DCTV: can we pose better questions to traces?

Daniel Colascione

October 25 2018

What is DCTV?

DCTV is an experimental framework for putting complex questions to trace files and getting back reliable answers.

- Trace database
  - Custom SQL dialect for heterogeneous time series
  - Column-oriented trace-backed query engine
- SQL REPL and batch query system
  - Explore traces with SQL-DCTV queries
  - Same queries useful for debugging, batch analysis
  - Multiple input, output formats
- GUI trace viewer
  - Flexible grouping, filtering, graphing, plotting
  - Asynchronously queries traces using DB backend
  - Interactively create data views; export them as SQL traces
  - Not discussed in this talk (no time)

Why is DCTV?

- Need powerful data exploration to find "unknown unknowns"
- Existing trace ecosystem fragmented, lacking power

Inspection	• Iterative evaluation of known scenario on live system • bpftrace, ftrace histograms, counters, perf • Powerful analysis, but analysis spec needed *before* running the scenario
Retrospection	• Backward-looking "forensic" analysis • Scenario frequency not repeatable • KernelShark, Trace Compass, LTTng viewer

- Data exploration just as important as metrics crunching
  - ... but should be easy to transform explorations intro metrics

Why is DCTV? Flexible Data Analysis

- What is analysis, anyway?
  - Fundamentally: iterated induction and abstraction-building
  - Least abstract: raw trace events
  - Most abstract: answer to question in your head
- The brain is just okay at abstraction-building
  - Example: vgrep raw `sched_switch` and glean useful stuff
  - Doesn't scale: working memory is small; we're bad at sums
- DCTV allows incrementally building complex abstractions
  - Ask questions in terms of answers to previously-asked questions
  - DCTV is largely written in itself!
  - Database core doesn't know about traces
  - Data model emphasizes composition

Why is DCTV? Error Avoidance

- I'm gravely concerned about silently incorrect analysis
  - We're inclined to believe and accept the plausible, even if it's ultimately wrong
  - DCTV is designed to reject nonsense operations
  - DCTV can't stop bad statistics, but it can reject mathematical nonsense
- Comprehensive dimensional analysis
  - Every quantity tagged with physical units
  - Queries fail on dimensional errors
  - Can define new units at runtime to avoid confusion
- Carefully-designed query language
  - Geared toward physically-meaningful "pit of success"
  - Metadata-level tracking verifies operation appropriateness
  - Trace-specific ergonomics
    - For example, automatic PID and TID disambiguation

Why is DCTV? High Performance on Big Data Sets

- DCTV is a GUI backend: query latency is important
- Designed for streaming processing on larger-than-core data sets
  - Trace files can get big: 50-60GB traces are common
  - Easy for off-the-shelf analysis tools to break down at this size
  - DCTV designed for larger-than-core data sets
- Sparseness, immutability of traces suggests columnar model
  - Opportunistic caching; modern techniques like "database cracking" obviate need for explicit indexes
  - Open research problem: simultaneous optimization of multiple queries
- Embedded use is a requirement
  - Runs on desktop, maybe inside browser one day
  - Precludes some big heavyweight DBMSes

DCTV Database Design

Basic Mechanics

- Trace events are rows in event-type-specific tables
- Work with a trace file by "mounting" it in the SQL namespace
  - Can mount multiple traces at once: run queries on multiple trace files
  - Multiple formats supported
    - ftrace text
    - systrace HTML
    - ftrace binary (planned)
    - perfetto protobuf (planned)
    - LTTng?

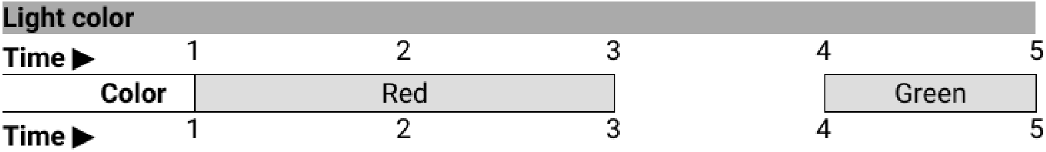
Example: basic trace mounting

```
~/dctv
$ ./dctv repl
Type .help for help.
DCTV> MOUNT TRACE 'local-data/dragonball.mini.trace' AS mytrace
DCTV> SELECT COUNT(next_comm) FROM mytrace.raw_events.sched_switch
COUNT(next_comm)
           32370
DCTV>
```

Data Model: SQL Superset

- SQL essence: set operations on high-dimensional points
- Query language supports "shape" as well as "point" operations
    - Data and *syntactic* support for intervals along the time axis
    - Each interval is called a "span"
    - Special table types: span tables and event tables
        - Optionally "partitioned" to capture higher dimensions
    - Special operators for working on span, event tables
        - `SPAN JOIN`
        - `SPAN GROUP`
        - `SPAN BROADCAST`
        - `EVENT BROADCAST`

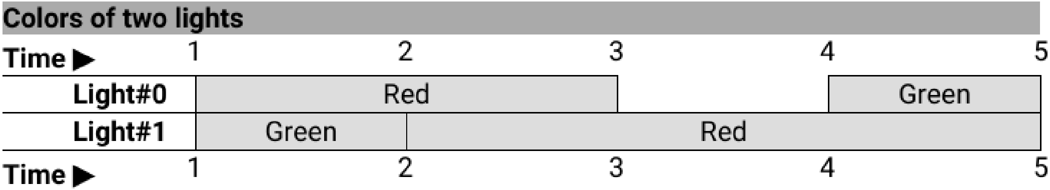
Example: span table



Light color (span table representation)

_ts	_duration	color
1	2	red
4	1	green

Example: partitioned span table



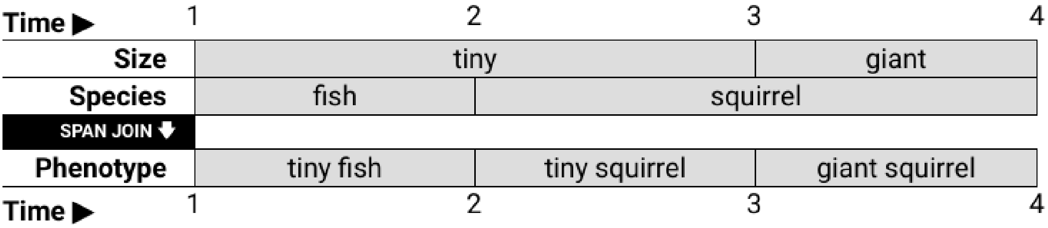
Colors of two lights (span table representation)

_ts	_duration	lightno	color
1	2	0	red
1	1	1	green
2	3	1	red
4	1	0	green

Span join

- `SPAN JOIN` aligns spans from different span tables
  - Output has a span boundary anywhere an input had a span boundary
- Comes in the usual SQL `INNER JOIN`, `OUTER JOIN`, etc. varieties
- Useful for asking questions of two time series with changes are different points.
- A regular, non-`SPAN JOIN` of a span table with a non-span table produces a span table partitioned by the join column of the unpartitioned table!

Span Join Diagram



Span Join Query: CPU times, frequency

```
DCTV> SELECT * FROM mytrace.scheduler.timeslices_p_cpu LIMIT 2;
_ts [ns] | _duration [ns] | cpu | comm       | pid   | prio
       0 |         164000 |   5 | swapper/5  |     0 |  120
   27000 |           7000 |   4 | trace-cmd  | 12202 |  120
DCTV> SELECT * FROM mytrace.scheduler.cpufreq_p_cpu LIMIT 2;
_ts [ns] | _duration [ns] | cpu | frequency [kcpu_cycle/s]
15035000 |         957000 |   0 |                  1171200
15051000 |         954000 |   1 |                  1171200
DCTV> SELECT _ts, _duration, frequency * _duration AS cycles FROM mytrace.scheduler
      cpufreq_p_cpu SPAN INNER JOIN mytrace.scheduler.timeslices_p_cpu LIMIT 2;
_ts [ns] | _duration [ns] | cycles [kcpu_cycle]
15035000 |         131000 |            153.4272
15051000 |         140000 |   163.96800000000002
DCTV>
```

Span Group

- `SPAN GROUP` is the "opposite" of `SPAN JOIN`
- Grouping puts spans together instead of breaking them apart
- Two kinds
  - Group by partition: collapse partitions, grouping payloads
    - Example: collapse per-CPU time series into system-wide time series
    - Example: collapse per-disk queue lengths into maximum queue length anywhere over time
  - Group by spans: force a span table into periods defined by another span table

Span Group Diagram

Time ▶	1	2	3	4	5	6	7	8	9
Number arms	2	5	0	7	2	4	9	0	
Periods	A		B		C		D		
Span group 🡇									
`MAX(arms)`	5		7		4		9		
`MIN(arms)`	2		0		2		0		
Time ▶	1	2	3	4	5	6	7	8	9

Complex example queries

Overall System Cpu Use Over Time

```
DCTV> WITH idle_by_span AS (
          SELECT SPAN SUM(_duration) AS total_time,
                      SUM(IF(pid=0, _duration, 0ns)) AS idle_time
                      FROM dctv.with_all_partitions(
                        mytrace.scheduler.timeslices_p_cpu) AS tscl
                      WHERE cpu=0
                      GROUP SPANS USING PARTITIONS)
      SELECT SPAN SUM(idle_time) / SUM(total_time)
      FROM idle_by_span
      GROUP AND INTERSECT SPANS INTO SPANS mytrace.quantize(1000us)
      LIMIT 5
 _ts [ns] | _duration [ns] | (SUM(idle_time) / SUM(total_time))
11000000  |        1000000 |                  0.945140131186642 9
12000000  |        1000000 |                                1.0
13000000  |        1000000 |                 0.9422283356258597
14000000  |        1000000 |                 0.8962510897994769
15000000  |        1000000 |                 0.7383268482490273
```

Frame Deadline Miss Attribution

```
DCTV> WITH frames AS (SELECT SPAN * FROM dctv.time_series_to_spans(
                      sources=>[{source=>(
                           SELECT * FROM trace.raw_events.`print|B`
                           WHERE name='eglBeginFrame'),
                      }],
                      columns=>[])),
           bad_frames AS (SELECT SPAN * FROM frames WHERE _duration > 17ms),
           bad_timeslices AS (SELECT SPAN * FROM bad_frames
                               SPAN BROADCAST INTO SPAN PARTITIONS
                               trace.scheduler.timeslices_p_cpu)
      SELECT comm, cpu, SUM(_duration) AS totdur FROM bad_timeslices
      WHERE pid != 0
      GROUP BY comm, cpu
      ORDER BY totdur DESC
      LIMIT 20
```

Development Status

- 40kLOC of Python, C++ and growing
- Includes object-based variant of Python multiprocessing that allows for concurrent query execution
- Interface still experimental
  - No guarantees about stability of the query language or other APIs
  - Patches welcome
- Perfetto includes a SQL-based "trace processor" as well
  - Perfetto's is SQLite-based: less powerful, less experimental
  - DCTV more focused on experimentation
  - Some ideas from DCTV imported into Perfetto
    - Columnar storage for trace data
    - Subset of span join operations

Future directions

- Incorporate perf events, system log events
- SQL/PL imperative query support
- Expanded "standard library" of trace building blocks
- Automatic slurping of ftrace event stacks
- Apache Arrow representation of result rows
  - Easier integration into SciPy ecosystem
- IPython notebook integration
- Collaborative annotation
- "Time table" spelling of span operations

Thank you for listening!
Questions welcome.