Declarative language on nested structures

Jim Pivarski

Princeton University – Project DIANA

September 16, 2016
StrangeLoop Unsession

Motivation

In High Energy Physics (HEP), >99% of data analysis code is

- compiled C++ (modules fitting into a larger framework)
- *interpreted* C++ (don't ask)
- Python (the young and hip)

Motivation

In High Energy Physics (HEP), >99% of data analysis code is

- compiled C++ (modules fitting into a larger framework)
- *interpreted* C++ (don't ask)
- Python (the young and hip)

But they are fairly regular operations, at least for data-reduction:

- filtering events of interest (skimming)
- selecting variables of interest (slimming)
- evaluating simple mathematical operations, e.g.
  - invariant mass: $\sqrt{(E_1 + E_2)^2 - |\vec{p}_1 + \vec{p}_2|^2}$
  - min/max of a collection
  - averages, standard deviations, ...

Motivation

In High Energy Physics (HEP), >99% of data analysis code is

- compiled C++ (modules fitting into a larger framework)
- *interpreted* C++ (don't ask)
- Python (the young and hip)

But they are fairly regular operations, at least for data-reduction:

- filtering events of interest (skimming)
- selecting variables of interest (slimming)
- evaluating simple mathematical operations, e.g.
  - invariant mass: $\sqrt{(E_1 + E_2)^2 - |\vec{p}_1 + \vec{p}_2|^2}$
  - min/max of a collection
  - averages, standard deviations, ...

Could be declarative: simple for users, optimized on the backend!

Typical data structure

Tree-like with a known schema. *Maybe* with cross-references (e.g. showers matched to tracks), but at least a DAG you can pretend is a tree.

```
event: struct
  |
  +--- timestamp: bigint
  +--- missing energy: float
  +--- tracks: array of struct
  |      |
  |      +--- momentum: float
  |      +--- theta angle: float
  |      +--- hits: array of struct
  |             |
  |             +--- detector id: int
  |             +--- charge: float
  |             +--- time: float
  |             +--- ...
  +--- showers: array of struct
         |
         +--- ...
```

Typical queries

Broad classes:

Reduce-like: transforms a set of records into a single record (maybe different type). Aggregations: e.g. Histogrammar.

Map-like: transforms a set of records into another set of records (maybe different type). Looking for a solution.

Typical queries

Broad classes:

Reduce-like: transforms a set of records into a single record (maybe different type). Aggregations: e.g. Histogrammar.

Map-like: transforms a set of records into another set of records (maybe different type). Looking for a solution.

Simplification: the source dataset is *read-only* and is *not* updating in real-time. No transactions are taking place.

Typical queries

Broad classes:

Reduce-like: transforms a set of records into a single record (maybe different type). Aggregations: e.g. Histogrammar.

Map-like: transforms a set of records into another set of records (maybe different type). Looking for a solution.

Simplification: the source dataset is *read-only* and is *not* updating in real-time. No transactions are taking place.

Examples:

- "Momentum of the track with the most hits with theta between -2.4 and 2.4."
- "Average charge of all hits with time in 0–10 ps on all tracks with momentum greater than 10 GeV/c."
- "Weighted average theta of the two tracks with highest momentum."

Illustrations in Scala

```
event: struct
  |
  +--- timestamp: bigint
  +--- missing energy: float
  +--- tracks: array of struct
  |      |
  |      +--- momentum: float
  |      +--- theta angle: float
  |      +--- hits: array of struct
  |             |
  |             +--- detector id: int
  |             +--- charge: float
  |             +--- time: float
  |             +--- ...
  +--- showers: array of struct
         |
         +--- ...
```

- "Momentum of the track with the most hits with theta between -2.4 and 2.4."

```
// missing check for when zero tracks passed filter!
{event => event.tracks                              // get tracks
                .filter(abs(_.theta) < 2.4)         // theta range
                .maxBy(_.hits.size)                 // most hits
                .momentum                           // return momentum
}
```

Illustrations in Scala

- "Average charge of all hits with time in 0–10 ps on all tracks with momentum greater than 10 GeV/c."

```
{event => mean(
            event.tracks                          // get tracks
                 .filter(_.momentum > 10)         // in momentum range
                 .flatMap(_.hits)                 // explode to hits
                 .filter(_.time < 10)             // in time range
                 .map(_.charge)                   // return charges
              )}                                  // to mean function
```

Illustrations in Scala

- "Average charge of all hits with time in 0–10 ps on all tracks with momentum greater than 10 GeV/c."

```
{event => mean(
            event.tracks                          // get tracks
                 .filter(_.momentum > 10)         // in momentum range
                 .flatMap(_.hits)                 // explode to hits
                 .filter(_.time < 10)             // in time range
                 .map(_.charge)                   // return charges
              )}                                  // to mean function
```

- "Weighted average theta of the two tracks with highest momentum."

```
// again missing check for less than two tracks!
{event => val List(one, two) =                    // unpack and assign
              event.tracks                        // get tracks
                   .sortBy(_.momentum)            // sort by momentum
                   .take(2)                       // take two
          // now compute the weighted mean of the two structs
          (one.theta*one.momentum + two.theta*two.momentum) /
              (one.momentum + two.momentum)
}
```

Use SQL?

It is valuable to use pre-existing standards if applicable. SQL supports "arrays of structs," can we use it?

Use SQL?

It is valuable to use pre-existing standards if applicable. SQL supports "arrays of structs," can we use it? **No.**

```
WITH hit_stats AS (
 SELECT hit.track_id, COUNT(*) AS hit_count FROM hit GROUP BY
     hit.track_id
),
track_sorted AS (
   SELECT track.*,
   ROW_NUMBER() OVER (
    PARTITION BY track.event_id
    ORDER BY hit_stats.hit_count DESC
   )
 track_ordinal FROM track
   INNER JOIN hit_stats ON hit_stats.track_id = track.id
   WHERE track.theta > -2.4 AND track.theta < 2.4
)

SELECT * FROM event
  INNER JOIN track_sorted ON track_sorted.event_id = event.id

WHERE
  track_sorted.track_ordinal = 1
```

Use RDF or a graph query language?

Maybe. This is SPARQL:

```
PREFIX : <http://yournamespace.com/accelerator/> .

SELECT ?momentum (MAX(?hitcount) as ?maxhits)
WHERE {
    SELECT ?momentum (COUNT(?hits) AS ?hitcount)
    WHERE ?track :momentum ?momentum .
          ?track :theta ?theta .
          FILTER (?theta > -2.4 AND ?theta < 2.4) .
          ?track :hits ?hits
    GROUP BY ?track
}
GROUP BY ?momentum;
```

Use RDF or a graph query language?

Maybe. This is SPARQL:

```
PREFIX : <http://yournamespace.com/accelerator/> .

SELECT ?momentum (MAX(?hitcount) as ?maxhits)
WHERE {
    SELECT ?momentum (COUNT(?hits) AS ?hitcount)
    WHERE ?track :momentum ?momentum .
          ?track :theta ?theta .
          FILTER (?theta > -2.4 AND ?theta < 2.4) .
          ?track :hits ?hits
    GROUP BY ?track
}
GROUP BY ?momentum;
```

XPath? (Idea from today's Cypher talk...)

Use RDF or a graph query language?

Maybe. This is SPARQL:

```
PREFIX : <http://yournamespace.com/accelerator/> .

SELECT ?momentum (MAX(?hitcount) as ?maxhits)
WHERE {
    SELECT ?momentum (COUNT(?hits) AS ?hitcount)
    WHERE ?track :momentum ?momentum .
          ?track :theta ?theta .
          FILTER (?theta > -2.4 AND ?theta < 2.4) .
          ?track :hits ?hits
    GROUP BY ?track
}
GROUP BY ?momentum;
```

XPath? (Idea from today's Cypher talk. . .)

I'm on the fence about this.

RDF is pretty well established and graph query languages are becoming established, but they introduce more concepts than are needed for this problem.

Having to address unnecessary concepts in every query (e.g. saying that you're looking for events with tracks when *every* event has tracks) is distracting.

What I've been thinking

From a user's perspective, I liked the Scala examples best (easily translatable into any other functional syntax).

But the semantics should also be declarative to develop better backend optimizations.

What I've been thinking

From a user's perspective, I liked the Scala examples best (easily translatable into any other functional syntax).

But the semantics should also be declarative to develop better backend optimizations.

Idea: introduce "higher order functions" to select-where SQL?

- add syntax like `x -> sin(x) + 3*x` to expressions
- add methods like the following to arrays
  - `COLLECTION.map(function1)`
  - `COLLECTION.reduce(function2)` *not* a DAF: operates only on *tracks within one event*.
  - `COLLECTION.count`
  - `COLLECTION.sum`
  - `COLLECTION.max(function1, N)` top-N subcollection.
  - `join(COLLECTION1, COLLECTION2)` *not* joining events, but *tracks within one event* (small).
  - `COLLECTION.pairs` like the above, but unique pairs of indexes.
- add assignments (syntactic sugar).

Other cases. . . ?

Common task: match simulated particles with their reconstructed counterparts within each event.

- Might assign same reconstructed to multiple simulated:

```
simulated.map(s -> (s, reconstructed.map(r -> dist(s, r)).min))
```

Other cases. . . ?

Common task: match simulated particles with their reconstructed counterparts within each event.

- Might assign same reconstructed to multiple simulated:

```
simulated.map(s -> (s, reconstructed.map(r -> dist(s, r)).min))
```

- Might assign same simulated to multiple reconstructed:

```
reconstructed.map(r -> (simulated.map(s -> dist(s, r)).min, r))
```

Other cases. . . ?

Common task: match simulated particles with their reconstructed counterparts within each event.

- Might assign same reconstructed to multiple simulated:

```
simulated.map(s -> (s, reconstructed.map(r -> dist(s, r)).min))
```

- Might assign same simulated to multiple reconstructed:

```
reconstructed.map(r -> (simulated.map(s -> dist(s, r)).min, r))
```

- Have to introduce new built-in function to get exclusive matches:

```
bestMatch(simulated, reconstructed, (s, r) -> dist(s, r))
greedyMatch(simulated, reconstructed, (s, r) -> dist(s, r))
greedyMatch(reconstructed, simulated, (r, s) -> dist(s, r))
```

Middle approach

Might develop new language that is *heavily inspired by* existing languages?

These types of queries are not too domain-specific, even `bestMatch/greedyMatch`. Does it have other applications?

What suggestions do *you* have? Help me shape a language (or convince me that I should really use language X)!

Now I stop talking.