Track Nodes

You may see calls to `trackInlineObjectsInRootNode()` and `findRootNodeAncestor()` in some parts of the code. These are both defined in `schema/node-model.js`. Node tracking is the tracking of relationships between a node's object values (not children), and the node's ID. E.g. Take, the following node:

```
let nodeA = {
  id: `id2`,
  internal: {
    type: `footype`,
  },
  foo: {
    myfile: "blog/my-blog.md",
    b: 2,
  },
  bar: 7,
  parent: `id1`,
  baz: [{ x: 8 }, 9],
}
```

Its sub objects are `foo` (value = `{ myfile: "blog/my-blog.md", b: 2}`), and those in the `baz` array (`{ x: 8 }`). Node tracking will track those back to the top level node's ID (`id2` in this case). The [`trackInlineObjectsinRootNode()`]() function takes care of this and records those relationships in the [rootNodeMap]() WeakMap. E.g. after calling `trackInlineObjectsInRootNode(nodeA)`, `rootNodeMap` would contain the following records:

```
// rootNodeMap:
{
  { blog: "blog/my-blog.md", b: 2 } => "id2", // from `foo` field
  { x: 8 } => "id2",        // from `baz` array
  {                         // top level object is tracked too
    id: `id2`,
    internal: {             // internal is not mapped
      type: `footype`
    },
    foo: {
      blog: "blog/my-blog.md",
      b: 2
    },
    bar: 7,
    parent: `id1`,
    baz: [ { x: 8 }, 9 ]
  } => "id2"
}
```

Find Root Nodes

To access this information, `schema/node-model.js` provides the [findRootNodeAncestor()]() function. It takes an object, and looks up its parent's nodeID in `rootNodeMap`. It then finds the actual node in Redux. It then gets that node's `parent` ID, and gets the parent node from Redux. And continues in this way until the root node is found.

In the above example, `nodeA` has parent `id1`. So `findRootNodeAncestor({ blog: "blog/my-blog.md", b: 2 })` would return the node for `id1` (the parent).

Why/Where?

Where is node-tracking used? First up, nodes are tracked in 2 places. Firstly, in createNode, every time a node is created, we link all its sub objects to the new NodeID. Nodes are also tracked whenever they are resolved in run-sift. This is necessary because custom plugin fields might return new objects that weren't created when the node was initially made.

Now, where do we use this information? In 2 places.

1. In the `File` type resolver. It is used to look up the node's root, which should be of type `File`. We can then use that root node's base directory attribute to create the full path of the resolved field's value, and therefore find the actual `File` node that the string value is describing. See File GqlType inference for more info.
2. To recursively look up node descriptions in type-conflict-reporter.ts