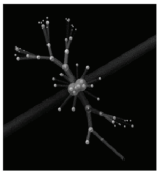
Optimization and Execution of Complex Scientific Queries over Uncorrelated Experimental Data

Rishi and Ronnie

LHC - ATLAS

- Produces large streams of *particle collision events*
  - Generate new particles
- Problem: Find interesting particles produced by collisions
  - E.g. Higgs bosons
- Recorded in large # of large binary files
  - C++ framework ROOT
- Scientists develop filters called *cuts*
  - Matched against streams of collision events



Data Analysis

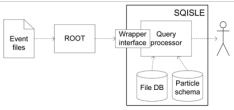
Conventional approach:
- Scientists express cuts as C++ programs calling ROOT
- Difficult to implement (reluctant programmers), bad code, difficult to understand and change

Our approach:
- Define cuts as *queries over streams of complex objects*
- Wrap ROOT to access collision event files in one pass

Challenges
- Large data volume
- Complex queries
- No a priori data statistics
- Very demanding performance requirements (c.f. C++)

SQISLE: *Runtime and mode changing query optimization, and query rewrites*

SQL interface




```
1: select e
2: from Event e, EventFile f
3: where name(experiment(f)) = "bkg2" and
4:     fileid(f) < 15 and
5:     e in events(filename(f)) and
6:     hadrtopcut(e) and jetvetocut(e) and
7:     misseecuts(e) and zvetocut(e) and
8:     threeleptoncut(e) and leptoncuts(e);
```

Source selection
Stream extraction
Cuts

Query plan?

```
6:     hadrtopcut(e) and jetvetocut(e) and
7:     misseecuts(e) and zvetocut(e) and
8:     threeleptoncut(e) and leptoncuts(e);
```

- Plan of these cuts contains
  - 22 operators and
  - 8 of them are aggregate functions over nested subqueries
- Plan of each nested subquery contains
  - between 9 and 59 operators
  - including further nested subqueries

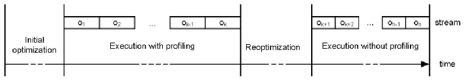
=> *Good optimization of the cuts query fragment is difficult*

Properties of data described in the paper

- Uncorrelated collision events
  - => *No joins* between the complex objects
- Most events are not interesting
  - => Queries testing hypoteses are *selective*
- Events are produced in controlled experiments
  - => Statistical properties are similar if they are from the same experiment

Query optimization

- Aggregate cost model
  - Estimates cost and selectivity of aggregate functions by cost and selectivity of its nested subqueries
- Mode-changing runtime query optimization
  - Operator to profile query execution and run-time reoptimize
  - Operator dynamically changes modes



Three strategies

- Attribute statistics profiling
  - Monitor statistics on attributes of complex objects
    - e.g. number of electrons produced
  - Runtime re-optimization when sample is big enough
- Group statistics profiling
  - Decompose query into minimal *groups* of subqueries uncorrelated per complex object
  - Monitor statistics per group
  - Check if statistics affect join-order of groups by runtime optimization
- Two-phase statistics profiling

  Mode 1: Attribute statistics profiling to produce cost model to runtime optimize group subqueries

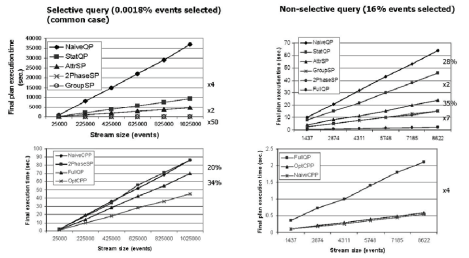
  Mode 2: Group statistics profiling to runtime optimize group join-order

  Mode 3: No profiling

Evaluation

- **NaiveQP** – no aggregate cost model, no runtime query optimization, default statistics
- **StatQP** – aggregate cost model, no runtime query optimization, default statistics
- **AttrSP** – M1: attribute statistics profiling M2: none
- **GroupSP** – M1: group statistics profiling M2: none
- **2PhaseSP** – M1: AttrSP, M2: GroupSP, M3: none
- **FullQP** – 2PhaseSP + *query rewrites*
- **NaiveCPP** – Naïve C++ implementation with unoptimized cut order
- **OptCPP** – Manually optimized C++ implementation

Results



Results interpretation

- For selective queries (most common)
  - Total improvement > 500 times
  - Dynamic *query optimization very effective*
  - Query rewrites marginal (20%) extra performance
  - Better than C++ performance possible?
    - 30% difference => with *algebra compiler* as fast as a manually optimized C++
    - Data copying can be avoided by *tighter integration* with ROOT
- For non-selective queries (not common)
  - Total improvement 30 times
  - Query optimization 77 % + rewrites 700 % => rewrite techniques most efficient
  - More challenging to approach C++ performance
    - Factor 4 away. Not impossible!

Future work (from the original presentation)

- Parallelization:
  - For massively parallel execution of scientific queries
  - SCSQ (SuperComputer Stream Query Processor)
  - SQISLE code partly integrated
- Algebra compiler, tighter integration with ROOT

Questions

- Why use SQL?
- Load time in [9] of 25000 events: 16.5 sec
  Overhead of StatQP/AttrQP: 29/26 sec
  makes sense for larger data…

More questions

- Assume statistics over the stream **stable**
- If the analysis is run for multiple time, better to load the data to database?
- How on earth the optimization happens? (The order of operators? Based on what strategy?)

Confusing terms

- Joined on *o* (p326)
- Transformation view

Other thoughts

- Why do the scientists come to database people?
- Using map reduce?
- Parallelize the computation?