Materialization is Evil

Bob Adolf, PNNL

Motivation

Real-world analysis is rarely a one-function game. While it might be transiently useful to count the connected components of a graph or the number and length of shortest paths from one vertex to another, these metrics seldom represent a final answer in and of themselves. In order to both leverage the productivity of reusable building blocks and simultaneously allow the construction of complex applications, composability should be a first-class design requirement of any code library.

Challenges for Libraries in Data-Intensive Graph Analysis

Composing graph algorithms is a non-trivial problem. At a high level, the issue is that the execution of a graph algorithm depends largely on the structure of the underlying data. There is no efficient, static ordering of the elements around which to base a program, and the variance in the size and shape of result sets can be extreme.

Unfortunately, this challenge has largely gone unmet in the community. Most existing graph libraries ignore the issue, opting instead to bypass the problem by materializing intermediate answers between operations. This approach has two serious consequences:

1: Global Work Inefficiency

The execution time of composed functions is *at least* the sum of the functions applied individually. At first glance, this seems benign–satisfying the triangle inequality does not seem like a weakness. However, this quality is not necessary (or desired). Graph algorithms often share similar subcomputations, so the overall work *required* by the composition of two functions is likely much smaller than the work implied by the execution of those two functions in tandem. We call this *global work inefficiency*, in reference to the redundant work incurred by specifying two local subproblems (library calls) instead of a single global one (custom implementation). Moreover, this inefficiency is exacerbated by the runtime variance common in parallel implementations. Specifically, since parallel graph algorithms are often dynamically-scheduled or, at very least, unpredictable in their execution, the termination time of parallel workers is often ragged, leading to poor parallel efficiency near the beginning and end of parallel functions. This incurs additional overhead between the completion of the first function while the answer set is materialized and before the inception of the second.

2: Working Footprint

For most composable graph analytics, the structure of the input and output are identical–usually a graph or attributed graph. As a consequence, very large inputs can produce very large outputs, and when overall memory or external storage constraints are tight, materialization can be a hinderance. Furthermore, on hardware platforms which depend heavily on locality and communication-avoidance, requiring additional resources to hold a materialized intermediate result can imply a lower overall computational density, hurting performance. (This secondary effect can be thought of not as running out of space on a node in a cluster but as being forced to use two nodes for the same problem and incurring an inter-node communication penalty.)

Some Possible Alternatives

For the sake of discussion, we present some alternative approaches to materialization. We have not done the appropriate research to validate any of these approaches in the implementation of a scalable graph library, nor do we claim to know which (if any) has more promise. Instead, we intend for these to be strawman arguments to encourage debate on this topic.

Distributive Operators

There exist some graph operators for which there may be a mathematical solution to the materialization problem. For instance, consider composing a global operator F_g over a per-node operator F_n. While the expression of the $F_g(F_n(G))$ implies that the *entire* result of $F_n(G)$ must be passed to F_g, the reality is the F_n can be applied to each node during the computation of F_g without changing the overall computational meaning. This observation is not new with respect to per-node operators; most visitor-oriented graph libraries already support this implicitly in their programming style. However, for general graph operators, this is much more complicated. Composing two or more global functions would require in-depth mathematical analysis and a clever algorithmic description to even think about automatically rewriting the operators to merge them. We see some hope in the work on sparse-matrix formulations of graph algorithms and in extensions to visitor semantics, but we acknowledge that a purely mathematical solution is possibly infeasible or even impossible.

Monadic Function Composition

An orthogonal (and, we believe, simpler) problem is the removal of unnecessary language artifacts preventing function composition analysis from even happening. In particular, we consider the substitution model of evaluation (evaluate subexpressions before evaluating the expression) harmful to efficient function composition. In most languages, the statement $f(g(x))$ is more or less equivalent to $t \leftarrow g(x); f(t)$. In other words, materialization is *implicit* in the normal language syntax for function composition. We propose the adoption (or at least investigation) of more declarative syntax. In particular, we note that the use of monads to encode composition provides a natural environment for deeper algorithmic analysis: $f(g(x))$ becomes $eval(f(g(x)))$ where eval is a special function used to construct an answer set. Note that using monads still requires deep insight into the system of representation of a function to enable efficient composition, but the notion of *where* that representation is explicitly stored is solved.

Continuation-Passing Style

Finally, as somewhat of a combination of the previous two ideas, we believe there might be value in investigating implementation of graph algorithms in continuation-passing style (CPS). At a high level, CPS can be thought of as a generalization of the visitor model embraced by several graph libraries. Instead of explicitly naming particular callback points in a single, fixed execution model, CPS allows a second function to pick up wherever the first left off. More accurately, composition is handled by requiring functions to accept a continuation argument and then passing the second function as a first-class entity to the first. Strictly speaking, CPS does not solve the composition problem, as a literal translation into CPS would simply materialize the graph before calling the continuation, but the ideas behind CPS might engender novel approaches to it. For example, if the result set of a function is intended to be a subgraph, perhaps a continuation could be generated for each element. Then, the first function could be evaluated lazily, allowing the second function to begin consuming answers, reducing the working footprint. If more than one continuation was allowed, it might be possible to reduce global work inefficiency as well, drawing upon lessons learned from use of the visitor model. Still, using CPS would require qualitatively different implementations of well-known graph algorithms, and it is unclear whether continuations provide sufficient flexibility to create a general composition strategy.