QuantumGraph Generation Algorithm Updates for DM-19988

The design on this page should be considered obsolete - it doesn't actually solve the problems I was hoping it would solve. Some of the discussion is still useful, as I think it provides at least some hints at a general path forward. In the meantime, I think I have a much smaller modification to the current in algorithm in hand that should address the DM-19988 issue (while leaving more challenging problems - particularly the "jointcal problem" noted below) for the future.

The Main Problem

DM-19988 - Jira project doesn't exist or you don't have permission to view it. is a symptom of a fairly deep problem in the current QuantumGraph generation algorithm: the algorithm at present assumes the set of data IDs to be processed is defined by a strict intersection of all relevant dimensions, according to their (usually spatial) relationships. Because dimensions also provide a form of discretization that expands what's covered by that intersection, this does what we want as long as there is just one pairwise relationship in the query (i.e. visit_detector_region to patch), but it falls apart when that's not the case, such as:

- In order to process a visit+detector, we also need to do a spatial query on skypix to find reference catalogs. But strict intersection can filter out any skypix entries that don't overlap the tract or patch of interest, even if they do overlap a visit+detector that overlaps the tract or patch of interest. This is the actual DM-19988 case.
- If some task (e.g. jointcal) needs full visits, including some visit+detectors that may not overlap a tract or patch of interest. And if the graph includes single-epoch processing that feeds jointcal, we need to generate quantums for those non-overlapping detectors, too.

Side Problems

The QuantumGraph generation algorithm has a few other known issues we'd love to fix at the same time if a rework presents an opportunity for this:

- It's *slow*. The Big Join Query (hereafter BJQ) is so big and complex that it must be confusing the query optimizer pretty badly (or perhaps it's confusing us, by making it unclear which indexes we need to add to get the right plan), because these queries take much longer than they should.
- The strict-intersection problem actually hit us once already, in the temporal matching of calibration_label to exposure. We worked around that by adding the "per-DatasetType dimension" concept, which was an attempt at generalization that still feels more like a case-specific band-aid. Temporally matching in calibration products feels a lot like spatially matching in reference catalogs, and a common solution to both problems would make me a lot more confident that such a solution actually is more than a case-specific workaround.
- We currently assume that a Quantum should be generated if there is at least one input dataset for each of its input dataset types. That's usually what we want, but tasks that expect multiple inputs of a particular type may require more than one to exist, and some tasks may not need any instances of a particular dataset type (or even expect none!), such as fringe frames in ISR for bluer filters. We already have a workaround for the first case, by having Tasks without enough inputs produce dummy outputs that can be correctly consumed by later stages, and a special-case fix for the latter is underway on DM-16805 - Jira project doesn't exist or you don't have permission to view it..
- This isn't a problem *per se*, but we've identified recently that it'd be nice if QuantumGraphs were a bit more reusable. That's mostly taken the form of talking about updating the Task software versions or configuration without needing to regenerate the graph, which I think is *mostly* orthogonal to the topics on this page, but I think being able to apply the same QuantumGraph to different input collections would also be quite useful, and *i* potentially related. That might look like first generating a QuantumGraph with no fully-resolved dataset_ids, just data IDs (even if the presence or absence of some inputs in one collection is still part of generating the graph), and then being able to "apply" (need a better verb for this) that graph to other collections, yielding a new QuantumGraph with dataset_ids and possibly some Quanta filtered out (either because they represent completed processing already done in that collection, i.e. the manual-retry case, or because some needed inputs aren't present in that collection).

Proposal

At least for now, I think we should just encode a lot of knowledge about the concrete set of dimensions we have into the algorithm. We can look for ways to generalize that later, but I think this plan will work for all current and known-future PipelineTasks in any combination, and it may well be easier to explicitly manually expand a bit rather than fully generalize when we encounter PipelineTasks that don't fit (like CPP tasks, which wouldn't have fit with the old algorithm, either).

We'll start with a BJQ formed with the following logic:

- Join in any instrument-dependent dimension tables that are part of any Task's *quantum* dimensions, and then recursively expand to include implied dimensions (e.g. visit implies physical_filter).
- Join in any skymap-dependent dimension tables that are part of any Task's *quantum* dimensions, [and then recursively expand, but this will always do nothing].

- If the joined dimensions include both "instrument" and "skymap", relate the above dimensions only via the broadest spatial relationship: visit_tract_join (other spatial join tables are ignored). As before, we will filter out non-overlaps using the actual visit and tract regions in Python processing of the result rows.
- If abstract_filter is included in any Task's quantum dimensions, join it in if it has not already been included (by expanding physical_filter).
- Join in a dataset subquery for any input datasets that are not also outputs and are not marked as "prerequisite" (these are the "required inputs").
- SELECT DISTINCT from the joined dimensions, but do not attempt to fetch matched dataset_ids at all - instead we aggregate over them with that DISTINCT. (Would a GROUP BY on all selected fields be better or worse than DISTINCT for this? Seems like it ought to be identical.)

The BJQ now thus yields DataIds only, not dataset_ids, and the first-stage QuantumGraph we create will as well. We'll construct that from the query results with the following logic, iterating over the Tasks in the Pipeline in **reverse order**:

- Construct a set of *proposal* Quanta by selecting rows from the BJQ outputs that are unique over just the Quantum dimensions of this Task.
- Optionally filter the proposal Quanta by keeping only those that produce outputs that are consumed as inputs by a (later) Quantum already in the graph. Whether we do this step should be controllable by the user, probably via some kind of per-DatasetType "minimal vs. maximal" setting - a Quantum is not filtered if it produces any DatasetTypes configured as "maximal".
- Attach unresolved (no dataset_id) DatasetRefs to each Quantum for all regular inputs and outputs - *not* prerequisite inputs.

This gives us a QuantumGraph that should be valid for the collection(s) used to create it, as long as we continue to assume that at least one input of each dataset type is enough to make a valid Quantum. I think it makes sense to follow that up with a new stage that attempts to resolve input DatasetRefs to the datasets in a particular collection (which may now differ from the one used to construct the initial QuantumGraph). This stage would iterate through Tasks in **forward order**:

- Aggregate across all Quanta for the Task the lists of data IDs for each input and output DatasetType for that Task.
- Perform a *bulk* follow-up query for all instances of each DatasetType. Cache the results and use these to avoid duplicate queries as iteration proceeds
- Iterate over all of the Task's Quanta again, attaching resolved DatasetRefs and removing Quanta for which inputs are not present in either the follow-up query results or the set of output DatasetRefs from previous Quanta. Resolve cases where some or all outputs already exist, raising exceptions or filtering out those Quanta (as indicated by user settings).

This last stage would now be one that could be repeated (starting from the original QuantumGraph) with different sets of input collections.

Implementation Details

I'm currently planning to put most of the initial code for this in pipe_base rather than daf_butler, which may involve having a lot SQLAlchemy-dependent code there. That will involve having that new code access private members of SqlRegistry for now, but I already have plans to make public interfaces for Registry to obtain SQLAlchemy "selectables" (tables, views, subqueries) that can be used for this in the future. Unlike the previous incarnation of this code, it seems too specific to QuantumGraph generation to belong in daf_butler. We may then be able to retire or simplify a lot of the QueryBuilder stuff in daf_butler (as it now seems at least some of that is more specific to QuantumGraph generation that it seemed at the time it was added).

Limitations

This design prohibits (for now) Tasks with label, calibration_label, or skypix in their quantum dimensions (note that we have no such Tasks right now) or output DatasetTypes. Input DatasetTypes *may* use these dimensions, regardless of whether they are prerequisites (but right now all DatasetTypes that do use these dimensions are in fact marked as prerequisites).

Future Extensions

- It would make a lot of sense to delegate to each Task the responsibility of filtering out Quanta that don't have the right number of inputs, as it's the Tasks that can determine that best. This would probably work best as an abstract classmethod on PipelineTask, but an abstract instance method or a helper class should be on the table, too. A complication is that the code that does this will probably want to retrieve information from the Registry in order to know how many inputs they *could* be getting, such as the number of detectors that could appear in a visit, or the number of visits that could have been included in a coadd. It's not clear how best to define an API to do that.
- If we have two stages of QuantumGraph generation, and the dataset_ids and prerequisite inputs are only included at the second stage, we should consider whether we should use the same QuantumGraph and Quantum classes to represent both stages. One possibility might be to have a class hierarchy for each, separating common functionality into base classes. This is closely related to the question of whether DatasetRefs should always represent "resolved" datasets.
- If the QuantumGraph BJQ only ever uses visit_tract_join for spatial relationships, we may be able to drop the other join tables. This is closely related to the refactoring underway (but now probably deferred until

  DM-19988 - Jira project doesn't exist or you don't have permission to view it.

  is done) on

  DM-17023 - Jira project doesn't exist or you don't have permission to view it.

  .