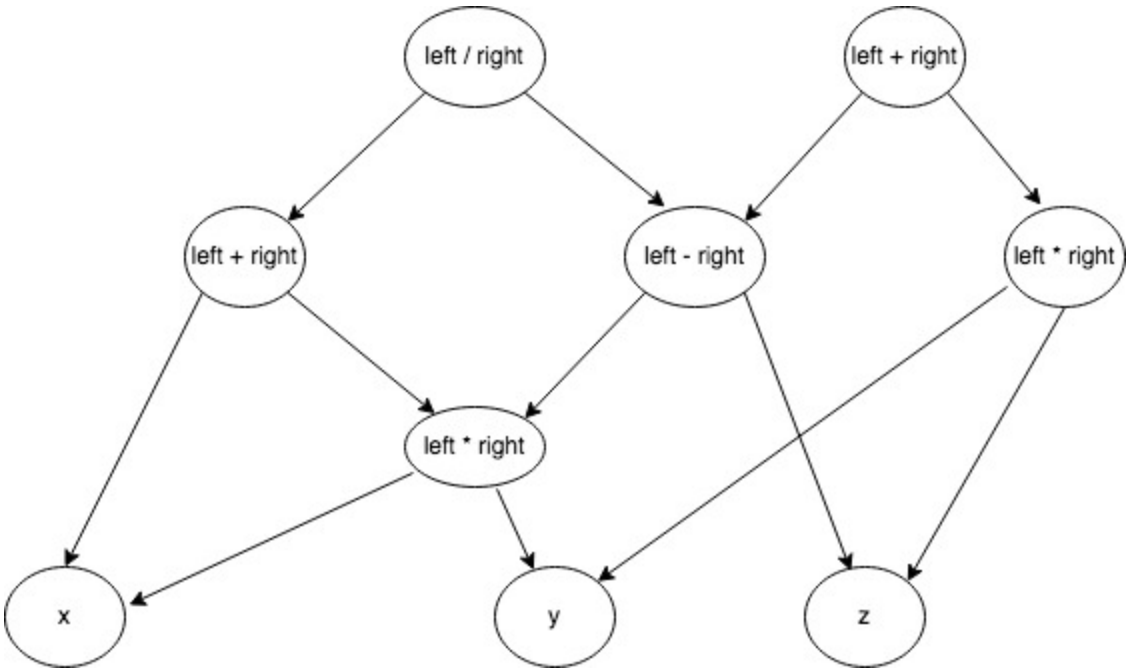
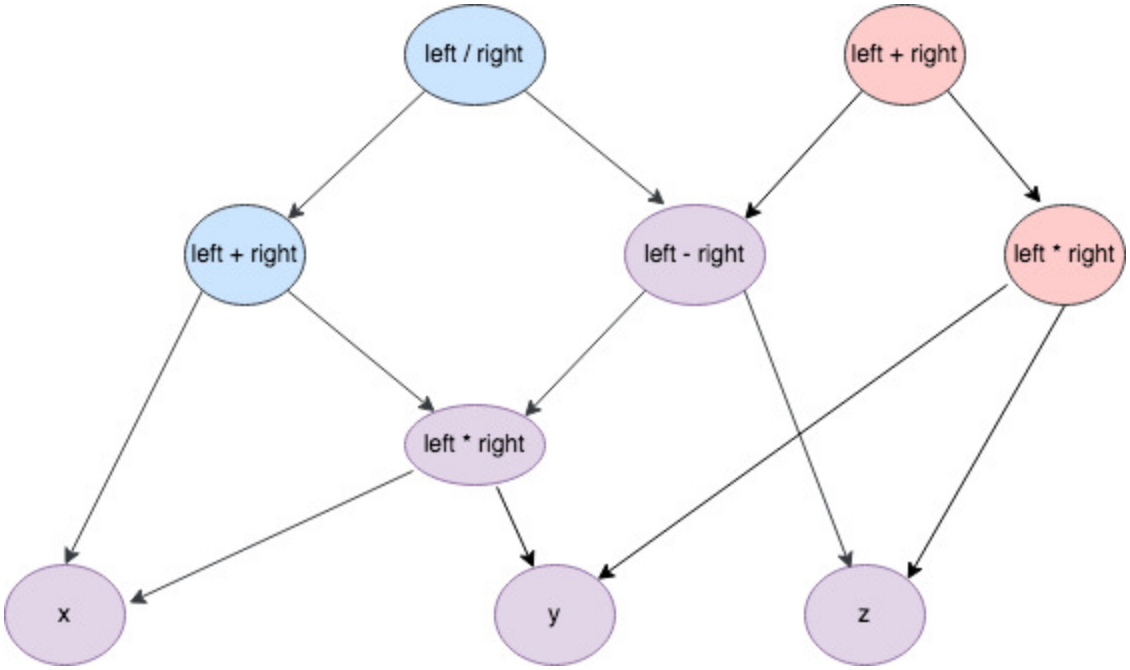
DICE: Going from O(graph) to O(changed) recomputations

Background

Consider some graph, where `x`, `y`, and `z` are input leaf nodes, and all other nodes are computation nodes. We introduce the following terminologies relating to graph computations.

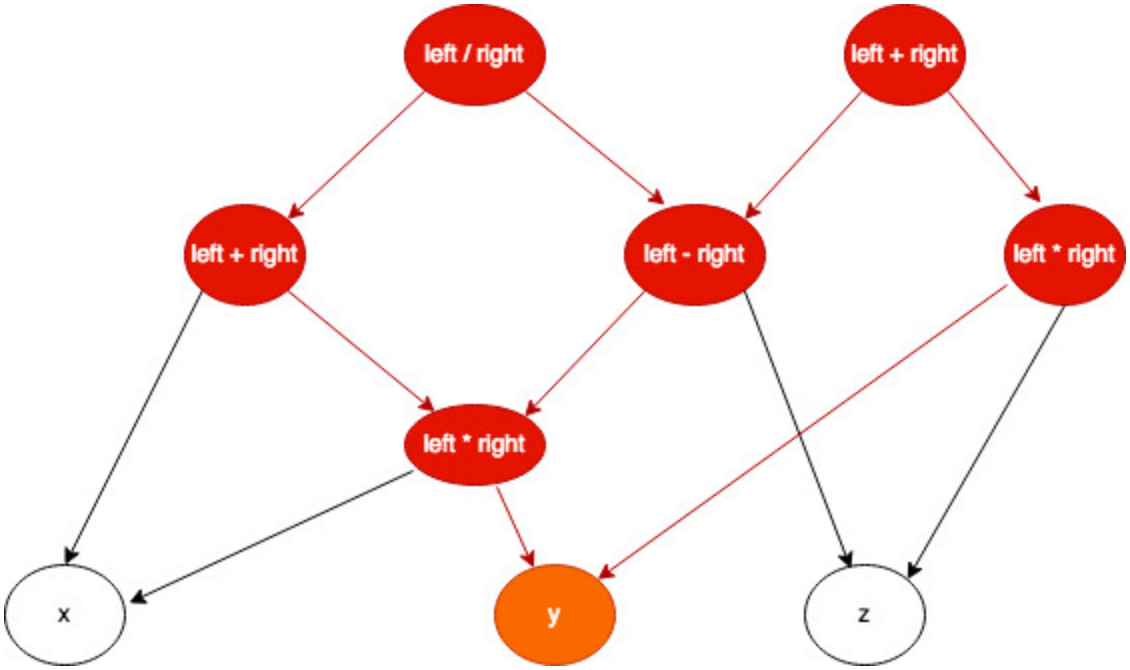


We define the **mountain** as the set of nodes required for a particular computation request. For example, in the above graph, we can have requests to nodes `left / right` or `left + right`. They will require values of a different set of nodes from each other, which doesn't necessarily require the whole graph.

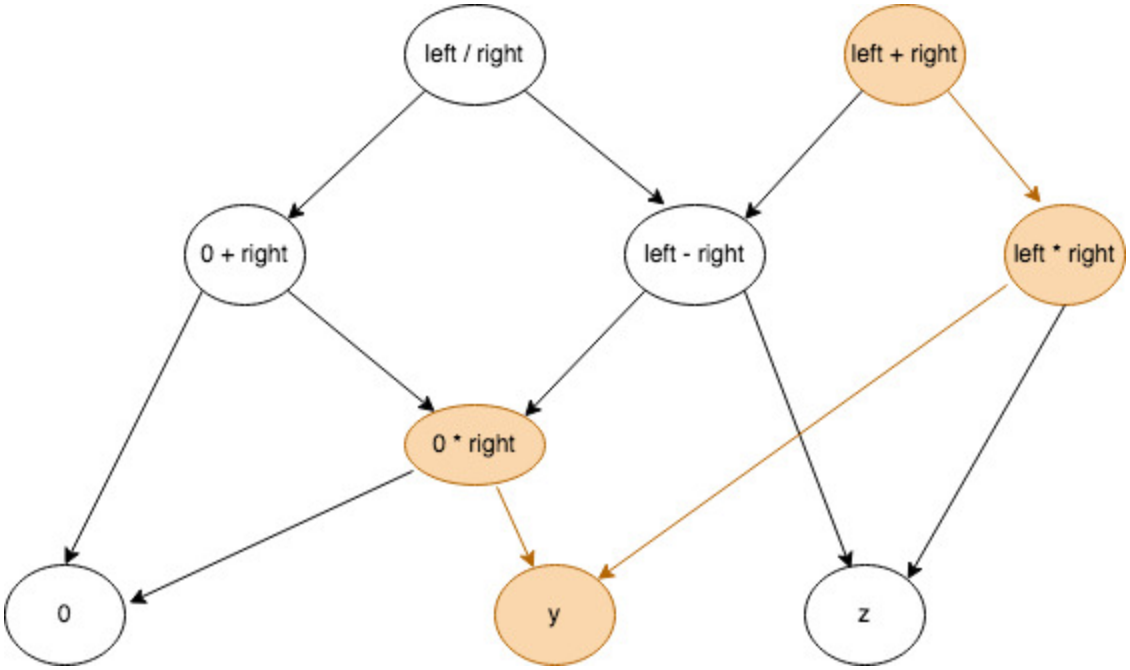


Blue = nodes needed by `left / right`, red = nodes needed by `left + right`, and purple are nodes needed by both.

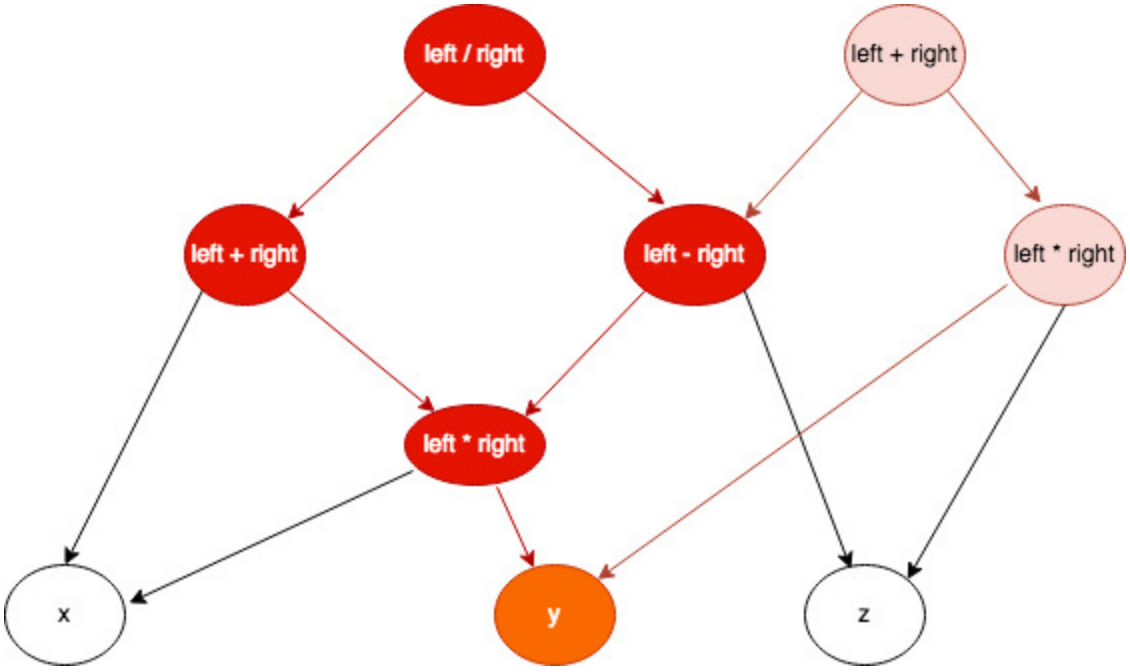
We define the **invalidate set** as all nodes that might be changed due to some changes to leaves. In the above example, the invalidated subset if `y` changed is marked red below.

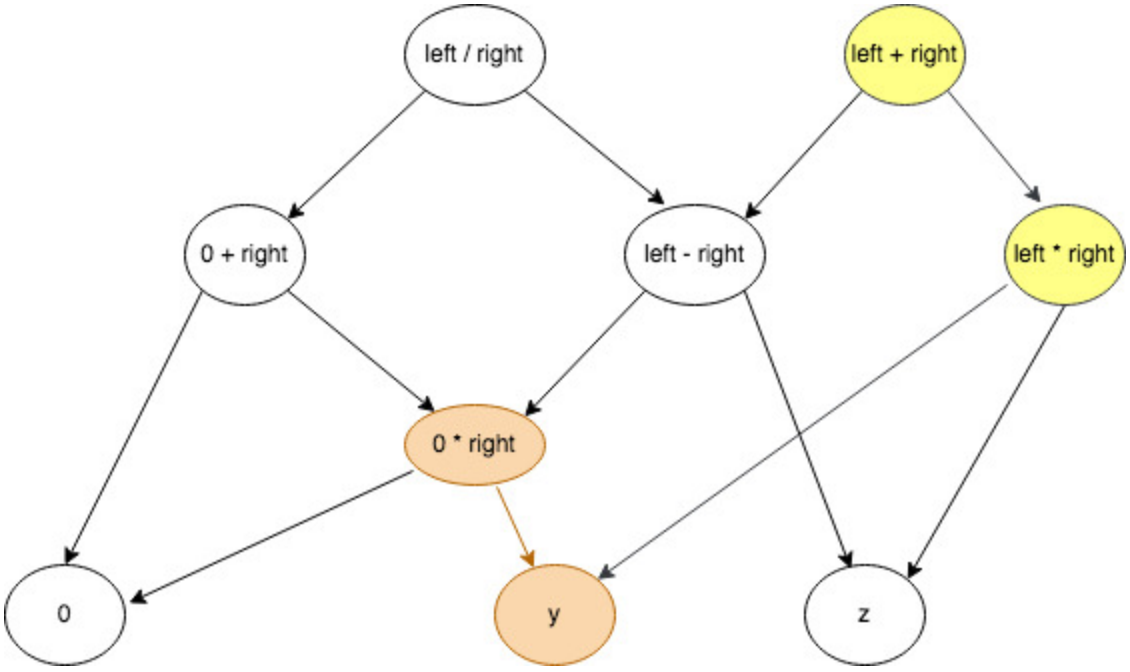


The **computed set** is the set of nodes that actually needs to be recomputed due to some changes to the leaves. It's always a subset of the invalidated subset. For example, if `x = 0`, then we can see that we would ever only have to re-evaluate a single node for changes to `y`.



We can also define **invalidated subset** and **changed subset**, which are the intersections of a mountain, and the invalidated set and computed set, respectively.
The figures below show the two disjoint invalidated subset and changed subset of the two top level nodes.





The below will outline how DICE plans to achieve `O(changed subset)` recomputations while performing `O(invalidated set)` graph traversals with concurrent requests and updates. There will also be support for an opt-in `O(changed)` computations and traversals invalidations.

I'll also float around some tentative ideas around how DICE may achieve "near" `O(changed subset)` recomputation and traversals in the future.

Prior Art

There have been prior incremental computation libraries.

Adapton is a general incremental computation framework that introduces concepts of `ref cell`s, values that are changeable, and `thunk`s, function observers of `refs`. Adapton has eager invalidation that traverses reverse dependency edges to mark dirty nodes. On evaluation, only dirty nodes are traversed for re-evaluation. Adapton achieves `O(changed subset)` recomputations and `O(invalidated)` traversals.

Salsa is based off of Adapton. It has notions of inputs which are a value and a version, and pure functions that consume those inputs yielding values, and dependency and version information. It's algorithm technically supports both lazy and eager dirtying by either propagating changed version information eagerly, or transitively checking for changed versions. It has performance characteristics of Adapton, or `O(changed subset)` recomputations but `O(mountain)` traversals if lazily dirtying.
Both Salsa and Adapton assumes a single client. Changes are available globally immediately, and that there are only requests for the latest version available, so concurrent requests and updates are not allowed.

Skip offered a MVCC based Adapton for concurrent requests. It stores multiple versions of each node via MVCC and uses transactions to commit changes globally, and has "graph coarsening" to reduce graph size. It offers the same performance characteristics as Adapton, using eager dirtying.

DICE

The below describes an algorithm that invalidates in `O(invalidated set)` traversals, and handles requests with `O(invalidated subset)` traversals and `O(changed subset)` recomputations.

Versioning

DICE allows multiple concurrent updates and requests to any node (not just leaves). Updates are batched in commits, so that updaters indicate multiple updates via `ctx.update(...)` that are only available to readers upon `ctx.commit()` (dropping the ctx without commit rolls back). Each commit is written to the immediate next version number, which is increased atomically with each commit so that we form a linear ordering of `commit`s.

Requests obtain the newest available version when they obtain the context via `dice.ctx()`. That specific version is used for the remainder of the request. It will never see values specific to any other version.

We guarantee that there is never a version being read that is larger than the most recent committed version.

The storage has a choice of whether to keep multiple nodes of different versions, or only keep the latest node.

Graph Nodes

Every node is a key `K` that is associated with a function `fn(K, ctx) -> R` to produce a node value `R`. For every evaluation of the function, we track all dependencies it accesses, forming both forward and reverse dependency edges.
This makes our graph an index store mapping keys `K` to a node containing one or more of a value `R` and metadata of the history (i.e. versions the result is valid at and dirtied at), and forward and reverse dependency edges. Note we store one or more of these information per node such that we have the ability to support storing multiple versions of a result.

I.e. we have `K => One or more (R, Deps, RDeps, History)`.

If for any two nodes with the same `K`, the `R` is equal, then they can be merged into a single node by merging their history by combining intervals of `verified_at` and concatting the `dirtied` versions. This lets any dependants of this node see if the current node has changed, and whether they need to be re-evaluated. Each node is locked individually throughout parallel computations.
This also ensures that for any key `K`, `verfied_at` contains all version intervals for which `K` evaluated to a specific `R`.
When merging, we will need to mark down for which version intervals the node's dependencies stored are valid, since it's possible that the dependency edges changed but the result value is the same. We can store multiple different versions of the dependency edges.

The History

Our history is a list of versions, and a state of Unknown, Dirty, or Verified at each version. We can say that `history.verified_at` contains all verified versions, and `history.dirty` contains all dirtied versions.

For any node, the history is unknown at every version until a version for which it becomes verified, for which it is verified for all future versions until dirtied, upon which future versions become unknown since the dirted version.

We can guarantee that there does not exist any interval `i` in `verified_at` such that there exists `v in dirtied` where `i.begin <= v` and `i.end > v`.

Invalidation algorithm

The invalidation algorithm is similar to Adapton, traversing up the reverse edges and marking the nodes as dirty.

```
changed(k: K, v: VersionNumber) {
  match resCache.getIgnoringVersion(k) {
    Some(res) => {
      res.history.dirty(v);
      res.rdeps.for_each(|edge| {
        if not max(edge.dest.history.dirty) > edge.version {
          changed(d, v)
        }
      )
    }
    None: resCache.putVacant(k).history.dirty(v)
  }
}
```

We will always propagate dirty markers on `ctx.commit()`, so we guarantee that unless `history.dirty(v)` was called, the result at `v` is the same as at `v-1`.
Note that we only traverse up the rdeps if the node has not been dirted after the version for which the rdep was recorded. This is because if the node was already dirtied after the rdep version, that change at the dirtied version may have changed the deps of the node such that the current rdep edge is not formed. We rely on the recomputation at the previous change to "delayed" propagate the dirty if necessary.

Compute Algorithm

The compute algorithm for a key `K` at a version `V` is to check the graph store index,

```
compute(k: (V, K)) -> Res {
  match resCache.get(k) {
    Match(res) => return res
    Mismatch(res) =>
      if (isDepsChanged(k, res)) {
        return recompute(k, res)
      } else {
        res.history.add_verified(v)
```

```
        propagateDirtyFromDeps(res)
        return res
      }
    }
    None: recompute(k)
  }
}
```

The result cache will return the entries based on the history stored in the node. If the `verified_at` matches the supplied version, we know that the entry is a match. Otherwise, it's a mismatch because we are unsure if the version is valid.

```
resCache.get(k, v) -> Result {
  match self.map.get(k) {
    None(r) => Result::None.
    Some(r) => {
      if r.history.verified_at.contains(v) {
        return Result::Match(r)
      } else {
        return Result::Mismatch(r)
      }
    }
  }
}
```

To check if dependencies actually changed for a dirtied node, we re-evaluate dependencies and check if there are any versions for which the dependencies and the versions that this result was computed at is the same. We can do so by checking if there exists a version `v` that is in both `res.history.verified_at` and, for all dep in deps, `v` is in `dep.history.verified_at` for `dep` evaluated at the target version. Since `verified_at` indicates the versions that evaluated to the same result, this check would indicate that for the current target version, there is some version that the deps currently are unchanged from, which means we can reuse the old result since none of the deps changed.

```
isDepsChanged(k: K, v: VersionNumber, res: Res) -> bool {
  for (d : res.deps) {
    dres = compute(d, v)
      if (intersect(dres.history.verified, res.hist.verified,
deps.versions.verified) not empty) {
        continue; // nothing changed
      } else {
        return true;
```

```
      }
    }
  return false
}
```

When re-evaluating a node, we check if the new result is actually equivalent to the existing result because if so, then we can reuse the result for dependant computations, so we update the current history to include the current version number. We also update the deps and rdeps. Now, since for dirtying, we only dirty when nodes are up to date, we do a delayed propagation of dirty at the new computed version up the graph. Locking the node ensures all data relevant to one version is consistent, and the versioning will guarantee the thread safety.

```
recompute(k: K, v: VersionNumber, old: Node) -> Res {
  computeIndex.put((k,v), async {
    t = usercompute
    if (t == old.t) {
      old.lock { // atomic operations in this scope
        last_change = max(current_deps.verified_at.begin)
        // the interval from which deps last changed to v
        // is all valid
        old.history.mark_verified((last_change, v))
        old.deps.update(current_deps)
        // also use difference in deps to update rdeps

        propagateDirtyFromDeps(old)
      }
      return old
    } else {
      node = Node(t, deps, rdeps, v)
      node.lock {
        last_change = max(current_deps.verified_at.begin)
        // the interval from which deps last changed to v
        // is all valid
        old.history.mark_verified((last_change, v))

        propagateDirtyFromDeps(old)
        resCache.put(Res(t, [k.0]))
      }
    }
  })
  computeIndex.get((k,v)).await
}
```

Delayed Dirty Propagation

Delayed propagation of dirty is needed because we didn't mark all nodes when traversing rdeps immediately due to the check for dirty versions in our invalidation dirtying algorithm. As such, when we recompute or determine a node as re-usable, we need to carry over any newer dirtie's from the deps.
We only need to propagate the earliest "dirty" from any of its deps, as we can rely on the recomputation from that dirty version to re-propagate any newer "dirty" as needed.

```
propagateDirtyFromDeps(node) {
  // then also carry over future dirties
  d = the the minimum in node.deps.history.dirtied where d > v
  node.history.dirty(d)
}
```

Run Times

The invalidations algorithm traverses the entire invalidated set. For each node, since marking dirty versions is always strictly increasing version numbers, in a sorted history, marking intervals and dirty is always `O(1)`. This makes for at worst a total of `O(invalidated set)` for invalidations, less if nodes aren't up to date, for which the work is deferred to when new nodes are computed.

The deferred propagation of dirty takes `log(h)` time to find the dirty and mark it per dependency where h is the length of history.

When requesting a value, if the value has already been requested and verified, this is a simple key lookup, followed by a contains validation in a sorted version history that is `O(log(h))`. Checking the history matching on dependencies when the value isn't verified will be `O(h)` to do intersection on sorted lists. We also do deferred propagation where we check the deps of the invalidate subset which is around `O(log(h) * invalidated subset)`. Together, we check the entire invalidated subset for a total of `O(h * invalidated subset)` work for cached scenarios under `O(invalidated subset)` node visits.

We only actually recalculate a node if dependencies are changed, and we skip re-computation of dependants if the current node evaluates to the same value for a total of `O(changed subset)` recomputations. For a recomputation, we also update the history which involves doing up to `O(log(h))` work for each dep. We also update the rdeps for each dep, or merge the history from each dep and perform dirty propagations. In total, we visit `O(changed subset)`

number of dependencies for a total of `O(h * changed subset)` additional overhead to recomputing. Note that we do still traverse the entire `O(invalidated subset)`.

If the value is never seen, analysis is not interesting as its just `O(mountain)` and there's nothing we can do.

One interesting caveat of the delayed propagation is that after an initial dirty that takes `O(invalidated set)`, for subsequent invalidations, we only need to defer propagate dirty for `O(invalidated subset)` nodes of the requested nodes since the initial dirtying instead of `O(invalidated set)` of the whole graph. Effectively, it lets us reduce our invalidation traversals to only the graphs portions that we've most recently requested after one complete dirtying of the graph.

Correctness

The main property for correctness in an incremental graph computation is that for a node `n` that was previously computed at `v0`, for the immediate next request at some version `v1`, if for any `d in node.deps`, `d.res at v0` != `d.res at v1`, then `n` is recomputed, else `n` does not need to be recomputed.

We can show the above correctness property:

(1). A node in our system is never changed from a previous version `v-1` unless the history was marked as dirty at `v`.
In our system, at version `v-1`, changes must begin by users calling `changed` on dice, which explicitly marks the history of that node as dirty. Changes are committed atomically to graph by incrementing the version number, so changes are only committed to the new version `v - 1 + 1 = v`. We also know that no nodes can be on the graph at `v+1` until the changes at `v` have been committed by nature of our versioning.

a. For any existing nodes in the graph at versions `v-1`, our `changed` algorithm will traverse rdeps to mark any node that might be affected as dirty at version `v` so that their results are only reusable up to `v` and guaranteeing (1).
b. For nodes that are yet at `v-1`, we check if they have been dirtied at a version `< v`. If they haven't, then they have no further changes until `v` since all changes are committed atomically at increasing versions, so we mark them as changed, guaranteeing (1). If they have been dirtied at a `v0 < v`, then those nodes will be re-evaluated at some point, and this falls into case c.
c. For nodes that need to be checked at some version `< v - 1`. When we either recompute, or reuse a previous result, we call `propagateDirtyFromDeps`, which will inherit dirty versions from the node's deps. Either the inherited dirty version is `< v`, in which case we fall back to case b, or the dep is never dirtied at `v` in which case we don't care, or the dep is dirtied at `v`, and we inherit it, satisfying (1).

(2) A node has result `r1` at versions `v1` and `r2` at `v2`, `r1 != r2`, then `history.verified_at.contains(v1) && history.verified_at.contains(v2) == false`.
A node's value can only change if the history was dirtied by (1). If it's dirtied, the new value must be recomputed, upon which, we check if the values are different, and if so, we put a new entry where the new history does not contain the old versions, so (2) is true.

(3). A node has result `r` at versions `v1` and `v2`, then `history.verified_at.contains(v1) && history.verified_at.contains(v2) == true`.
Now by there's a few cases:

a. The node is never marked dirty between `v1` and `v2`, for which its never changed since it's never recomputed..
b. The node is dirtied, and recomputed, but the result is the same. In this case, our `recompute` checks for equality with the previous values, and adds the new version if equal.
c. The node was dirtied since `v1`, but the dependencies were found to satisfy `history.verified_at.contains(v1) && history.verified_at.contains(v2) == true` for every dep. This, by induction, means that the dependencies `r` has not changed, which means that this node couldn't have possibly changed, and we mark the node as verified at `v2`, guaranteeing (3).

(4). A node in the graph is never recomputed if we know it won't change at some `v`.

a. Either this was never marked dirty, in which case we just return the node.
b. The dependencies never changed, for which this result can never change. We detect this by validating that the dependencies evaluated at `v` contains `v'`, where the current node also contains `v'`. By (3), we know that the dependencies values didn't change between these versions, so we can reuse the current nodes result at `v'` and not recompute it.

Together, these mean that we always avoid recomputation where possible, but never fail to recompute when the values are changed.

Alterations for other Performance Characteristics

For our `changed` algorithm, we can recompute the value at time of dirtying. This allows us to check if the value actually changed and decide whether to further propagate the dirty. The early cut-off of dirtying reduces the invalidations to `O(computed set)`, and requests will be `O(1)`
This change would still be compatible with the existing algorithms, and can be mixed such that

some nodes of the graph uses eager recomputation while other parts of the graph uses lazy recomputation. However, the amount of recomputation we perform will be `O(computed set)` instead of `O(changed subset)`.

We can reduce invalidations closer to `O(invalidated subset)` in the original algorithm and `O(changed subset)` in the eager recomputation algorithm by introducing a “graph partition” formed based on the computation mountains. When dirtying, we can delay the invalidation traversal until the next request is received, upon which we enter the relevant graph partition(s) of those requests, and perform the invalidations logic traversing only rdeps belonging to the relevant partition(s). When we hit other partitions, we’ll mark those nodes as dirty at those versions, but do not traverse rdeps beyond that. Those will occur when that graph partition is loaded. Depending on how well we can form our subsets, and the complexity of discovering so, we can get close to `O(invalidated subset)` and `O(changed subset)` traversals if we do bottom up computation. The details of how to do the partitioning will be figured out in the future. The rest of our algorithm still applies within each graph partition withoutft any alterations needed.