, a URL can be understood to be a kind-of s-expression.

In a graph store, one has these three basic choices for storing attributes, both for vertexes and for edges. One might even contemplate a mixture; 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, as this has little or no impact on the primary topics here. With one exception: query and indexing. This is reviewed in a distinct section later on.

Emulating Tables

This section aims to make the above commentary a bit more concrete by working though some examples. Although the examples in this section are written in text, the reader is very strongly urged to try to imagine how these structures might be represented in RAM, and how much storage they might take. A later section will return to these questions of in-RAM representations.

Consider the need to store data about some students. Expressed in JSON format, it might be written as:

```
{ grades:
   [
      { student: {name: Joe},    {gpa: 3.5}},
      { student: {name: Mary},   {gpa: 3.6}},
      { student: {name: Rachel}, {gpa: 3.0}}
   ]
}
```

The square brackets denote a list; as is conventional with a list, all list elements are uniform and of the same type. Clearly, this appears to be a very regular structure. If there are hundreds of students, one might try to save some space by eliminating the repeated attribute names. If JSON is used for this, this becomes a column-store:

```
{ table:
   { schema: {name: string}, {gpa: float}}
   { names:  [Joe, Mary, Rachel] },
   { gpa:    [3.5, 3.6, 3.0 ] }
}
```

That is, each entry in the table is a column name, followed by a list of all of the values in that column.

This is distinct from a row-store. In a row store, one conventionally provides a schema describing the columns and their formats, much as above, and then provides the rows, one by one, as "records", or inhomogeneous lists of fixed length. That is,

each entry in the list has a different type (it has the type of the column), but each list is exactly the same length; thus, a "record".

Students	
name	gpa
string	float
Joe	3.5
Mary	3.6
Rachel	3.0

There does not appear to be any way of representing a row-store in JSON, at least not in the JSON as naively conceived here. One wishes to write something like the below, but the syntax makes it illegal:

```
{ table:
   { schema: {name: string}, {gpa: float}}
   { students:
      [
         (Joe,    3.5),    # This is not valid JSON!
         (Mary,   3.6),
         (Rachel, 3.0)
      ]
   }
}
```

It is this last example that makes clear that an s-expression store can offer the best of both worlds. This is shown in the next section.

OGRE: Open Generic Representation

A particularly nice and human-friendly API can be found in the OGRE module in BAP.[2] It is an s-expression database that allows new data structures to be defined in generic ways. In short, any s-expression is a valid record. Records do not need to be of the same length, or of the same type.

Thus, the previous example can be written as

```
(student (name Joe)    (gpa 3.5))
(student (name Mary)   (gpa 3.6))
(student (name Rachel) (gpa 3.0))
```

Note that these are three distinct records, and are NOT a list of three rows!

As written, this has a distinctly JSONic feel to it, in that every value is tagged with the field type that names it. But then, the OGRE documentation notes that this is equivalent to

```
(student Joe    3.5)
(student Mary   3.6)
(student Rachel 3.0)
```

provided that one already knows the column structure of the data. But this is easily achieved:

```
(declare student (name str) (gpa float))
```

This last statement resembles a conventional table-database table declaration.

One can go farther: there is no particular need to tag each row with the row-label 'student'. The most compact representation then appears to be the following:

```
(students-table
   (Joe    3.5)
   (Mary   3.6)
   (Rachel 3.0))
```

This last form now obviously has the shape of a row-store. Each row is a tuple, all tuples look alike.

This last form enables a rather conventional SQL query system to be defined. There is a table definition, providing column types, and column labels (the labels are needed for query/search), and the individual records (rows) in the table are uniform. The uniformity allows both for very compact storage and easy query.

The nature of query in these three styles, and what it implies for RAM consumption and CPU use, is quite dramatic. They can have remarkably different performance profiles, and even some remarkable limitations in the kinds of queries that can be imagined or can be written. This is a more complex topic, and will be returned to later in this text.

Tables and Algebraic Data Types

Category theory provides a modern foundation for a lot of thinking in computer science,[2] and one particular aspect of it, type theory, provides a theoretical framework for the practical construction of data types.[3] Much of this text presumes these theoretical foundations are percolating in the background, and so a few brief words are in order.

An SQL Table declaration corresponding to the above example is

```
CREATE TABLE student (
   name text,
   gpa real
)
```

The interpretation is meant to be straight-forward: a table with two columns, one of which is an unlimited length character string, the other a single-precision floating point number. We could do the same in C++ (or Java, or any conventional object oriented language) and write

[2] See Wikipedia: https://en.wikipedia.org/wiki/Category_theory but perhaps much more instructive and practical would be a book such as "Category Theory for Programmers", Bartosz Milewski (2019) available here: https://github.com/hmemcpy/milewski-ctfp-pdf.

[3] See Wikipedia, https://en.wikipedia.org/wiki/Type_theory for an overview of the abstract theory of types, stemming from the nature of functors in category theory. The word "type" here is the same as that in computer programming: the data type, see https://en.wikipedia.org/wiki/Data_type for a review of the more concrete ideas.

```
class student {
   char* name;
   float gpa;
};
```

In terms of knowledge representation, these all express the same idea. The practical differences between these are whether the structures are created at runtime (as they would be for SQL, or scheme/lisp) or compile time (as they would be for C++ or OCaml).

All of these are examples of "compound types" or "composite types", or more loosely, the "product type", or, more narrowly, the "Cartesian product".[4] The Cartesian product of two sets A and B, in set-builder notation, is

$$A \times B = \{(a,b) \,|\, a \in A \text{ and } b \in B\}$$

We want A to be the set of students and B to be the set of grades they could get; these are not only different sets, but the types of the members of these sets are different. This leads to the product type $\mathbf{A} \times \mathbf{B}$ where the $\mathbf{A}$ is the type of text strings, and $\mathbf{B}$ is the type of numbers with decimal places. Members are again written as (a,b) or perhaps $(a : \mathbf{A}, b : \mathbf{B})$ and are sometimes called "tuples", for the obvious reason that the comma is a great way of writing lists of things, which incidentally are Cartesian products.

If this starts feeling circular, that is because it is: the SQL statement and the C++ class declaration above are both examples of Cartesian products. Tuples and records are generically examples of products.

Category Theory and Coproducts

We're going to jump the gun a bit, and briefly mention coproducts, as they are dual to products.[5] Examples of coproducts in programming is the "union" statement in C/C++, or the variant type in other programming languages.[6] In type theory, these are referred to as "sum types", and are dual to the product type.[7] In type theory, these are referred to as the sigma type Σ and the pi type Π,[8] and in formal logic (set theory) lead to the notion of the sigma-pi hierarchy.[9]

[4] See Wikipedia `https://en.wikipedia.org/wiki/Product_type` and `https://en.wikipedia.org/wiki/Cartesian_product`. Here, and in many subsequent footnotes, the reader will be directed at Wikipedia pages. This is meant for clarity: if the reader is unfamiliar with a topic or phrase, the Wikipedia page provides the requisite details. These Wikipedia references are not meant to be hand-waving appeals to abstract concepts: they are meant to provide precise, formal definitions for the ideas discussed here. The intended sense of the words and concepts brought up here are meant to be precisely, exactly those described in the Wikipedia articles. Although this text attempts to be understandable without prior, indepth understanding of that content, it would also be the case that all of the finer points and subtleties will be missed, without this acquaintance. Read the Wikipedia articles. You should be familiar with what they talk about.

[5] See Wikipedia `https://en.wikipedia.org/wiki/Coproduct`.

[6] See Wikipedia `https://en.wikipedia.org/wiki/Union_type`.

[7] See Wikipedia `https://en.wikipedia.org/wiki/Tagged_union`.

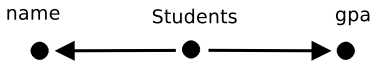
[8] See Wikipedia `https://en.wikipedia.org/wiki/Dependent_type`.

[9] See Wikipedia `https://en.wikipedia.org/wiki/L%C3%A9vy_hierarchy` and `https://en.wikipedia.org/wiki/Descriptive_set_theory`.

Category theory is sometimes called the "theory of dots and arrows", and a central notion is the "opposite category" C^{op} that is dual to the category C. In the opposite category, the direction of all of the arrows are reversed. Naming-wise, the prefix "co-" is prepended to names: thus products are dual to coproducts. The coproduct type and the sum type are the same thing. Insofar as the product can be drawn as

the coproduct can be drawn as

It is common in category theory to leave the dots and arrows unlabeled, as often it is more interesting to just talk about the shapes, rather than the labels. But to make it clear, here is the product type again, with the labels:



The arrows are the "canonical projections" or the "projection morphisms" of the product. The union type or coproduct will not be illustrated. The reader is encourages to imagine a C/C++ "union" statement and arrange the arrows appropriately.

The product and the coproduct, taken together, are often referred to as Algebraic Data Types (ADT).[10] In the text that follows, there will be lots of arrows, pointing both forwards, and backwards. These can all be understood in terms of the type-theoretic foundations sketched here. A more formal presentation of the type theory is deferred to a different text in this series, describing the jigsaw puzzle-piece metaphor, and how it is used to construct sheaves.

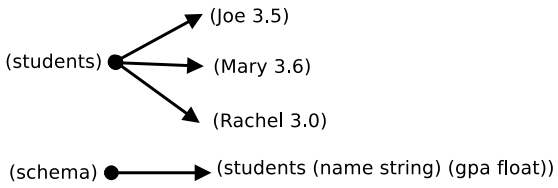
Not all of the arrows that will be drawn in the subsequent text can be understood as category-theoretic arrows. Category theory has some fairly strict ideas about what one can do with arrows, and thus, not every diagram will be a valid category. However, many of them will be. The most important examples of arrows that can be interpreted as category-theoretic arrows are the indexes on SQL tables (PRIMARY KEY and FOREIGN KEY) and the opposite arrows in noSQL key-value databases. This is gleefully articulated by Meijer and Bierman.[1] We shall need arrows going in both directions in order to perform efficient metagraph queries, to e explained in later sections.

Graphs for Storing Data

The above listed three ways of storing attributes on a graph, and yet ignored the graph itself. Thus, for completeness, lets draw the diagram of how this might be stored in a graph.

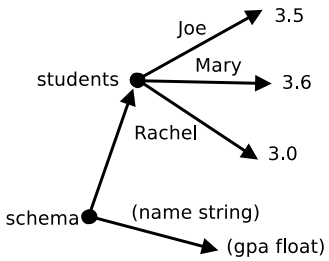
Obviously, its "graphical" if it is drawn as follows:

[10] See Wikipedia, `https://en.wikipedia.org/wiki/Algebraic_data_type`.



In the above, the vertexes are labeled with s-expressions; the edges are not labeled. It consists of two disjoint graphs; one of the graphs is used to encode the type information used in the other graph. There are a large variety of choices that can be made in how this is done. The above just shows one way; it might not be the best way, although that depends on the problem domain.

Below shows another alternative, this time with labeled edges:



As before, the graph explicitly includes a description of the schema used in another part of the graph. For the first time, a non-trivial graph is drawn. It begins to give a hint of the utility of graphs for general knowledge representation.

Mixed Representations

In both of the above examples, the graph store was assumed to have some distinct, separate attribute store for each vertex, and possibly each edge. Thus, in a sense, this is not a "pure" graph store, where each edge or vertex can only have an atomic singleton value on it (a single number or a single string). Of course, one could rework the above graphs so that each vertex/edge does have just one single value on it. While the simplicity of singleton-valued graphs is perhaps intellectually appealing, it proves to not be practical for everyday use. Assuming singleton-valued graphs also has some very strong implications on RAM and CPU use, and the ability to perform searches/-queries. This point will be returned to below. In the meanwhile, this kind of mixed model will be assumed: some data will be stored graphically, and some data will be stored as complex attribute sets attached to the edges and vertexes.

Some historical hand-waving can be done to justify the origins of a mixed-model graph store. In predicate logic, and in model theory, one distinguishes the predicates and terms that one is discussing, from the truth values or valuations that can be assigned to them. It is relatively straightforward to envision predicates and terms as trees or DAG's. The truth values are distinct from the graphs themselves; instead, they are

an assignment of true/false to each expression.[11] A straight-forward example can be found in Prolog: in the beginning, there are expressions; by means of inference, truth-value labels are inferred. The truth values are not a part of the original graph; they live outside of it, as an attribute.

Of course, things get interesting as soon as one leaves the domain of crisp-logic T/F values, and considers instead valuations that are Bayesian probabilities. Some monkeying around then leads one to distinguish Bayesian probabilities from fuzzy logic valuations. Other kinds of data worms it's way in: confidence intervals; frequentist counts; marginals. Some of this data is non-numeric, but are choices (*e.g.* true, false or unknown) or explicitly symbolic (*e.g.* red, blue or green). It usually does not take long to realize the practical need for a mixed-model graph database, supporting both good graph features, and also supporting complex attribute systems attached to each edge/vertex.

Representing Graphs in RAM

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[12], and the Gnu gdbm.[13] These have been followed by Google's LevelDB[14] and Facebook's extensions RocksDB.[15] 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.[16]

Effectively all the discussion in this text assumes uniform memory access; that is, a "flat" memory topology, where any location in storage can be accessed with the same latency as any other. Modern CPU's are all NUMA machines;[17] it is too much for this text to dive into the issues that these pose.

[11] This is the grand leap from Aristotelian, "classical" logic, to predicate logic. The disentanglement of truth-valuations from propositions enables giant leaps in reasoning abilities. This in turn opens the ability to further disentangle syntactic from semantic entailment, thus allowing a connection to grammar and language to be made. The adjointness between grammar and graphs is precisely the core, central reason why properly constructed graph databases are central to the pursuit of AGI.

[12] See `https://en.wikipedia.org/wiki/Berkeley_DB`

[13] See `https://www.gnu.org.ua/software/gdbm/`

[14] See `https://dbdb.io/db/leveldb`

[15] See `https://rocksdb.org/`

[16] One must resist one wide-spread and common temptation: layering an in-RAM database on top of another database that is RAM-hungry. This is at best counter-productive: for every byte consumed in one, one risks a byte consumed in the other. As a result, one can only store half as much data, or must purchase twice as much RAM. Despite the self-evidence of these last statements, such a layering is widely recommended, on the basis of the supposed superiority of the lower layer. One must be particularly cautious if the lower layer requires the use of network communications. Networking requires the kernel to become involved, tasking-switching and performing IPC. This can easily become a performance bottleneck.

[17] See `https://en.wikipedia.org/wiki/Non-uniform_memory_access`

Naive RAM Representations

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 or b-trees, depending on desired access and update performance.

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. In the most obvious, direct form, the edge table will have the shape

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	outgoing	incoming	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.

Table updates must be both thread safe and fast. It is easy to lock the table with a mutex, but this can quickly limit the amount of concurrency. The latest lock-free technology promises reasonable solutions; however, the technology remains immature. There are several implementations of concurrent hash maps, but none for concurrent multimaps (that also support erase). Likewise, lock-free tree implementations are absent; trees offer a considerably more compact storage format when tables are small.

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 lists of atoms.

Hypergraph representations

The naive representation for the hypergraph is a straight-forward extension of the edge table (the table on the preceding page above). The table below provides an example that is shown in the figure following below.

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 (quickly) is a core requirement. This means that, given a vertex, one must be able to quickly find the edges attached to it. This requires modification to the vertex table given before.

Several choices are possible. One is to add a new column for each positional location in the vertex list. That is, 0'th column holds the edges which have the corresponding vertex in the 0'th position of the vertex list, the 1'th column likewise. This can be read off from the table above:

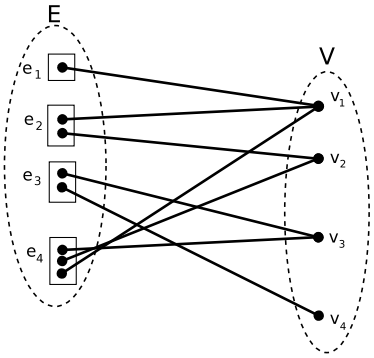
vertex id	edge-set-0	edge-set-1	edge-set-2	...	attr-data
v_1	$\{e_1, e_2\}$	$\{\cdot\}$	$\{e_4\}$		...
v_2	$\{\cdot\}$	$\{e_2, e_4\}$	$\{\cdot\}$		...
v_3	$\{e_3, e_4\}$	$\{\cdot\}$	$\{\cdot\}$		...
v_4	$\{\cdot\}$	$\{e_2\}$	$\{\cdot\}$		

To actually store this table, one must have 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; that 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.[18] Thus, the vertex table can take the simpler form

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 in the edge table 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.

[18] We've worked with natural language, genomics and robotics datasets.



The boxes denote the fact that the hyperedges are ordered lists. The E and V ellipses are the hyperedge and vertex tables. If the boxes could be collapsed to single points, this would be a 'true' bipartite graph; but they cannot be. The ordering is needed and important.

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 example 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

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

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 count of the incoming set is equal to the average count 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,[19]

[19]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

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

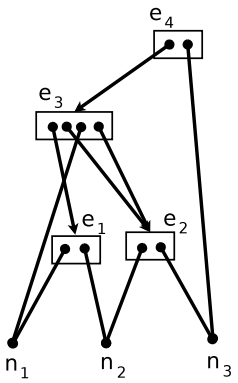
The scale factor A is data-dependent. For Wikipedia page views, $A \approx 20$, see `https://en.wikipedia.org/wiki/Wikipedia:Does_Wikipedia_traffic_obey_Zipf%27s_law%3F` for graphs and

so 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 'true 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".[20] 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. Neither of these conversions are faithful. The first erases the ordering within the box, while the second erases the grouping that the box provides.

The node table for a metagraph is effectively the same as the vertex table for the hypergraph (on page 14) for the hypergraph, before. For the example metatree, it is

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

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.

[20] Later on, metatrees will be called "Atoms". An Atom is simply a term for something that can be either a "Node" or a "Link". As we haven't yet clearly defined a Node or Link yet, the term "metatree" will suffice.

The link table now requires both an outgoing-atom list, and an incoming-link set. The incoming set can only consist of links; the outgoing list can consist of either nodes or links.

link id	outgoing-list	incoming-set	attr-data
e_1	(n_1, n_2)	$\{e_3\}$	...
e_2	(n_2, 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 (on page 12) of an ordinary graph store. It has both 'incoming' and 'outgoing' sets, as before. There are two differences: the outgoing set is no longer a set, but a list. The other is even stranger: the first column (the column of vertexes) has been replaced by a column of links!

Perhaps we making the wrong comparison? Oddly, it also resembles the edge table (on page 12). For ordinary directed graphs, each edge is a vertex pair. The vertices of an edge may be termed the 'head vertex' and the 'tail vertex'; equivalently, the outgoing vertex and the incoming vertex (or *vice-versa*, depending on how one envisions the arrows). These two endpoints are replaced by the incoming set and the outgoing list. Now, the first column remains the same: a column of edges, in both cases. The overall four-column form remains the same.

Looking at this table, one might imagine that 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. This will be looked at next.

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 a collection of tables, or key-value database, or a JSON-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. Some examples of "difficult" or "impossible" queries will be looked at in a later section.

It is worth remarking that the so-called noSQL databases are effectively 'identical' to SQL databases. They are 'identical' in the sense of being categorical opposites: the directions of all arrows are reversed. This was explicitly articulated in a famous paper by Meijer and Bierman.[1] Much of the following uses an SQL-style notation; if the

reader is more familiar with key-value databases, then simply reverse the directions of all arrows to obtain the equivalent discussion.

A few comments are in order regarding the SQL-style query notation. It has become dominant. One can look at systems as different as SparQL and GraphQL, or even the OGRE query language[2] (*op. cit.*) and clearly see not just the influence of SQL, but in fact a nearly verbatim copy of it. There is a reason for this dominance, and it is not (just) history. The reason is anchored (once again) in table representations, and the presence (or rather, lack thereof) of inbuilt indexes in the tables.

This is also the place to make a heretical claim: SQL or it's variants (GraphQL, SparQL...) are *NOT* the best choice for a metagraph query language. The shackles of *QL thinking are remarkably hard to escape. Even the principled, category-theoretic foundations such as the Functorial Query Language (FQL)[3] fail to evade the problems presented here, primarily because it is still table-based at it's heart (FQL is reviewed below, to illustrate several problems). A better alternative will become clear, once a more careful examination is made of the role of indexes in queries.

However, in a metagraph database, just as in a table-based database, 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';*".[21] 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 all records having that value. The `SELECT` is then straight-forward: given D, it need only return $\{R\}$. 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.[22] The point of the index with to replace an $O(N)$ cpu-time search with an $O(1)$ cpu-time search. The price of doing so was an $O(N)$ RAM structure.

Table-based information has certain kinds of representational difficulties: imagine the case of an employee with dashed-line reporting to multiple departments. This might

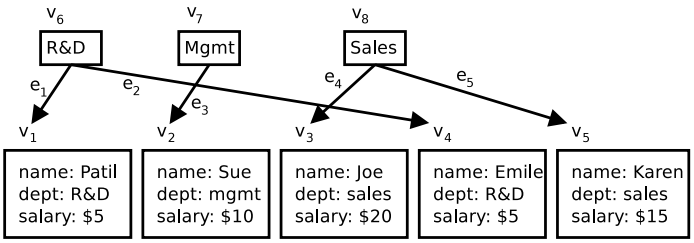
[21] The author would like to apologize for this seemingly non-sexy example. It is the stereotypical example from database textbooks, and harks back to the 1960's, when working out the fine details of management science actually was a sexy research topic, and helped power the economic ascent of the Western 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.

[22] Well, in practice, they often do; but now imagine the task of finding all records with a null value in some column...

motivate one to cast an eye towards graph databases.

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 creates an unlabeled, attribute-free edge, from the department name back to the full employee record. Finding all employees in "*sales*" is fast: one searches the vertex table (on page 12) to find the vertex "*sales*", and then traces all outgoing edges to the full record. The graph can be thought of as a table with a special "built in" index: the index of outgoing edges.

This is to be contrasted to a conventional table database. The contents of a conventional table (row) 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 documentation for conventional table databases does not 'talk about' vertexes and edges; but, *de facto*, this is how system architects think about things. When they think about creating an index on a table, this is what they think of.[23]



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: If one is interested only in sales, then not every employee has to be indexed! In a graph database, it is possible to create only one vertex, "*sales*", and hook up edges to only that one. Effectively, this is a partial index, with correspondingly less RAM usage! This is not possible in a naive table system; one needs a system which explicitly supports partial indexes. As it happens, most-all SQL systems do. An experienced SQL DBA knows exactly how to achieve this effect: "CREATE INDEX ON employees(department) WHERE department=sales;" This is not a big deal, and 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 internal graph-based nature is *ad hoc*, evolved over the decades and mostly hidden from the

[23]This is not meant to be a psychological assessment; rather it is meant to provide a translation between the 'algebraic' form of typed commands sent to relational DB's, and an equivalent 'geometric', visual form. It is the denotational semantics of the index. The geometric form can be understood as being either metaphorical and abstract, or it can be taken literally, as a collection of pointers and the things they point to.

user. There are no explicit graph-walking directives in SQL.[24]

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 overt. Surprise!

The explicit hypergraph representation does cost more than its implicit form in SQL databases. 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 traversable. 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. Mashing it up two levels deep is tougher. Can you count to three? It would be a significant challenge, even for experienced DBAs. Some more examples of challenging queries will be presented later.

Suppose you are clever enough to write deep table joins in SQL. Is your query planner as clever as you? Most SQL systems have a query analyzer or query compiler, which takes a given SQL statement, analyzes it's structure, and then creates a plan as to which data shall be fetched first, and in what order. Poor planning results in poor performance, sometimes disastrously poor performance. For large databases, there has been a vast (multi-billion-dollar) investment in sophisticated query planning. Walking graphs, represented in terms of tables and indexes, is not for the faint of heart.

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

[24] Aside from table joins; more on that later.

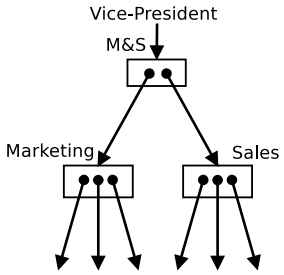
that had to be parsed to figure out "what to do". Let's assume that this cost has been amortized, and that there is a way to get a handle to a query that has already been analyzed. Query run-time execution is then a matter of finding the vertex "*sales*", tracing the edges to each of the employees, and 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. That is, if you have "*sales*", you already have the set $\{R\}$ and no further lookups are required. The dominant cost is almost surely the procedures 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 optimization. Some key aspects of this are reviewed in the next two sections.

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.[25]

[25]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".

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, $5N \gg 2N + 2N_D$.

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 (this is the 'transitive closure' of a recursive relation.) 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 = orgchart.dept AND orgchart.division = 'marketing & sales';*".[26] 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", as you organize data into graphical form. 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 one's perspective about data and it's structure.

Comparing metagraphs to graph stores, one sees a different improvement. By discarding the edge table (on page 12) 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 of the metagraph link table on page 18) enables something quite dramatic: an easy graph walk, which is anything-but-easy in a traditional SQL database.[27]

[26] 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.

[27] 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 favorite programming language of choice), and once you are "programming", it is no longer "easy". A properly designed graph query system makes (should make) graph walks "easy". A properly-designed metagraph query system makes graph walks invisible (see next section.) 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

Index Maintenance

Indexes are central to defining the relationships between bits of data. But such relationships cannot stay fixed: as a database is used, reasons are found to add new relationships, and to modify or delete existing relationships. Practical issues arise: changes must be made without corrupting existing data or (unintentionally) losing information.

This proves to be a particular challenge to SQL-style databases, and it is educational to review the reasons. In an SQL system, the relationships between bits of data are defined by the initial database architect: the person who first creates the table definitions, the "database schema". Changes to those definitions (additions of new tables, the reorganization of columns in existing tables, *etc.*) requires a process of "data migration". It is conceived in this way, since one is changing not just a table definition, but one is also moving all of the rows in that table. This is a CPU and RAM-consuming process: for tables with millions of rows, or more, this may take hours or days. When migrating datasets, it is very important to not accidentally corrupt the structures containing the data through poor data migration design (*e.g.* by unintentionally dropping columns, or by breaking primary-key-foreign-key constraints, or breaking normalization by duplicating data). A means of performing database schema migrations in an accident-free way is important. Defining a principled approach to table rewrites is even better.

The Functorial Query Language (FQL) provides such a principled approach.[3] It is reviewed below. The key observation here will be that data migration (database schema migration) can be thought of as a form of rewriting, analogous to the idea of term rewriting or graph rewriting. Looked at more strongly, this is more than an analogy: table rewriting *is* graph rewriting. The idea to be examined here is this: If one keeps the adjoint functors $\Sigma \dashv \Delta \dashv \Pi$ of FQL, but discards the SQL tables on which they are founded, can the FQL approach provide a principled means for rewriting graphs? The answer appears to hinge on the question of how indexes are conceived of within the system, how they are maintained.

Data Migration and Graph Rewriting

The concept of data migration in database practice refers to the idea that, after a system architect or data architect has designed some database, and a company has populated that database with millions of records, at the expense of millions of dollars, it is realized that the original design of the database schemas are inadequate to meet future business needs. The data needs to be migrated to a new set of schemas.

It is often the case that the data is so voluminous, and so valuable, that it will take months of system architect effort and weeks of management review before one can press the button and cross one's fingers, hoping that nothing is lost or corrupted. Mistakes are not hard to make: some important PRIMARY KEY - FOREIGN KEY constraint might be accidentally dropped. Maybe some column will be forgotten about. Maybe normalization will be broken, resulting in duplicated data appearing in multiple tables, with no means of keeping that duplicated data in sync, leading to long-term

differs so dramatically from everything else out there.

data rot. It sure would be nice to do data migrations in a less time-consuming, less error prone fashion. This requires a principled approach; but how?

Well, what is data migration really, at the most abstract level? It is a graph rewrite. The initial database design should be understood to be a graph, with the various key constraints between tables being the directed edges of the graph. The revised database design, with its new table schemas, is a different graph. The goal of the data migration is to rewrite the first graph into the second. Data migration is in fact graph rewriting, but narrowly conceived.

Looked at this way, the situation is jarring. In the former case, data migration is a difficult, time-consuming manual process playing out at a human timescale. In the later case, any given graph rewriting system or term rewriting system is expected to do it's operations in milliseconds.[28] How is it these two scales are so different?

The idea of doing data migrations in hours instead of days is radical; the idea of doing it automatically at millisecond speeds is revolutionary. Moving forward in the domain of knowledge representation requires bridging this gap. If one has data represented one way, one must be able to efficiently, rapidly and easily transform it into a different representation. Even more: for AGI, such rewrites must be fully automated and automatic.

To make data migration easy, one needs to be able to represent the graph rewrites within the system itself. Later sections will describe a metaquery system that accomplishes this. This is in contrast to the situation with SQL. The graph that is the schema of an SQL database is not itself stored in a table, nor is it queriable with SQL commands. When a DBA hand-crafts an SQL expression to perform a graph rewrite, they do not store that expression within the database itself; it exists externally, in a distinct location. By working with metagraphs, it turns out that it is not hard to craft a query system in which the queries themselves live as data within the database.

Graph Rewriting and Indexes

The second part of automating rewrites of knowledge representation schemas is to provide a principled approach. One such principled approach can be found in the Functorial Query Language (FQL).[3] This theoretical development has a working open-source implementation called Categorical Query Language (CQL).[29] The approach is founded on category theory,[30] and so the remainder of this section will switch to that language. The reader unfamiliar with category theory is encouraged to learn it.[31]

Basic definitions are in order. A database schema $\mathscr{C}$ is a category. The objects of that category are the tables; the morphisms of that category are the indexes.[32] For

[28] See, for example, Maude,[4] a term rewriting system that stands on it's own, disassociated from any other processing system.

[29] See `https://github.com/CategoricalData` and `https://www.categoricaldata.net/`.

[30] See Wikipedia, `https://en.wikipedia.org/wiki/Category_theory`.

[31] A reasonable introduction, adapted for software developers, is "Category Theory for Programmers", Bartosz Milewski (2019), available at `https://github.com/hmemcpy/milewski-ctfp-pdf`.

[32] More precisely, the category is the database schema; see Wikipedia `https://en.wikipedia.org/wiki/Database_schema`. This includes not just the tables and the indexes, but also all of the triggers, stored procedures, queues, views and other elements. For simplicity, we consider only the tables as

example, given any particular row in some table, the index provides a map from that row to some other row in some other table. It provides the morphism between them. The morphisms are pointer-chases, in that both are composable: given some row, one can chase the pointer to some other row in some other table, and then repeat this again.

An instance of a database is a functor $\mathscr{C} \to \mathbf{Set}$. That is, the database schema $\mathscr{C}$ itself is just the collection of empty tables, and the indexes between them. To populate a database with data, one must specify what goes into it. For example, given a table of employees, there is a corresponding set of employees that go into that table. The mapping is from $\mathscr{C}$ to **Set** and not the other way around, because it is "forgetful": the morphisms in **Set** are functions from one set to another, but they do not describe how the individual elements of the set are to be paired up: those relationships are forgotten. All that remains of the database index morphism is the idea that it connected one set to another, without specifying how the elements of the set are connected.[33]

Two different database instances I and J (for the same schema $\mathscr{C}$) differ by the addition, deletion or modification of rows in various tables. The relationship between them can be phrased as a homomorphism: it maps tables to tables (obviously) and rows to rows. Deleted rows are mapped to the "empty row", and modified rows to their modifications. Added rows are not part of the homomorphism. Since the instances I and J are both functors $\mathscr{C} \to \mathbf{Set}$, the homomorphism between them turns out to be a "natural transformation" $I \Rightarrow J$ in that it obeys all of the requirements for being one.[34] This allows the category $\mathscr{C}$-**Inst** to be defined, the objects of which are the instances I, and the morphisms are the natural transformations $I \Rightarrow J$.

A schema mapping is a functor $F : \mathscr{C} \to \mathscr{D}$ from a database schema $\mathscr{C}$ to $\mathscr{D}$. To be a functor, it must provide maps between the objects of each: $\mathit{tables}(\mathscr{C}) \to \mathit{tables}(\mathscr{D})$ and between the morphisms of each: $\mathit{indexes}(\mathscr{C}) \to \mathit{index\text{-}paths}(\mathscr{D})$. A data migration is then a lifting of this functor F to a functor on $\mathscr{C}$-**Inst** to $\mathscr{D}$-**Inst**. The FQL paper points out there is not just one such lifting, but three: a contravariant functor Δ_F and it's left and right adjoints Σ_F and Π_f. The contravariant functor Δ_F is defined by composition with a particular database instance:

$$\begin{aligned} \Delta_F : &\mathscr{C}\text{-}\mathbf{Inst} \to \mathscr{D}\text{-}\mathbf{Inst} \\ &I \mapsto \Delta_F I = I \circ F \end{aligned}$$

where a specific database instance

$$I : \mathscr{C} \to \mathbf{Set}$$

is mapped to

$$\Delta_F I : F(\mathscr{C}) \to \mathbf{Set}$$

where (of course, by definition) $\mathscr{D} = F(\mathscr{C})$, and so one can write

$$\Delta_F I : \mathscr{D} \to \mathbf{Set}$$

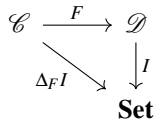
---

the objects, and the integrity constraints (the indexes) as the morphisms.

[33] See Wikipedia, `https://en.wikipedia.org/wiki/Category_of_sets` and `https://en.wikipedia.org/wiki/Forgetful_functor`.

[34] See Wikipedia, `https://en.wikipedia.org/wiki/Natural_transformation`. It is a non-trivial and worthwhile exercise to verify that two different database instances, defined as functors, obey the requirements for being related to one-another as a natural transformation.

This is perhaps clearer with a diagram:



Adjoint to the functor Δ_F is the left-adjoint functor $\Sigma_F \dashv \Delta_F$, which acts as giver of an initial mapping, and the right adjoint $\Delta_F \dashv \Pi_F$ which provides a terminal mapping.[35] These two functors then provide an efficient, formulaic way to migrate data from one database schema to another.

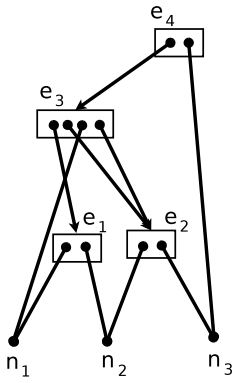
But what does all this abstract nonsense mean, anyway? Informally, all it is saying is "keep your indexes straight, and you'll be OK." Understanding this is best served by an example.

A Data Migration Example

TODO: provide worked example. Conclude: its still tables, in the end.

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,

[35] See Wikipedia, `https://en.wikipedia.org/wiki/Adjoint_functors`. See also `https://en.wikipedia.org/wiki/Initial_and_terminal_objects`.

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

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

Alternately, in Unix directory style (URLs), with slashes. Each e_k and its trailing slash denotes a subdirectory. Each n_k denoting a file: the terminal leaf in the tree. Observe that the same file appears in multiple locations in the directory tree. This is accomplished with either hard or soft links ('`man ln`'). It is somewhat uncommon for ordinary users to do this, but is a widespread technique used in package installs.

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

Either of these requires understanding indentation. 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"
   }
}
```

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,[36] 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.

S-expressions

Let return to the string expression $(e_4 : (e_3 : (e_1 : n_1, n_2), (e_2 : n_2, n_3), n_1, (e_2 : n_2, n_3)), n_3)$. What's wrong with just writing it as a basic s-expression? It appears to be entirely unambiguous to just drop the colon and the comma. It becomes:

```
(e4 (e3 (e1 n1 n2) (e2 n2 n3) n1 (e2 n2 n3)) n3)
```

Can't get any shorter than that.

This is not exactly an 'ordinary' s-expression, though. Key differences:

- Every open-paren must be followed by a link '`ek`': this is what a meta-edge must necessarily be.
- Every node is globally unique. That is, each time the node '`n2`' appears in this expression, it is exactly the same '`n2`', instead of being a different instance. Referring back to the metagraph node table (on page 17), we see that there is only one true instance of '`n2`': it just has one set of attributes on it. When parsing the s-expression, whichever apparent copy of '`n2`' that we grab hold of, the attributes attached to it will be the same.
- Every link is globally unique. For example, the sub-expression '(e2 n2 n3)' appears twice; in both cases, it is exactly the same subexpression. This is clear by referring to the metagraph link table (on page 18): there is really just one copy.

We conclude that s-expressions offer a marvelous compact string representation for metagraphs. Yet one should not be mislead; these are not ordinary s-expressions. This will be further amplified below, when considering queries performed over s-expressions.[37]

[36] See `https://tools.ietf.org/id/draft-pbryan-zyp-json-ref-03.html` and `https://redoc.ly/docs/resources/ref-guide/`

[37] The word "representation" is being used here in the formal sense. For example, in group theory, one

UUID's

Because each node and link is globally unique, it is very tempting, at this point, to say "oh, hey, 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 for 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.

Just how big is a 64-bit hash, vs. the expression it is standing in for? Well, 64-bits is 8 bytes. Suppose that the data that one was storing consisted of English words (there are about a million of them, if you include geographical place names, product and corporate names, historical events and a smattering of Latin and foreign loan-words.) The average length of these is about six letters, thus six byes; seven if including the null terminator. How about the names of genes and proteins? These are not particularly lengthy either. "Ah, but tree structure!" you might think. All those parens, they need to be counted too! If you dump those plain-ASCII (plain-UTF8) s-expressions to a file, and then run gzip on it, or better yet, bzip2 or 7z, you find that those raw string s-expressions compress very well, outperforming by a wide margin the 64-bit UUID hash.

"But", you say, "we can't compress; the data lives in memory, not a file!" Ah, yes, this is true. The in-RAM representation is explosively larger than the string s-expression form. Reviewing the two tables on page 17, each appearance of a node or link in those tables must be a (64-bit) pointer. The example metagraph being worked here is two levels deep, and is quite 'small'. Yet, counting the edges in the drawing of it, there are ten edges. Each edge is bidirectional: there are pointers going both ways; those 20 pointers are required, or $160 = 20 \times 8$ bytes. Plus three bytes each for the null terminated strings 'n1', 'n2' ... 'e1', 'e2' for a total of $160 + 24 = 184$ bytes minimum. The string-length of the string '`(e4 (e3 (e1 n1 n2) (e2 n2 n3) n1 (e2 n2 n3)) n3)`' is 48 characters: 48 bytes. The s-expression representation is four times more compact than the in-RAM representation!

---

can talk about a specific group, as a "thing in itself". One can also have a "matrix representation of a group", which is a collection of matrices that behave the same way as the group. The matrix representation will also have additional properties and relationships that the group does not have. These are termed "accidental relationships". Likewise here: there is the metagraph as the "thing in itself", and the representation of it as an s-expression. The s-expression has additional aspects to it that the metagraph does not have. Also, the s-expression can also represent other things, completely different things, that are not metagraphs! Similar remarks apply to the graphical drawings: they are not the "metagraph in itself", they are a visual representation of it. See Wikipedia, `https://en.wikipedia.org/wiki/Representation_(mathematics)`.

Finally, there is the problem of the UUID-to-meta-tree mapping. Where is it stored? New meta-trees could be added weeks or months later. Is there some UUID-to-meta-tree mapping service, live, available on-line 24x7? What happens if it goes down? How is it kept unique? With locks? How many UUID's per second can be issued? What happens if it gets corrupted? Who has spare copies? Do byzantine generals show up and ruin the day? How much storage does this all require?

To conclude: having hashes around can be useful, as hashes are needed for hash tables. However, conceiving of them as if they were truly universally-unique ID's is problematic in a dozen different ways. By contrast, each and every s-expression really really is globally unique. There is only one of any given kind! No centralized authority was needed to issue this unique string! Anyone can mint it at any time, at very low cost! If forgotten, it can be recreated! And it's ASCII representation, compressed with bog-standard compression routines, is formidably tiny. UUID's are horrible; s-expressions are smart.

Insertion, Deletion and Mutability

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! But where is this stored?

One concludes that there is really only one option: there has to be a mutable, top-level index that holds all of the meta-trees in the metagraph. In a sense, it is the one link to rule them all. Put differently, a metagraph store is, *de facto*, just a store for one single mutable meta-tree.

The word 'mutable' was used several times in the last few sentences. This is important. For all other meta-trees, it is not only convenient to treat them as if they were immutable, it becomes a necessity. Returning to the example meta-tree:

```
(e4 (e3 (e1 n1 n2) (e2 n2 n3) n1 (e2 n2 n3)) n3)
```

Suppose one wishes to change "just one" of the 'n3's into an 'n5'. But which one? They are, after all, all the same 'n3', so if edited, they all change together, atomically. Suppose now that this meta-tree is a subtree of something larger and more complex. Performing an edit, changing the 'n3' into an 'n5' in this meta-tree requires examining the incoming set of this tree, and determining what to do in each case. This might be a non-trivial decision procedure. Should each of these larger trees get the new, modified version, or should they keep the original unmodified version? If the latter, then the unmodified version becomes *de facto* immutable, and the new, modified version just becomes a completely new meta-tree.

So immutable meta-trees are quite all right. Besides avoiding the decision problem of what to do with the incoming set, there are some much more mundane advantages. Immutable meta-trees can be traversed lock-free; there is no concern that one thread is traversing it even as another thread is altering it. The importance of having small structures being lock-free cannot be overstated.

This all changes when one considers the top-most, master index. It *must* be mutable. It *must* be possible to add new trees, and remove stale ones from this master index.

Otherwise, it cannot be called a database!

Yet, oddly, this top-level index/table looks exactly like any other metatree. Yes, its large: it may have many millions of (immutable) subtrees in it. But it itself is just another tree, and there is no particular reason why it cannot be the subtree of yet other trees. The only difference is that it is mutable.

Thus, we've identified two fundamentally different utilities, serving different purposes, yet both having the same structural form. The mutable form is a 'database'. The immutable form is the 'data'. In all other respects, they are the same.

Within the OpenCog system, the mutable form has been historically called the 'AtomSpace'. It can be stacked and nested and used to form diamond-shaped inheritance diagrams, however one wishes. The immutable Nodes and Links have been called 'Atoms', as a stand-in term for something that can be either a Node or a Link.

Metagraph mutability vs. SQL table mutability

There are two ways to delete a row from an SQL table. Either one already knows the "*PRIMARY KEY*" of that row, and one asks for that key to be deleted, or one does not. If one does not know, then one has to "*SELECT key FROM employees WHERE name='Agent Smith';*" and present the resulting key to be deleted. Yes, of course, this can be done with a single statement: "*DELETE FROM employees WHERE name='Agent Smith';*" but this only hides the fact that, without the key, nothing can be done.

In SQL, effectively all tables have a primary key: it is the master key for each row. For the discussion above, it is effectively the UUID. As the *de facto* UUID of SQL systems, it suffers from all the UUID drawbacks previously reviewed. SQL databases are notoriously hard to shard across multiple network nodes. Consistency and atomic updates are hard: there is effectively one centralized bottleneck. This is the foundation stone on which the ACID vs. BASE debates are built.

How does this compare to the one mutable index of all immutable meta-trees? Well, there are at least three 'obvious' ways in which the deletion of a meta-tree could be implemented.

- If the master index is a hash table or a tree of raw memory pointers, and one has the raw pointer to the memory to be deleted, the deletion can be immediately and directly performed. This is analogous to already knowing the primary key of an SQL table row.

- If the master index is a hash table or a tree of abstract indicators, and one has the abstract indicator of the meta-tree to be deleted, the deletion can be immediately and directly performed. Unlike the case of UUID's, there are abstract indicators that can be computed uniquely and in a relatively rapid fashion. An example of this is the Merkle tree. It is 'decentralized', in the sense that, no matter who has a copy of a given meta-tree, or where it might be located, everyone will agree as to what it's Merkle tree hash will be. It is not 'unique', in the sense that there might be hash collisions. However, for practical use in the database, being collision-free is not so important; being computable in a decentralized fashion

is. It avoids the UUID/primary-key pitfall. This certainly turns the ACID vs. BASE debate a bit on it's side. The current actual OpenCog AtomSpace in-RAM implementation uses Merkle trees.

- If the master index is an ordered key-value-store (OKVS)[38] of s-expressions, and one has the s-expression corresponding to the meta-tree to be deleted, the deletion can be immediately and directly performed. But it is already apparent that it is easy and cheap to compute (in a decentralized fashion) the s-expression corresponding to a meta-tree. This is effectively a variant of the above, where the "abstract indicator" is the s-exp, instead of the Merkle tree. Given the earlier discussion concerning the size of s-expressions vs. the size of pointers, this is certainly a viable storage format for the in-RAM metagraph database. That it is not used by the current in-RAM AtomSpace is largely a historical accident. However, one of several of the current file-storage backends to the AtomSpace is RocksDB. That backend *does* use s-expressions as the "primary key", and RocksDB itself is a kind of OKVS.

Metagraph insertion can be compared to SQL table insertion along the same lines.

As should be clear, there is not really all that much difference here, at least conceptually. The biggest and by far the most apparent difference is that SQL explicitly and overtly exposes UUIDs in the form of primary keys, thus dragging all that baggage along with it. By contrast, a metagraph store can keep all of this hidden, and has multiple implementation options for the analogous concept of a primary key: either Merkle trees, or s-expressions, or possibly something else. From the implementers point of view, this is nice: there are various possible implementations (and they are all hidden from the user! No backwards-compat concerns!)

This is also nice from the user's point of view: 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!

Metagraph mutability vs. Graph databases

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 at the same time that one is inserting data.

The answer to be supplied here is utterly opaque. This section is short because the problem is a bit mind-boggling. To conclude: we seem to have discovered, quite accidentally, with no explicit intent, that hypergraphs offer an elegant property that graph databases lack.

[38] See `https://en.wikipedia.org/wiki/Ordered_Key-Value_Store`

Query Languages and Graph Traversal

The majority of popular query languages (QL) are modeled on SQL, taking SQL as not just as an inspiration, but rather directly borrowing (stealing) most of its keywords and syntax. This is good, in that it makes it easy for users who are already familiar with SQL to learn the new system. This is good, in that it acts as a guidepost to the developer of the NewQL: if things get confusing, just take a look at how SQL did it, and make NewQL do it more-or-less the same way.

This is bad, in that, when the QL is to be applied to a new domain, such as graphs, hypergraphs or metagraphs, the mind-set that was suitable for table queries might not be appropriate for graphs. But this is not obvious: just half an hour with GraphQL documentation and examples will easily leave you with the impression that you can do anything you want with GraphQL. Got a graph? GraphQL can traverse it! That there might be some other way, maybe even a better way, never really enters your mind.[39]

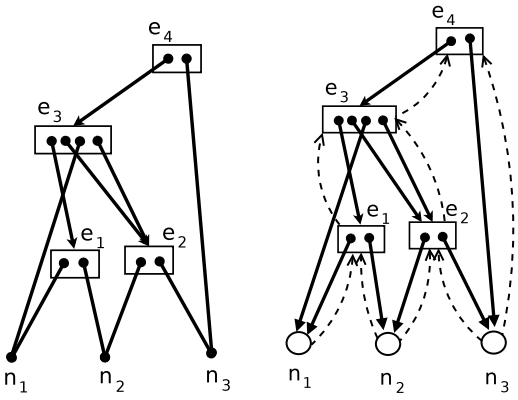
There is no doubt that the mathematical theory of relational algebras is the "correct", or at least the "best" theory for tables and the records stored in them. It's overwhelmingly dominant, and there does not appear to be any reason to question it. Likewise, the manifestation of relational algebras as SQL has the same effect: it works really really well for records and tables, and there is no compelling reason to replace it with something else. (A little voice asks: "if that's the case, then why graphs?") The success of this theory has the effect of shutting down any thought process that might lead elsewhere.

The aim of this section is to make this thought process overt, and show how there might be QL's that are not SQL clones. It begins by (once again) reviewing the actual structure of metagraphs, and then examines how they compare to conventional table queries.

Weak Pointers

The example metatree diagram, as originally drawn (on the left) is misleading as to how it is actually represented in memory. If one examines the metagraph link table (on page 18) and includes arrows for both the incoming set and the outgoing list, one gets the diagram on the right:

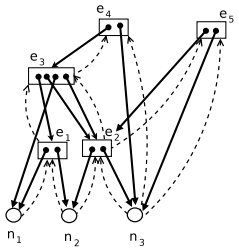
[39] This is a reply to the Sapir–Whorf Hypothesis in the strong affirmative. The Sapir–Whorf hypothesis is briefly worded as "language limits thought." Originally, the language was natural language, say English or Chinese. The hypothesis seems a bit absurd: if I speak English or Chinese, why *wouldn't* I be able to think of anything (if I'm smart enough)? What's limiting about language? It's hard to imagine what one wouldn't be able think of. In the present case, the language is SQL, and if you are a programmer/DBA who thinks in terms of SQL, it is very hard to imagine what kind of queries one could perform that *don't* look like SQL. This is the topic of this section: exposing the limitations of SQL-thinking.



The dotted arrows are back-pointers from lower levels to higher levels: they are the incoming set pointers, in contrast to the solid arrows, which show the outgoing links. They are almost in one-to-one correspondence, but not quite: notice there is only one dotted arrow from e_2 to e_3, although there are two distinct downward links.

The back-pointers are needed to make the graph fully traversable; that is the topic of this section. The back-pointers do eat up RAM; but this RAM usage has already been counted, above. Here, it is only the diagram that has been corrected to more appropriately show the actual situation. In actual implementations, the dotted arrows are weak pointers. This is required for memory management, whether through garage collection (GC), or through reference counting (RC) with smart pointers. Both kinds of systems (GC and RC) can only work with DAG's, and do not tolerate loops, as these cannot be freed. Yet programmers need to sometimes create loops of pointers; weak pointers are provided for this use. That is what is shown here.

It is natural to use weak pointers for the incoming set, and regular pointers for the outgoing list. This is because, as observed earlier, metatrees must be immutable after they have been created. They cannot be edited; they can only be deleted. There is, however, one minor point of confusion, illustrated in the diagram below.



This shows that a new link e_5 has been added to the previous diagram. This is *not* an edit! The earlier tree, rooted at e_4, is completely unchanged. None of the regular pointers have changed. Two of the atoms e_2 and n_3 have enlarged incoming sets: they have gained a weak pointer each. That the original metatree has not changed can be most easily observed by the fact that it's corresponding s-expression is the same as before:

```
(e4 (e3 (e1 n1 n2) (e2 n2 n3) n1 (e2 n2 n3)) n3)
```

The diagram above shows two metatrees. The second one is

```
(e5 (e2 n2 n3) n3)
```

The new root e_5 points to two existing metatrees: e_2 and n_3; neither has been modified. The point here is that immutability does not conflict with creation and deletion. Creation and deletion do alter the weak pointers/incoming sets, but not the metagraph itself.

Queries and Tables

What do weak incoming sets have to do with queries? That they allow complete traversal of the metagraph is intuitively obvious: there are plenty-enough arrows (both regular and weak pointers) to be able to take walks from any Atom to any other. But are these really needed? The aim here is to show that the answer is 'yes', they really are.

As an initial example, consider the metatree without the incoming-set pointers. Consider a tree-walk starting at e_4. Using conventional recursion techniques, it is easy to recurse (left to right) down to e_3 then e_2 and then n_1. Implemented as a stack machine, the next step is to pop, and then visit n_2. The next stack pop brings us back to e_3 and traversal proceeds as normal. Nowhere in this stack machine recursion is it possible to visit e_5. If one has started at e_4, one cannot get to e_5. Worse: perhaps one's starting point was n_1. From there, one cannot get to anywhere, at least, not when there aren't any incoming-set pointers (the dotted lines).

But is this relevant in any real-world application? Consider the following interpretation of the above diagram, where the Nodes are individuals, taking school classes and working part-time jobs:

Atom	Interpretation
n_1	Joe
n_2	Mary
n_3	Rachel
e_1	Intro Computer Science
e_2	Intro Beauty and Hair Care
e_3	Dean of Students
e_4	Whatsamatta University
e_5	Ayurvedic School of Cosmetology II
$e_3 - n_1$	Joe assists in the Dean's Office
$e_5 - n_3$	Rachel is a TA in cosmetology

A natural question to pose in this interpretation is "what schools does Rachel attend?". Starting at n_3, one can trace upwards to find that Rachel attends both Whatsamatta U. and ASCII.

How might this have worked in conventional SQL? Well, for starters, we need something like this:

```
CREATE TABLE Schools (name TEXT, uuid INT PRIMARY KEY);
```

```
INSERT INTO Schools ... ;
CREATE TABLE Courses (name TEXT, school_uuid INT FOREIGN KEY, uuid INT PRIMARY KEY);
CREATE TABLE Students (name TEXT, course_id INT FOREIGN KEY, student_id INT PRIMARY KE
```

One continues in this fashion, creating a few more tables. Denoting that Joe assists in the Dean's Office and that Rachel assists in the School of Cosmetology poses a bit of a challenge. Perhaps this might do?

```
CREATE TABLE Employment (name TEXT, employer TEXT, taxpayer_id INT PRIMARY KEY);
```

The need to denote employment adds a challenge to representing everything properly with tables, but is not relevant to the query about the schools that Rachel attends. Mostly, we just observe that, even for this rather simple graph, there already is a profusion of tables with some rather complex relationships between one-another. Textbooks insist that normalization is the correct thing to do; students often find that normalization is just plain hard.

How does the query work?

```
SELECT Schools.name FROM Schools WHERE
    Students.name = 'Rachel' AND
    Students.course_id = Courses.uuid AND
    Courses.school_uuid = Schools.uuid;
```

The above has the form of an explicit path walk. It starts at n_3 and explicitly names a path that takes it to e_2 and then ... Well, obviously, we've left out a few details, in that e_2 is jointly managed both by ASCII and by the Dean of WU. Representing this properly requires a few more tables, and an additional clause in the `SELECT` statement.[40]

This query works precisely because the DBA who is creating the query has a very concrete grasp of all of the tables involved, and is able to thread an explicit path between the tables. The reason that the SQL database engine can execute the query is because the DBA explicitly inserted `FOREIGN KEY` constraints into appropriate locations. These foreign keys have the side effect of defining an index that manifests the dashed arrows in the metagraph diagram. The foreign keys are the dashed arrows. Not all of them; just some of them, only the ones that the DBA saw fit to add during the table design. Just like the dashed weak pointers allowed the metatree to be traversed upwards, so also the foreign keys allow the tables to be traversed "upwards".[41]

Query planning

Query planning concerns the topic of how the database engine will actually perform the query. The above example was written in a fashion that highlights an obviously efficient

[40] We've left out more than a few details. There is the concept of '`JOIN`', and its variants of '`INNER JOIN`', '`OUTER JOIN`' and so on, all of which exist to allow the DBA to more carefully control and craft the nature of the graph walk needed, and how the data flows through the query, so that the query will run in an efficient, performant manner. The DBA really needs to actually think of what the query is doing (and to know the concepts, and to consult the documentation). I'd like to suggest that programming in SQL is like programming in assembly code, but for data. Perhaps it's time for a higher-level query language that abstracts away these details.

[41] This is the point of the Meijer and Bierman result.[1] In a noSQL database, the arrows point in the opposite direction. In either case, useful database queries require arrow-chasing. Turns out that both SQL and noSQL databases are, under the covers, graph databases with awkward API's.

and easy way to run the query. Start at the join 'Students.name = 'Rachel'' and trace the dashed arrows upwards, to immediately arrive at the desired answer.

Of course, there is no guarantee that the SQL engine will actually do this. It might decide to start with 'TABLE Schools', examine each school in turn, and then trace downwards from the top, until it eventually reaches Rachel, and is able to satisfy this final constraint. Table joins are a constraint satisfaction problem. Solving constraint satisfaction problems (CSP) is non-trivial;[42] there is a vast literature devoted to this, both for the case of general CSP, and to the specific case of query planning.

Why would an SQL engine run this query in an unexpected or inefficient way? Well, the query planner may have decided that it is too expensive to create the dashed-arrow indexes. Without the dashed arrows, one obviously has to start at the top. Why would it decide that the dashed arrows are too expensive? Well, it might have looked at all the tables, noted that they are all very small, and that a direct downward-recursive query could be performed rapidly. Why splurge on indexes if they are not needed?

There are also more mundane reasons to run the query top-down: the particular SQL engine may not even have much of a query planner. It might lack the sophistication to do anything fancy.

Example: OGRE QL

A worthwhile example to contemplate is the query language for OGRE[2].[43] The influence of SQL in this query language is clear. Interestingly, OGRE was designed to be a database for s-expressions, and, as we've noted, s-expressions seem to make for an almost ideal representation for metatrees. So what's the catch?

Well, the OGRE s-expressions are just that: strings. They are not metatrees; they are not interpreted as metatrees. They are interpreted only as ordinary, conventional abstract syntax trees. Because of this, there is no particularly compelling reason to automatically manage the incoming sets for these trees. Or rather, there is no particularly compelling reason, until one considers the implementation of the 'JOIN' keyword. In OGRE, there are no overt primary keys, and no foreign keys, so these need to be generated implicitly, as needed, to perform a query that joins together multiple s-expressions.

Does OGRE actually auto-create these primary-key/foreign-key relations, under the covers? Without looking at the implementation, it is hard to guess, but there are a few hints. OGRE works as follows. Each s-expression has the form

```
( <attribute-name> <v1> <v2> ... <vM> )
```

That is, the very first word appearing after the opening parenthesis has a special meaning. This is entirely conventional: in LISP/Scheme, the first word appearing after an opening parenthesis also has a very special significance. It is an operator. For our metatrees written as s-expressions, that first word is necessarily a Link. And so in OGRE, the first word is called the "name of a proposition"; the subsequent words are the object (or subject) of the proposition.[44]

[42] See https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

[43] Documentation for it is currently located at this URL: http://binaryanalysisplatform.github.io/bap/api/odoc/ogre/Ogre/Query/index.html

[44] The word "proposition" is no accident. It is meant to make you think of propositional logic. This point

As noted in the introduction to this text, an s-expression such as

```
(student (name Joe) (gpa 3.5))
```

can be shortened to

```
(student Joe 3.5)
```

provided that one has declared a "predicate"

```
(declare student (name str) (gpa float))
```

This last statement resembles a conventional table-database table declaration. The resemblance is more than superficial. The explicit BNF syntax for the declaration is

```
declaration ::= ( declare <attribute-name> <field> <field> ... )
field ::= ( <field-name> <field-type> )
field-type ::= int | str | bool | float
```

The field type is explicitly a concrete type.[45]

The need for explicit declarations, and the explicit identification of concrete types for fields, suggests that, under the covers, OGRE is building up all the machinery needed to perform table joins. Presumably this means the creation of the dashed-arrow incoming sets that are needed for upwards traversals. Or perhaps not: maybe the OGRE query planner always initiates queries top-down, starting at the uppermost declaration, and running a stack machine to perform recursive joins downwards. If one has well-defined tables, a top-down query would seem like a viable approach.

Note, by the way, that the declaration statement is of fixed length, and is *NOT* variadic! The declaration statement really is defining fixed-length records which can be placed into a completely conventional row-table. This is perhaps why the OGRE query language looks a lot like SQL: it has not departed from the table paradigm. It has merely slapped on a prettier API, prettier in the sense that the primary key and foreign key constraints have disappeared into the woodwork.

As a meta-comment about table declarations and queries: this all works fine, as long as one has a small number of declarations. A dozen, a few dozen, less than a hundred. Pondering the situation of a million different table declarations is .. imponderable, if one comes from an SQL background. Because: oh good lord, how could one ever write meaningful queries if there are a million tables? Can't even imagine. Why, one would need to write meta-queries to query tables, and perhaps query meta-tables of tables. If only there was a meta-query language to query meta-tables in recursively-organized meta-trees... oh wait, what?

We'll get back to you on that one.

In the meanwhile, it is important to observe that OGRE was designed to be a plugin for OCaML. A significant part of the query apparatus is defined in terms of OCaML

---

will be returned to when we look at ProLog. It is notable that there is confusion between subject and object; that these are taken to be synonymous. In the general sheaf theory, subjects and objects are not synonymous; they correspond to different connector types. But this is a topic outside of the scope of the present text.

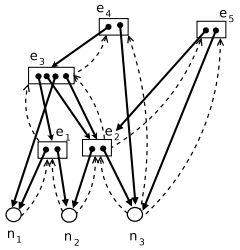
[45]This is a good time to start talking about type theory. The correct handling of metatrees requires recognizing that the first word following the open parenthesis, the so-called Link, is a *de facto* type-theoretical type. Perhaps there is some way for it to not be a type, but, as Sapir–Whorf suggests, it proves to be difficult to imagine how it could be anything other than a type.

and not in terms of s-expressions. Portions of the query are explicitly typed; many of the type declarations are functional-programming function arrows (in type theory, these go by the name of "function type", "arrow type" or "exponential", the later name harking back to size of the codomain of set-theoretic functions).[46] The query language is usable only if one is writing OCaML.

Perhaps the most innovative aspect of the OGRE query language is that it is wrapped in a Kleisi triple (a Monad). Somewhat oddly, though, that monad seems to reinvent some subset of LISP, or at least of the SRFI-1 list iterators. This is perhaps not entirely unexpected: s-expressions do look like lists, and it is natural to want to do list operations on them.

A Meta-Query Language

How else could the query about the schools that Rachel attends could be formulated? Returning to the example diagram



and once again noting that the presence of the dotted arrows allows rapid graph traversal starting from any Atom, we can offer the following as a possibility:

```
(Meet (Variable "?what-school?")
   (Present
      (Link (Variable "?what-school?") (Variable "?course?"))
      (Link (Variable "?course?") (Node "Rachel"))))
```

The reading of this query is as follows:

- '`Meet`' is a special keyword defining the query.[47]
- '`Variable`' is a special keyword, that, when it follows a '`Meet`', it indicates what is to be returned. It has other (related) meanings in other contexts.
- '`Present`' is a special keyword, asking that all of the clauses must explicitly appear in the database. They must be matched, as patterns in a pattern matcher. It is an unordered Link; the order of the clauses in this Link carries no significance.

[46] See `https://en.wikipedia.org/wiki/Function_type`

[47] The name 'Meet' comes from the mathematical concept of meets and joins on a lattice. See `https://en.wikipedia.org/wiki/Join_and_meet`. The 'Present' link is a listing of several points in a lattice, all of which must be met by any solution to the query.

- The 'Node' and 'Link' keywords are special keywords, explicitly denoting the Nodes and Links about which we've been talking all along.

The intent is that the '(Variable ?course?)' is grounded by e_2, a course Rachel (n_3) is attending. The '(Variable ?what-school?)' is grounded by e_5, the School of Cosmetology.

This should be contrasted with the earlier SQL query:

```
SELECT Schools.name FROM Schools WHERE
    Students.name = 'Rachel' AND
    Students.course_id = Courses.uuid AND
    Schools.uuid = Courses.school_uuid;
```

The most notable difference is that the meta-query does not need to explicitly name the 'Schools' table. Indeed, the metagraph does not have a 'Schools' table, and so it is not nameable (this could be "fixed" by adding more nodes and links to the metagraph, but that is besides the point.)

Another fundamental difference is that SQL embodies an explicit "equational theory", whereas the meta-query does not.[48] The SQL statement has implicit variables in it: the expression 'Students.course_id' is meant to be understood as a variable, ranging over all possible values of 'course_id' in the 'Students' table. The 'Courses.uuid' is another implicit variable. The equals sign is needed, because of the way that the variables appeared implicitly. There is no other way of saying "hey, these two variables are really the same thing", and thus the concept of table joins is born.[49] The meta-query made use of the property of meta-graphs that any given Atom is always the same Atom. That is, the '(Variable "?course?")' is the same Atom in both subtrees. There is no need to use an equals sign to equate these two positions; they are already "the same thing" as metagraphs; they can't not be "the same thing".[50] There is no explicit need for equality.

[48] See https://en.wikipedia.org/wiki/Theory_of_pure_equality and https://en.wikipedia.org/wiki/Equational_logic. In mathematical logic and model theory, the equational theory is the first theory that is more complicated than the "free theory" (see https://en.wikipedia.org/wiki/Free_theory.) In brief: with the free theory, you can freely "write down anything", as long as the parenthesis balance. Both s-expressions and Prolog can be taken to be examples of free theories. The theory of pure equality adds the equals sign, and says "you can write down anything", as long as the parenthesis balance and you also take transitive closures of the things that are equated. Problems of consistency first show up here, as it is possible to write down nonsense and paradoxes using equals signs. Relational algebra (*e.g.* SQL) is the theory where you can "write down anything", but you can also use relations between things. Each relation corresponds to an SQL table. That relation is "true" if the corresponding row exists in the SQL table, else it is "false"; thus relations are sometimes called predicates. A pure relational algebra, without equality, is not quite interesting enough to be practical. As all of these examples show, you need to fold in the theory of equality in order to perform table joins. The equals sign just says "this variable appearing in this predicate is equal to this other variable or constant appearing in some other predicate".

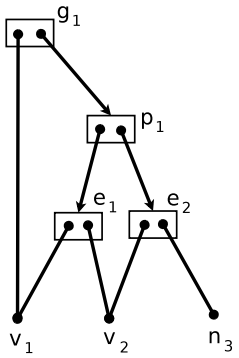
[49] The complexity of table joins is in part a side-effect of the complexity of the theory of equality. You might think that equality seems pretty darned easy; it's just equals signs, right? Contemplating the complexity of SQL table joins, and the complexity of SQL query planners in planning table joins should quickly disabuse you of the idea that "equality is simple". It's hard. If you are still not convinced, review the Wikipedia article on equational logic again: https://en.wikipedia.org/wiki/Equational_logic. It has a number of axioms to make it work, and it is no accident that it was developed by computer scientists who were also known for their contributions to relational algebras and query languages.

[50] Variables, are, however, always named, given a name. It is an open question is to how anonymous (unnamed) variables might be written and used.

A third fundamental difference is that the metaquery did not require any prior table definition. The situation is thus quite different than the aforementioned OGRE QL. Thanks to the implicit presence of the dashed arrows, there is no need to explicitly define indexes (i.e. there is no '`CREATE INDEX ON Students(name);`' which would be needed if we wanted to have fast-running SQL queries on student names. The primary keys, of course, did not need explicit index declarations.)

A fourth fundamental difference is that the metaquery is itself a metatree. It does not live outside of the system of metagraphs. It does have a peculiar syntax and a collection of syntax rules (*i.e.* it must begin with '`Meet`', it must have some variable declarations and use the '`Present`' link, and so on) but that syntax is a sub-language within the language of free s-expressions. The fact that the metaquery is itself a metatree opens the possibility of querying for queries. This may sound utterly absurd, but is in fact very commonplace in chat-bots and rule engines! This is expanded on below.

It is perhaps instructive to draw the metatree diagram for this query:



The reading for this metatree, in terms of its Atoms, is as follows:

Atom	S-expression
v_1	`(Variable "?what-school?")`
v_2	`(Variable "?course?")`
n_3	`(Node "Rachel")`
e_1	`(Link (Variable "?what-school?") (Variable "?course?"))`
e_2	`(Link (Variable "?course?") (Node "Rachel"))`
p_1	`(Present (Link (Variable "?what-school?") (Variable "?course?")) (Link (Variable "?course?") (Node "Rachel")))`
g_1	`(Meet (Variable "?what-school?") (Present (Link (Variable "?what-school?") (Variable "?course?")) (Link (Variable "?course?") (Node "Rachel"))))`

It is not entirely obvious, but performing this query requires solving a subgraph isomorphism problem. Clearly, the query implicitly demands that the variables should be grounded. The variables are to be grounded by Atoms, which may be either Nodes or Links. In the case of Links, these are (by definition) metatrees themselves, and so the grounding matches a variable to a subgraph. The collection of subgraphs must be those specified in the '`Present`' link: *i.e.* they must be isomorphic to it. Thus, the subgraph isomorphism problem.

The careful reader may have noticed that the above query only returns that Rachel is a student of ASCII. The full query, which would also reveal her to be a student of WU, is given below. The comparable SQL statement is considerably more complex and not given.

```
(Meet (Variable "?what-school?")
   (Present
      (Link (Variable "?course?") (Node "Rachel")))
   (Choice
      (Present
         (Link (Variable "?what-school?") (Variable "?course?")))
      (Present
         (Link (Variable "?what-school?") (Variable "?deprtment?"))
         (Link (Variable "?department?") (Variable "?course?")))))
```

The above introduces the 'Choice' Link. For the most part, it can be read as a logical-OR of the two predicates that it wraps. More precisely, it denotes a menu-choice: pick one or pick the other (but not both). Choice commonly appears as a fragment of intuitionist logic and of linear logic.

Metatree Query vs. String Pattern Matching

Since metatrees can be written as s-expression strings, and since metaqueries can be likewise, the question arises: can metaquery evaluation be reduced to (regex) string matching? The initial example certainly feels like this might be possible: the variables are just wild-cards, and everything else is just constant strings that can be matched verbatim. Unfortunately, this is not the case.

This is doubly unfortunate, since pattern matching is well-understood and widely developed. Besides regex and pcre syntax case scheme[51] itself, the 'syntax-case' subset[52] of "hygienic macros" in the scheme programming language provides rudimentary pattern matching and term rewriting. In that direction, most (all?) functional programming languages include pattern matchers as a basic utility.[53] The well-developed ones provide pattern languages that are quite expressive, and the implementations tend to be compact and direct.

(Metatree) querying is not just (string) pattern matching. Key differences are listed below.

- In the s-expression for a metatree, when a sub-expression is repeated, it is understood to be the "same subexpression", *i.e.* the same Atom. There is no such guarantee in a string match: the first occurrence of a subexpression could be matched one way, and a second occurrence a different way, possibly with a conflicting match. Thus, if string pattern matching were to be used, it would need to be enhanced with some sort of conflict-free subterm matching. But how to implement this? That line of thinking seems to lead down the path of reinventing metatrees, but in an application-specific setting.

- Practical metatree matching requires the ability to match into unordered collections (sets). This means that each and every permutation of an unordered set must be attempted during the match. String pattern matchers conventionally cannot do this. When they can, there is often no ability to handle nested unordered sets, subterms of which might need to be identical in the aforementioned conflict-free fashion. That is, if a variable occurs in multiple places in multiply-nested unordered sets, and it is grounded one way in a specific permutation of one set, it must be grounded exactly the same way in all permutations of all other sets in which it occurs. This is the 'conflict resolution' aspect. How can this be accomplished? Well, one can use either metatrees, or one can use a 'theory of equality'. But, as noted above, the theory of equality provides it's own challenges to implementation: it introduces brand new conflict resolution issues that are difficult to resolve.

- Practical metatree matching needs to provide menu-choice alternatives for subterms: one could match this or this or this. The choice can be buried arbitrarily

[51] Perl-compatible regular expressions. See `http://www.pcre.org/`

[52] See `https://www.gnu.org/software/guile/manual/html_node/Syntax-Case.html`

[53] Examples include Racket: `https://docs.racket-lang.org/reference/match.html` OCaml: `https://ocaml.org/manual/patterns.html` and Scala: `https://docs.scala-lang.org/tour/pattern-matching.html`

deep in the pattern. As a string pattern matcher proceeds left-to-right through the string, and encounters a choice, it must explore each and every choice, individually. That means that it must backtrack if some given choice fails to match. Backtracking means that the pattern matcher must be implemented as a stack machine (because partial matches are held in a score-board, and that score-board must now be placed on a stack). So, two issues here. First: regexes correspond to finite state machines, whereas stack machines correspond to context-free languages. So already, with choice, we've left the (simple, fast, direct) domain of regexes and finite state machines. Second: if the implementation of the pattern matcher is forcing you to think recursively, then why are you still thinking "strings"? Why not think in terms of trees, which is what recursion (and context free languages) really are? Once one has entered that domain, it is easier to conceptualize in terms of graphs, than strings.

- Practical metatree matching requires the ability to quote and unquote significant keywords in the pattern. For example, one might want to search for all expressions that contain the term 'Variable' in it. Obviously, this would have to be quoted (say, as '(Quote (Variable ?x?))') as otherwise, the variable would be interpreted as a variable in the pattern, instead of a constant term to be matched.

- Practical metatree matching requires the ability to match lambda expressions. This requires the ability to find bound variables, and not only treat them as constants, but also to alpha-rewrite them. Consider, for example, the lambda expression '(Lambda (Variable $x) (Plus (Variable $x) (Number 2)))'. This expression is obviously equivalent to '(Lambda (Variable $y) (Plus (Variable $y) (Number 2)))' – the formal name for this kind of equivalence is "alpha equivalence". Any pattern that is searching for lambdas that involve 'Plus' and the number 2 must be able to determine the alpha-equivalence of the match to the pattern. This is no longer a simple string match: the pattern matcher needs to be aware that 'Lambda' is a special keyword that binds its variable declarations. And that bound variables can be alpha-rewritten.

After reading the above, the reader might feel as if there was a bit of a bait-n-switch performed with the pattern matching requirements. Some deeper thinking would have avoided this impression. The overall topic of this text is that metagraphs are suitable for generic knowledge representation. This means representing not only 'simple' graphs and metagraphs, but also quotidian systems like ontologies, phylogenies, first-order logic, Bayesian networks, lambda calculus, probabilistic programs, functional programs, declarative programs, genes, proteins, natural language, visual objects, sounds, 3D spaces, ... As such, a knowledge representation system has to be practical and utilitarian, useful as a tool for the day-to-day knowledge worker. Thus, funny business like quotes and lambdas and alpha-rewrites and unordered sets must necessarily enter into the picture, and be supportable in the query system. Users ask for these things, users want them. You can't say no to users and expect them to come back anyway.

Note, by the way, that the goals of a knowledge representation system resemble the goals of set theory or topos theory for mathematics: generic frameworks on which it is hoped that all of mathematics can be founded. Knowledge representation is likewise

foundational. It is no accident that knowledge representation must borrow or steal ideas from topos theory, ideas such as sheaf theory. Sheaves are actually useful in mathematics. At this time, it is not yet widely understood that sheaves are also useful in knowledge representation. But we wander off-track, here; this is a different topic.

Assorted OpenCog documentation and web pages refer to the query engine as "the pattern matcher". This is misleading, as much of the tech industry understands "pattern matchers" to be simple finite state machines with pattern languages. The actual OpenCog system is as described above, a stack machine capable of recursive multi-component graph walks.

Query Analysis

Graphs may include equality predicates. In rare cases, these can be statically analyzed, without performing a query.

A full-fledged query language can include evaluatable predicates to accept or reject a particular grounding. During the analysis phase, these are set aside into a distinct bucket. They cannot be grounded (by definition), they can only be evaluated after a grounding is found.

Some evaluatable predicates, such as "less than", may cause a graph to fall apart into disjoint, disconnected components that are bridged only by evaluatable terms. During the analysis phase, the connected components must be identified and set aside. During evaluation of the query, each connected component is grounded separately, and then brought together, to evaluate "less than" on the resulting groundings. Evaluatable terms, such as "less than", act as a filter on the Cartesian product of the components. Terms such as logical-or act on the disjoint union of the components. That is, both pi-type and sigma-type arrangements of disjoint components are possible.

A full-fledged query language can include an 'Absent' predicate, indicating one or more clauses that must be absent in the dataset, in order for the query to be satisfied. Such clauses are extracted and set aside during the analysis phase. They resemble evaluatable clauses, but have their own distinct peculiarities.

A full-fledged query language can include an 'Always' predicate, required to be present in not just one grounding, but in every grounding. They resemble evaluatable clauses, but they cannot be evaluated until after all possible groundings have been obtained (They can fail early, though.)

Query Planning

The most basic query planning involves choosing a starting point for a graph walk. Thanks to the incoming set, the query can be started anywhere. However, it is advantageous to start the graph walk at the thinnest, most distant part of the graph, so that it does not need to be regularly revisited.

Sometimes, finding a starting point this way is impossible, because the query consists entirely of variables. In other cases, a query may consist entirely of evaluatable terms, which, by definition, are not groundable, but only evaluatable. These cases must be dealt with.

Queries may consist of multiple disconnected components; these must be individually grounded, and then reassembled with pi-type or sigma-type bridges between them.

It is advantageous to borrow some ideas from SAT solving, and prune simple trees from the query, before performing an exhaustive search of a multiply-connected tight nucleus. The pruned trees can be re-attached later. Equivalently, they might only need to be pattern-matched once. In this case, pruning is equivalent to maintaining a cache of pattern matches achieved earlier during traversal. (That is, a recursive graph walk will typically revisit certain parts of the graph. But if those parts have already been grounded, they do not need to be revisited again; it is sufficient to have a cache of the groundings.)

The OpenCog query engine makes use of these query optimizations.

Attributes (Tags)

This text opened with an extended discussion of attributes, and about how attributes are attached to the nodes and vertexes of a graph. This is still the case, still a desirable thing to do, when working with metagraphs. "But why?" you may wonder, "why can't everything be encoded as as metagraph?"

Well, of course, everything can be encoded as a metagraph. However, there's a cost; several costs, actually. One cost is that metatrees are immutable. If one has a list of 100 floating point numbers, and wishes to change the 42nd one, it seems like overkill to be blocked from doing so by immutability. One might like to have something mutable, for this kind of a low-level vector. Another cost is the weak pointer (the dashed arrow). There is no particular need to maintain an incoming set for each floating point in a list. It is unlikely that two different Links will share the same floating point number (to all of it's glorious 18 decimal places!? Really?) It is unlikely that there will ever be any need for a query to traverse backwards (through the incoming set) of that float. Traversing forward into the vector of floats is plenty enough, if the only query decision to be performed is greater-then, less-than on some arithmetic ops on those numbers. Saving some RAM by not storing the weak pointers is a laudable thing do to.

If the attributes are stored as key-value pairs (KVP), then the attributes on any given Atom can be understood conceptually as a key-value database per Atom. This might feel like overkill, but it isn't really any different from graph databases in general. If a vertex or an edge of a graph has some attributes attached to it, then those attributes can be thought of as residing in a database specific to that edge or vertex.

In the present case, there is one fun little twist: if the KVP's are hierarchical, *i.e.* a tree, then they can be thought of as a special case of a metatree (as, indeed, trees are special cases of metatrees), but it is a metatree that is shorn of it's weak back-pointers, and thus can be mutable. One saves RAM and regains the mutability that was lost. The price to be paid is that such attributes are no longer searchable. (More precisely, they are no longer searchable except by brute force exhaustive search. The weak pointers acted as local indexes; without the ability to do an index lookup, brute force exhaustive search is all that one has left.)

The OpenCog AtomSpace supports attributes in the form of KVP trees. It has made one unfortunate naming choice for these attribute trees: they got called "Values" with a capital V. This naming choice forces all sorts of contortions when one wants to talk

about ordinary values with a lower-case v. The naming choice was arrived at under the influence of model theory, where a clear distinction is made between sentences of a language and their valuation. In model theory, and in logic, sentences are assigned "truth values". Historically, these are crisp true/false values. But in Bayesian belief networks and in Markov logic networks, these valuations are given floating-point probabilities as their truth values. On OpenCog, these were historically called TruthValues, written in CamelCase. When generalized to arbitrary KVP valuations, they become Values. A better name might have been Attributes or Tags, as they are treated as ... attributes, or tags.

Chatbots, Rule Engines and Inverted Queries

Here's something SQL cannot do: it cannot search for SQL. Absurd, right? One of the early chatbot technologies is called SRAI (Stimulus-Response AI), and the scripting language for SRAI is known as AIML (AI Markup Language). A snippet of AIML is given below.

```
<aiml>
   <category>
      <pattern>I love *</pattern>
      <template>I like <star/> a lot.</template>
   </category>
   <category>
      <pattern>I * you</pattern>
      <template>Well, I <star/> you, too!</template>
   </category>
</aiml>
```

The goal of this is to specify some conversational patterns to be matched, and, if matched, to produce the indicated output. Thus, if a human says *"I love baseball"*, the chatbot responds *"I like baseball a lot."* If the human says *"I hate you"*, the response *"Well, I hate you, too!"* is elicited. In the case of *"I love you"*, there are two possible responses, and a coin is flipped to pick one.

What is being done here is a kind of inverted query. One is presented with the 'answer', *"I love you"*, and the goal is to find all 'queries' which can be satisfied by that answer. In this example, the two queries are '<pattern>I love *</pattern>' and '<pattern>I * you</pattern>'. Abandoning the XML markup in favor of s-expressions, the two patterns are

```
(I love (Variable "?star?"))
(I (Variable "?star?") you)
```

An AIML chatbot consists of tens of thousands of such patterns, and the matching responses. In order to have rapid query responses, chatbot implementations organize the patterns into a trie[54] and typically implement some variation of the Rete algorithm.[55]

[54] A prefix tree, see https://en.wikipedia.org/wiki/Trie

[55] See https://en.wikipedia.org/wiki/Rete_algorithm

All this can be expressed in the Atomese meta-query language. It is necessarily a bit more verbose, but perhaps still readable without assistance. The AtomSpace is populated with SRAI rules, having the form

```
(Query
   (Present (Word "I") (Word "love") (Variable "?star?")))
   ((Word "I") (Word "like") (Variable "?star?")
      (Word "a") (Word "lot")))

(Query
   (Present (Word "I") (Variable "?star?") (Word "you"))
   ((Word "Well,") (Word "I") (Variable "?star?")
      (Word "you") (Word "too!")))
```

The reading of this content is as follows:

- 'Query' is a special keyword defining the rewrite rule. It is much like 'Meet', introduced previously, except that, this time, instead of delivering a grounding, it delivers a rewrite making use of that grounding. The first Atom following the query is the pattern to be matched. The second Atom of the query is the rewrite to apply, after a match has been found.
- The 'Variable' and 'Present' keywords are just as before.
- The 'Word' is a handy variant of a Node, reminding readers that the indicated string is a natural language word. One could have just as easily said 'Node', here; this is just some visual bling.

This is not SQL! We've said this before, but it is worth saying again: the metagraph database is populated with tens of thousands of such statements, "statements" which just happen to have the form of queries. In the database, these queries are dormant; they are not running, they are just sitting there, waiting for their day in the sun. The actual query to be performed looks like this:

```
(Dual (Word "I") (Word "love") (Word"you"))
```

Notice that this query has no variables in it: it consists entirely of constants! It is an 'answer', not a question! The subgraph isomorphism engine, aka the query engine, aka the 'pattern matcher', finds all matching patterns. In this case, it is the two queries shown earlier. To run the chatbot, one then flips a coin, picks one of the two queries, runs it, and prints the response. Running the query is computationally 'trivial', as there is only one pattern to match: *"I love you"*.

Pay attention: the above is *NOT* saying that there are ten thousand queries that are each performed, until a match is found! *No!* It is saying that only two queries are run: first, the 'Dual', to find some candidates, and then one of the candidates is chosen and run on a tiny, itsy-bitsy dataset consisting of one sentence. Its fast. Insofar as all of this content is stored as metatrees, all of the prior considerations about RAM usage, search performance and tree walking apply.

The above is more or less a peculiar variation of the Rete algorithm. The Dual link is used to narrow down the selection of rules to apply, then (users choice) either one of the rules is selected and fired, or maybe all of the selected rules can be fired.

The take-away lessons are:

- Inverted queries are still queries.
- SQL-inspired languages cannot support inverted queries. As a corollary, most graph databases cannot support inverted queries.
- Chatbots are highly specialized, customized database engines, highly tuned for performing inverse queries.
- Rule engines are highly specialized, customized database engines, highly tuned for performing inverse queries.
- A metagraph-style query language can easily support both queries and inverted queries, because the query engine itself is a subgraph isomorphism solver, and it does not particularly care about the types of the Nodes and Links that it is matching, as long as it is able to match.
- Because metatrees contain both the forward pointers and the weak pointers pointing backwards, they effectively have a trie (prefix tree) built in. That is, the organizational advantages of using prefix trees to perform rapid string-matching searches come "for free" in a metagraph, as a part of the general infrastructure.
- Because metaqueries can support both direct and inverted queries, they provide a natural infrastructure on which to build both rule engines and forward-inferencing chainers.

That chatbots can be unified with conventional query is perhaps one of the more entertaining aspects coming out of metaqueries.

Types

It was observed above that metatrees have a natural string representation as s-expressions. Every system that works with s-expressions gives a privileged interpretation to the first word after an open parenthesis. In Lisp and Scheme, that first word is an operator.[56] In a key-value store, the first element is the key, and the trailing elements are the value.

What is the first element for metatrees?

The answer appears to be that it is a type, a type-theoretical type. This is not a proclamation or a theoretical result, this is an experimental result. After working with meta-trees for a decade, and examining how they represent data, it appears that there isn't any other obvious interpretation than to say that it is a type.

The observational evidence is as follows:

- From the beginning, it is noted that there is a distinction between Nodes and Links, roughly analogous to the vertexes and edges of graphs. So it seems these are two different types.

[56] See John N. Shutt's blog post `https://fexpr.blogspot.com/2011/04/fexpr.html` for careful terminology.

- Could this could be reduced to just one type, by replacing Nodes with nullary Links? Not easily. Nodes need to be distinguishable, and thus are given unique string names. There is no particular need to give a Link a string name: it's outgoing list uniquely disambiguates that Link from any other.[57] In order to get rid of the Node/Link distinction, Links would have to be given names. A compelling reason to do this would need to be found.

- The description of the meta-query language introduces a Variable, which is obviously a kind of Node, but is distinguished from a conventional Node, in that it has special semantics in the context of a query.

- Similarly, the metaquery language introduces a Present Link, which is a kind of a Link, but, again, has special semantics in the context of a query.

- In the current OpenCog Atomese system, there are well over a hundred distinct Node and Link types, each introduced as a need arose. Booleans: AndLink, OrLink, NotLink.[58] Arithmetic: PlusLink, MinusLink, GreaterThanLink.[59] Set theory: SetLink, MemberLink.[60] Functionals: LambdaLink.[61]

- For many (but not all) of these Nodes and Links, it became convenient to create a corresponding C++ class that performs some function. A silly example is the PlusLink: when this is executed, the arguments are added together, if the arguments are numeric. The C++ class performs this addition. Now, it is generally regarded that C++ classes are types. By extension, PlusLink must also be a type.

- After recognizing that Nodes and Links and their variations are all types, it became easy to create a type system: SignatureLink, TypedVariableLink, TypeNode, ArrowLink.[62] It appears to be entirely consistent and, as a type system, entirely conventional.[63]

Ergo, it is natural to take the concept of a metatree, which already distinguishes between Node and Link, and recognize that this naturally blossoms into a type system.

As a type system, it is dynamically typed, not static-typed. New types can be introduced at any time. It doesn't even make sense to talk of a "static" type system, because the metatrees are not being "compiled" into the database: they are merely being inserted. There is a long-running REPL loop, and nothing is gained by preventing

[57] 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.

[58] See `https://wiki.opencog.org/w/AndLink`

[59] See `https://wiki.opencog.org/w/PlusLink`

[60] See `https://wiki.opencog.org/w/SetLink`

[61] See `https://wiki.opencog.org/w/LambdaLink_and_ScopeLink`

[62] See `https://wiki.opencog.org/w/SignatureLink`

[63] The OpenCog AtomSpace includes all the easy type-theoretical stuff. What's currently missing are type equations, type variables and dependent types. These are missing mostly because no one has asked for them yet. Atomese has evolved on an as-needed basis: if something isn't urgently needed, it does not get implemented.

users from defining new types at runtime. Note that this does make the type system clash with that of OCaml, Haskell, Scala, all of which deploy a static type system.

As a practical example: Nodes and Links are of type Atom. The biology subsystem uses GeneNodes and ProteinNodes, since these are useful for biological data representations. These are also type Atom. The biology subsystem is a distinct subsystem, and not part of the core: it is dynamically loaded. This creates issues for the OCaml binding, which might have known about Atoms at compile time, but had no clue about GeneNodes and ProteinNodes at compile time. Static types and metatree types do not play well together.

The second bullet point above idly wondered about a monotyped system, where everything is a Link, and Links have a string name. This would bring the system closer to a Lisp/Scheme conception of the world. Yet, this changes nothing: there is still a need to have variables, as well as and, or, not, set, member, plus, minus, *etc.* atoms. These would then have to be demarcated in the string name... and thus have to be recognized *de facto* types.

Examples of Types

By "types" it is meant both the types of computer science,[64] and of type theory.[65] In conventional computer 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, this is the arrow type, pointing from the input arguments to the output type. The ML, CaML, Haskell, Scala and F# programming languages provide an explicit type system, allowing the programmer to define new types (In OO languages, new types can only be defined by defining a new OO class).

Before proceeding further, a reference example will help clarify the idea. Consider the English language sentence *"The cat sat on the mat."* This encodes some information; how might it be encoded as a metagraph? There are many possible choices, but one of the simplest is shown here:

```
(SittingLink
   (ObjectNode "cat")
   (ObjectNode "mat"))
```

This clearly has the Link-Node structure of a metatree; but appears to be using some derived types. The denotational semantics is obviously that of the English language sentence; the denotation of the remaining pieces is left to the imagination of the reader. This is not laziness on the part of the writer: the word "denotation", when used in a computer science context, means "what the computer programmer is imagining it to be".[66] So, for example, a "float" is only a "number" in the programmer's head; what is actually in the machine code is just some bytes, and not "numbers". Likewise here,

[64] See `https://en.wikipedia.org/wiki/Type_system`

[65] See `https://en.wikipedia.org/wiki/Type_theory`

[66] Playing a bit fast and loose here: "denotational semantics", see `https://en.wikipedia.org/wiki/Denotational_semantics` is about converting what the programmer is thinking back into mathematical formulas. Computer science, just like math, loves to go meta whenever it can.

SittingLink is just some metatree sitting in RAM. It has no inherent meaning by itself; its meaning emerges only in relation to other metatrees in the system.[67]

The '(ObjectNode "cat")' could be replaced by a CatNode, although the usage would appear to change: one would then write '(CatNode "Fuzzykins")'. Alternately, one could choose to move in the opposite direction, and write:

```
(Link
   (PredicateNode "Object")
   (Node "cat"))
```

which denotes that cats are objects. The 'PredicateNode' denotes something that behaves like a truth value; the assertion is that it is true that cats are objects. But there is also a different denotational idea lurking here: The above is behaving as a kind of type declaration! It appears to be declaring that things of type "cat" are things of type "Object". One could sharpen this, and write

```
(TypeDefinitionLink
   (TypeNode "CatNode")
   (TypeNode "ObjectNode"))
```

After this type definition is provided, it appears to be appropriate to write things like '(CatNode "Fuzzykins")'.

To round out this example, it might also be useful to define a 'CatLink', which could then be used to link together disparate properties:

```
(CatLink
   (PredicateNode "has")
   (Node "fur"))
(CatLink
   (PredicateNode "URL")
   (Node "https://example.com/cutie-fuzball.gif"))
```

This is just generic knowledge representation. Caution: although the metatrees (and metatypes) above are encoding information about cats, they may not be the "best" way to represent knowledge. A more conventional approach would be to define 'IsALink' and 'HasALink' and 'PartOfLink': holonyms and meronyms, and use those in one's lexical ontology. There is freedom in how this can be done; there is nothing in the metatrees that is forcing certain types upon the user, over other types.

Properties of a Metagraph Type System

In a metagraph, the concept of a primitive type appears to be fluid; new types are naturally definable by means of type inheritance. By "primitive", it is meant a type that is not built with one of the usual type constructors: a list constructor, a function-type constructor, a product (pair) constructor.

There does not seem to be any need to prevent the creation of new types at run-time. This is in contrast to statically-typed programming languages. Languages such as OCaml demand static typing so that type inference can be done at compile time.

[67] Philosophically, this is a pan-psychic view of the world: individual ur-items are meaningless; all meaning comes from the relationships between objects.

After a module has been compiled, the resulting binary has an API with fixed types in the interface. This provides safety: the user cannot call into the binary with bogus arguments, leading to undefined behavior or crashes. In that sense, static types "make sense" for compiled programming languages. The case is different here: with metagraphs, the primary goal is not programming but data representation. Metagraphs don't "do anything", they are not an "executable program", they just "exist". At least this particular reason for static type-safety seems to disappear.

Signatures and Arrows

It appears that metatree types are also implicit type constructors, in that they are trivially reified.[68] Thus,

```
(ListLink
   (ConceptNode "some")
   (ConceptNode "thing"))
```

obviously has the type signature[69] of

```
(SignatureLink
   (ListLink
      (TypeNode 'ConceptNode)
      (TypeNode 'ConceptNode))
```

It has two other signatures as well:

```
(SignatureLink
   (ListLink
      (ConceptNode "some")
      (TypeNode 'ConceptNode))
```

and

```
(SignatureLink
   (ListLink
      (TypeNode 'ConceptNode)
      (ConceptNode "thing"))
```

This kind of easy reification makes it possible to easily attach complex types to type definitions. Returning to the previous example of a 'CatLink', one could write

```
(TypeDefinitionLink
   (TypeNode "CatLink")
   (SignatureLink
      (TypeNode 'PredicateNode)
      (TypeNode 'Node)))
```

which indicates that the only valid way to use a 'CatLink' is as a binary Link, where the first element of the outgoing pair must be a 'PredicateNode' and the second must

[68] See https://en.wikipedia.org/wiki/Reification_(computer_science)

[69] See https://en.wikipedia.org/wiki/Signature_(logic) for the formal definition of a signature, and https://en.wikipedia.org/wiki/Type_signature for a quotidian, non-abstract working-programmer definition.

be a 'Node' (and not a 'Link'). There does not appear to be anything challenging to introducing sigma types: one could have written

```
(TypeDefinitionLink
   (TypeNode "CatLink")
   (SignatureLink
      (TypeNode 'PredicateNode)
      (TypeChoice
         (TypeNode 'Node)
         (TypeNode 'Link))))
```

which states that the second element of the outgoing pair can be either a Node or a Link. To reiterate: type reification appears to be trivial with metatrees.

To round out the above examples, we present the very common case of the signature of lambdas. Here, the lambda

```
(LambdaLink
   (VariableList
      (TypedVariable (Variable "$x") (TypeNode 'ConceptNode))
      (TypedVariable (Variable "$y") (TypeNode 'ConceptNode)))
   (FooBarBodyLink ...))
```

obviously must have the signature of

```
(ArrowLink
   (VariableList
      (TypeNode 'ConceptNode)
      (TypeNode 'ConceptNode))
   (TypeNode 'FooBarBodyLink))
```

That is, the arrow points from a list of the input types to a list of the output types.

Jigsaw Puzzle Pieces and Sheaves

Caution: the idea of Lambdas, and of Arrows, while extremely widespread in programming practice, is the source of tremendous confusion and difficulty in knowledge representation. Even worse: just as fish are unaware of water, most programmers are unaware of alternatives to lambdas.

There is one extremely important alternative, discussed in other texts in this series: the jigsaw puzzle piece, and its tab and slot connectors. Telegraphically: if one has a function $f(x)$ and beta-reduces it with 42 to get $f(42)$, one is 'connecting' the jigsaw-puzzle slot x in $f(x)$ to the jigsaw-puzzle tab 42. That is, beta reduction is an example of jigsaw-puzzle assembly. Now, full-fledged jigsaw puzzle pieces can connect in every-which way (provided that the tabs and slots are correctly mated). This is more general than term algebras, in which term trees are always DAGs. That is, a lambda is an anonymous term, an arrow is a component of a term tree. For computer programming (functional or imperative), one always thinks in terms of calling functions, methods with arguments, leading to the notion of an abstract syntax tree. For knowledge representation, it does not have to be that way. The directionality of the arrow can be discarded, and mating can be indicated with the connectors themselves. That is, the ArrowLink is a special case of a more general jigsaw-puzzle-piece link:

```
(ConnectorSeq
   (Connector
      (TypeNode 'ConceptNode)
      (DirectionNode "input"))
   (Connector
      (TypeNode 'ConceptNode)
      (DirectionNode "input"))
   (Connector
      (TypeNode 'FooBarBodyLink)
      (DirectionNode "output")))
```

This example fails to show that there can be a vast multitude of directions, beyond just "input" and "output". The rules for allowed connectivity patterns of connectors lie outside of the definition of individual jigsaw pieces.

To conclude: metatrees are naturally typed; those types are naturally reified; the reifications are recursive, and the level of recursion is limited by the imagination.

Evaluation / Execution

Metatrees generalize the notion of trees. A special case of trees are the "abstract syntax tree".[70] The denotational semantics of abstract syntax trees is that they are executable programs. Thus an expression such as "*if (a > b) return a+b;*" can be represented by an abstract syntax tree, given below:

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

As such, some metatrees can serve a dual purpose: both to record syntactic structure, and to be executable. Perhaps this feels painfully obvious at this point; if this is the case, it should be contrasted to the situation of programming in lisp or scheme. First of all, in lisp, one is programming, not representing data. The source code for lisp programs resides in files, not in databases.[71] In the end, a lisp program is always executable (evaluatable); if it is not, that is because it is broken in some way. This is not the case here: there is nothing about metatrees that makes them inherently evaluatable or executable. Indeed, the above fragment is not executable until the variables are bound to some values. Worse, its ambiguous as to what should happen if a is not greater than b. As a program, it is not really valid. As a metatree, it is as good as any other, differing only in having the hint of being possibly, potentially evaluatable.

[70] See `https://en.wikipedia.org/wiki/Abstract_syntax_tree`

[71] Let's not be silly. Of course one could stick the source code into a database. But how would you write a query to get it back out, and evaluate it? In lisp? Well, yes, of course, one could write some wrappers to do that. So what? Are you representing knowledge yet? Did you create a knowledge query system?

A subset of metatrees can be considered to be a programming language. This may appear to be self-evident; that this has not been the case historically is demonstrated in the comments below.

Fexprs, Macros and Rewriting

In lisp, functions that act on abstract syntax trees are called "fexprs".[72] Modern lisps no longer explicitly support fexprs; an influential paper from 1980 labeled them as harmful, and made the argument that macros were superior.

That's all well and good, but macros, as defined in lisp/scheme, can only be "run once". Macros run on source expressions, and they expand the source expression into something evaluatable. That is, macro languages allow the programmer to define a set of rewrite rules to transform input syntax trees into output syntax trees. However, conceived of as macros, they can only be run once, during a pre-processing stage; they cannot be run later, once the resulting program has started executing. In this sense, they are limited in a way that fexprs are not.

This is in sharp contrast to the needs of predicate logic[73] and proof theory.[74] Theorem provers are constructed out of term rewriting systems.[75]

Note that SQL does offer a poor-mans term rewriting system. It is not general, and not suited to the general needs of term rewriting, but it is plenty enough to enable "data processing".[76] For example

```
CREATE TABLE foo (name TEXT, number INT);
CREATE TABLE bar (person TEXT, phone INT, nickname TEXT);
INSERT INTO bar(person, phone) SELECT name, number FROM foo;
```

performs a rewrite of records (rows) from one table to another. Some sort of ability to perform rewrites is innate to any database system, as a primary need is not only to store data, but to perform transformations on it. These transformations must be done at run-time, dynamically; they cannot be done at "compile time", the way that the lisp/scheme macro system performs them.

It should be clear that metagraphs in and of themselves are fairly useless without some sort of term rewriting system to go along with them; this was already implied in the earlier discussions concerning database queries.[77]

What is new here is that, if the metagraphs are represented as s-expressions, some of those s-expressions are potentially evaluatable, and thus lisp-like. Unlike conventional

[72] See `https://en.wikipedia.org/wiki/Fexpr`

[73] See `https://en.wikipedia.org/wiki/First-order_logic`

[74] See `https://en.wikipedia.org/wiki/Proof_theory`

[75] See `https://en.wikipedia.org/wiki/Rewriting` and `https://en.wikipedia.org/wiki/Abstract_rewriting_system`

[76] And I do mean "data processing" in all of its glorious 1960's generality: performing database queries to associate an electric utility customer's account with their electric usage for that month, and thier mailing address, and print a bill to be mailed to the customer. This is, in a sense, quintessential data processing, and central to it is the database holding names, accounts and monthly usages.

[77] Maude[4] is an example of a term rewriting system disassociated from theorem provers, databases and inference engines. See `http://maude.cs.uiuc.edu/maude1/manual/maude-manual-html/maude-manual_0.html` and `http://maude.cs.illinois.edu/w/index.php/The_Maude_System`

lisp, the evaluation system must live in harmony with the term rewriting system; thus, enabling the evaluation of metatrees seems to require some sort of clearly defined fexpr or $vau interfaces in the evaluator.

ProLog

An alternative to the lisp/scheme fexpr/$vau mindset is provided by ProLog. It is both a programming language, in that ProLog programs are executable, and it is a knowledge store, in that the Datalog fragment of ProLog is explicitly intended to store data.

ProLog makes abstract syntax trees 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:

```
parent_child(tom, erica).
parent_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 per person, 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*". [78] 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.

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.[79] For

[78] 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?

[79] See `https://en.wikipedia.org/wiki/Common_Intermediate_Language`

Gnu GCC, it is GIMPLE.[80] For LLVM, it is the LLVM IR.[81] These are somewhat like assembly code, but abstract, and not specific to CPU instruction sets. They encode abstract syntax trees, and thus sit above the bytecode/instruction-set 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 suit 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 during debugging, when they are presented to the programmer 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 typed 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.

Human-oriented vs. Machine-oriented

ProLog is a human-oriented programming language. It is expected that humans are going to write prolog, and thus the language features are designed to be easy for humans to manipulate. In this sense, it is like almost all programming languages – almost all, but not all.

The intermediate languages are not human-oriented, but machine oriented. It is not expected that programmers will be writing in intermediate languages, no more than they are expected to write in assembly code. Intermediate languages are designed to be easy to use by machines: specifically, by the internal algorithms of compilers. A compiler needs to turn high-level languages into assembly code. The resulting assembly code should run fast; but how to perform the optimization, the rearrangement of terms that enables fast assembly? Some compiler optimizations can be done directly on the input source: for example, loop hoisting. Some compiler optimizations can be done directly on the assembly: for example, peephole optimizations. The vast majority of desirable program transformations cannot be done in either the source language or the target language. This is why intermediate languages exist. Intermediate languages are fine-tuned to make it easy for algorithms to read and write them. They are designed for machines, and not for people.

It appears that any kind of programming language constructed from raw metatree s-expressions is more machine friendly than it is human-friendly. The example given earlier:

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

[80] See https://gcc.gnu.org/wiki/GIMPLE

[81] See https://hub.packtpub.com/introducing-llvm-intermediate-representation/

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

is stunningly verbose, as compared to the far more succinct "*if (a > b) return a+b;*". Although humans could write metagraph programs using the s-expression representation, it would be rather tedious to type up, and rather painful to read.[82]

To conclude: although some metatrees might be executable or evaluatable, and thus constitute a programming language, it does not appear to be a language that is particularly human-friendly, at least, not in it's s-expression representation.

Lets ruminate one more paragraph on this. One *could* create a human-readable language on top of of metatree s-expressions. There have been at least two larger efforts: "ghost"[83] and "MeTTa",[84] as well as some lesser efforts (a chatbot triplestore). Doing so defeats the machine-readability of the s-expressions themselves. Much of the primary utility of using metatrees comes from being able to apply graph rewriting rules to them. A graph rewrite rule consists of a pattern template to be matched, and the resultant transformation. When the template is an s-expression, of the kind given in the section on MetaQuery Language, then it is easy to match. When it is something else, the application of the pattern rules becomes difficult. The pattern language can no longer be simple, and worse, the pattern language diverges from the source language itself. The s-expression form seems to hit a happy medium: the pattern language is a valid subset of the whole language. It is easy to parse, making it ideal for algorithmic manipulation. It is too low-level to make it comfortable for humans; but given that we live in a machine age, this is perhaps a not unexpected twist.

Atomese

Almost all of the various ideas articulated in this text have been implemented in the OpenCog AtomSpace, nominally going under the name of Atomese. Historically, it was created to be a knowledge representation system in which probabilistic reasoning could be performed. Regular use and the pressures of user demands have forced the sharpening of various concepts and features, most of which have been reviewed above.

The platform continues to be deployed in production systems, while also being an experimental platform for the development of new ideas. As these are found to be sound, the system does, as a whole, mutate.

If the reader of this text finds any of the above concepts to be dubious, or perhaps ill-defined and in need of sharpening, they are encouraged to contact the mailing list. The AtomSpace, and Atomese are malleable. Bad ideas do need to be replaced by good ones.

[82] Heh. In fact, I (we: other users too) write metatree s-expressions daily. Perhaps this is a case of "do as I say, not as I do".

[83] See `https://wiki.opencog.org/w/Ghost`

[84] See `https://wiki.opencog.org/wikihome/images/b/b7/MeTTa_Specification.pdf` which can be found in the `https://wiki.opencog.org/w/Hyperon:Atomese` wiki page.

Conclusion

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

- Metagraphs are a simple and relatively minor generalization of graphs.
- Metagraphs are more representationally compact than graphs. They are more efficient at representing data.
- Metagraph query is more general than SQL-inspired query systems, and breaks out of the table-join paradigm that is most commonly offered in graph-query systems.
- 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.
- Metatrees are naturally typed. Ordinary graphs are not.
- Metatrees are naturally (even trivially) reified. Ordinary graphs are not.
- Metatrees are naturally abstract syntax trees; this is possible but not natural when working with ordinary graphs.
- Metagraphs are foundational, providing a common platform for a broad variety of computing tasks: ranging from data storage, to symbolic computing and inference, to functional (or imperative) programming.
- Metagraphs are low-level. They appear to be ideal for algorithmic manipulation. They are sufficiently human-readable to allow daily authoring, but they are not a substitute for a proper high-level programming language targeting humans as the programmers.

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.

Thanks

The second revision of this text has benefited from the (witting or unwitting) helpful input of Amirouche Boubekki, Kino Coursey, John Cowan, Marc Nieper-Wißkirchen, Adam Vandervorst and Steven Wiley. Absolutely none of this would have been possible without a decade's worth of gracious support and tutelage of Ben Goertzel.

References

[1] Erik Meijer and Gavin Bierman, "A co-Relational Model of Data for Large Shared Data Banks", *ACM Databases*, 9, 2011, URL `https://queue.acm.org/detail.cfm?id=1961297`.

[2] Binary Analysis Platform, *Module Ogre: Open Generic Representation*, Tech. rep., Carnegie Mellon University Binary Analysis Platform, URL `http://binaryanalysisplatform.github.io/bap/api/odoc/ogre/Ogre/index.html`.

[3] David I. Spivak and Ryan Wisnesky, *A Functorial Query Language*, Tech. rep., 2014, URL `https://popl.mpi-sws.org/2014/dcp2014/wisnesky.pdf`.

[4] Manuel Clavel, et al., *Maude: Specification and Programming in Rewriting Logic*, Tech. rep., SRI International, 1999, URL `http://maude.cs.uiuc.edu/maude1/manual/maude-manual-html/maude-manual_0.html`.