Proposal: Dataset Views

The Dataset API needs a simple way for users to express logical constraints and then apply them to efficiently limit readers and writers.

This enables readers to both *easily* and *efficiently* work with subsets of the data, even if those subsets don't align with partition boundaries. For example, under the current API, partitions are exposed to users directly and readers are responsible for any filtering within a partition. A reader running a crunch job over a logical subset that crosses two physical partitions must read each separately, apply a filter function, and then union the two collections (not *easy*). To avoid this, a user may simply read an entire partition higher in the hierarchy, reading more data than necessary but filtering just once (not *efficient*).

This enables writers to have minimal guarantees about how additions will be written and to enforce sanity checks. The current partition API primarily uses an unrestricted writer to add data. With unrestricted writers, errors can be persisted anywhere within the Dataset. For example, log events from a host with a misconfigured time zone are stored in an unexpected partition, sometimes creating a small additional file. Logical errors when writing in parallel can cause writers to overlap and create more, smaller files. These small files accumulate without warning and hurt map/reduce performance.

Requirements

The first version of this API was focused on expressing constraints for the partition strategy. In addition to needlessly exposing the storage details, this led to the assumption that multi-field constraints[1] are generally useful. This version approaches the problem with the assumption that users want to express constraints about their data and have partition selection applied transparently. This expands the scope from filtering partitions to filtering entities, but fixes the API problems.

- Constraints should be expressed independently, as in SQL
- Multi-field constraints should be handled as a special case (if necessary)

Because this new approach expands the scope, this proposal needs to address all of the commonly-supported logical operations. This divides those operations into required and optional. Required operations are:

- Filtering: exists, equality, inequality
- Boolean: and, or
- Column selection or projection (when reading)

[1] This is what led to needing a mutli-field `Boundary` or `Marker`.

Optional operations are:

- Filtering:
    - set inclusion: identical to equality with and
    - has all, has any: identical to exists with booleans
    - like: beyond initial scope
    - custom predicates: beyond initial scope
- Boolean not: mostly satisfiable with filtering[2]

Proposed operation API

```
View<E>:
  // from the current interface
  DatasetReader<E> newReader()
  DatasetReader<E> newWriter()
  // get the covering partitions
  Iterable<View> getCoveringPartitions()
  // delete all records in the view
  boolean deleteAll()
  // check if the view contains the given record
  boolean contains(E)
```

The operation methods are not changed, except that the `contains(Marker)` method is no longer needed. `newReader` and `newWriter` are factory methods that return readers or writers that enforce the view's constraints. `getCoveringPartitions` allows callers insight to the underlying partition structure, and `deleteAll` will delete all records in the view (where `contains` would return true) or throw an exception if not supported.

Proposed expression API

```
View<E>:
  // logical or
  View<E> union(View<E> v)
  // exists
  View<E> with(String name)
  // equality
  View<E> with(String name, Object value)
  // inequalities
  View<E> from(String name, Object value)      // >=
  View<E> fromAfter(String name, Object value) // >
  View<E> to(String name, Object value)        // <=
  View<E> toBefore(String name, Object value)  // <
  // projection (only when reading)
```

[2] "Not exists" is not possible, but altering constraints can handle the other cases.

```
  View<E> select(String... columns)

Views:
  // logical or
  <E> View<E> unionOf(View<E>... views)
  // logical not
  <E> View<E> complementOf(View<E> view)
```

With independent constraints on data fields, each end-point is a single value for a field. This eliminates the need for a `Marker` object, which represented a multi-field end-point (for the partition strategy). Function calls require both a field name and a value as a consequence of removing `Marker`, which previously held one or more name/value pairs.

As a consequence of independent constraints, each method call results in a view backed by a well-defined (unambiguous) expression when using the semantics from the earlier API: each equality or inequality constraint must be limiting.

Using this API is almost identical to the previous version, except that `View#of(Marker)` is replaced with an equality view:

```
  // users with favorite color orange
  View<User> orange = users.with(“favoriteColor”, “orange”)
  // in the previous version:
  Marker orange = new Marker.Builder(“favoriteColor”, “orange”).build()
  View<User> v = users.of(orange);
```

There is no longer a separate, non-view object that holds constraints.

The main drawback is that callers are required to use the same domain as the data, which is not always simple to construct. To express a time interval:

```
  // events between 4 October and 10 April
  long OCT_4 = new org.joda.time.DateTime(2012, 10, 4, 0, 0).getMillis();
  long APR_10 = new org.joda.time.DateTime(2013, 4, 10, 0, 0).getMillis();
  View<Event> v = events.from(“timestamp”, OCT_4).to(“timestamp”, APR_10);
```

The `select` method registers a list of columns that should be projected.

View refinement

View methods refine the view on which they are called. Each new view has the constraints from the old, with an added constraint from the method call that produced it. In the above example, the first “from” method call produces a view with one constraint, which is used to create a new view by calling “to” to add a second boundary for the timestamp.

Chaining method calls produces an “and” expression, where each constraint must be satisfied, while an “or” expression is produced by the union of two views. When chaining constraint methods, java's execution order is used: "from" must be executed to produce the view refined by "to". This applies universally, and any additional constraints are applied to the entire view on which a constraint method is called. This means that views do not keep track of the "last" constraint added to alter just that constraint.

If a view already has a list of selected columns, additional calls to "select" add columns to the selected set.

API limitations

- There is no proposed way to un-select columns without rebuilding a view
- Union creation is duplicated between Views.unionOf and View#union. We should probably decide on which one is more clear.
- Views.complementOf may not be necessary if not-equals is added because views could be constructed using the complement of individual operations. Adding a "without" or some other method for not-equals may be a better solution than view complement.
- Some chained constraints look odd because they appear to be related:
    - `events.from("logLevel", Levels.INFO).to("time", 12345678)`
    - One solution is to rename inequality methods. Guava uses "atLeast" and "greaterThan":
      `events.atLeast("logLevel", Levels.INFO).lessThan("time", 12345678)`

Data limitations

In the previous version, views were tied to the partition strategy, which defined an ordering for all of the fields that supported constraints. In this proposal, potentially any field could be used in an inequality. This leads to cases where constraints are not reasonable. For example, using "from" on a map field isn't reasonable because map fields are unordered.

One way to cope with this problem and limit scope is to restrict constraints to fields in the partition strategy, which are comparable, or to restrict constraints to fields with primitive types, with well-defined orderings.

Another option is to remove the inequality methods, `to`, `from`, etc., and add `with(String name, Range range)` to use a guava-like `Range`. This puts the responsibility for constructing a coherent range on the caller.

Multi-field constraints

The use case for multi-field constraints may still exist as a special case, if a data set has fields that relate to one another as a hierarchy. The simplest example is a date stored as separate year, month, and day fields. Each field is independent, but expressing meaningful constraints independently is tedious:

```
  // events between 4 October and 10 April
  View<Event> oct = events.with(“year”, 2012).with(“month”, 10).from(“day”, 
4);
  View<Event> nov_dec = events.with(“year”, 2012).from(“month”, 
11).to(“month”, 12);
  View<Event> jan_mar = events.with(“year”, 2013).from(“month”, 1).to(“month”, 
3);
  View<Event> apr = events.with(“year”, 2013).with(“month”, 4).to(“day”, 10);
  View<Event> whole_range = Views.unionOf(oct, nov_dec, jan_mar, apr);
```

The ideal solution is to use a single ordered field rather than a hierarchy, but there may be cases where this isn’t possible. In that case, callers must supply a domain that describes their hierarchy in terms of the data’s existing fields. An extension of the general API may look like this:

```
  // events between 4 October and 10 April
  TupleDomain time = Tuple.domain("year", "month", "day")
  events.from(time, Tuple.of(2012, 10, 4)).to(time, Tuple.of(2013, 4, 10))
```

This improves upon the `Marker` version by not assuming the ordering given by the partition strategy. By adding the domain users are forced to specify the fields they are using separately and can use more familiar `Tuple` objects. This also expresses the multi-field constraint in a single method call that can return a well-defined view.