LLVM Query Runtime

VALKyrie

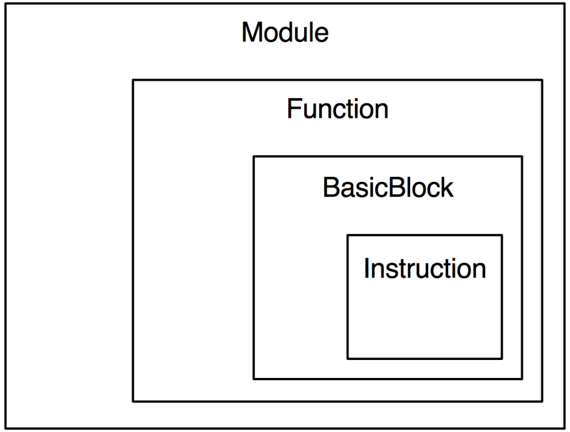
Arindam
Kaushik

Ladan
Vinayak

Progress

1. Components in LLVM IR
2. Program to Generate IR
    a. (Internals of IR Builder)
3. Design decisions

Components in LLVM IR



1. 'Value' class
   a. Function
   b. Basic Block
   c. Instruction

```
%0 = add i32 %1, 5
```

Components in LLVM IR

1. Instruction
   a. C++ Instruction class
   b. Members
      i. opcode
      ii. type
      iii. list of operands (pointers)
   c. Static Single Assignment (SSA)
      i. Reaching definition analysis

2. Inspecting IR
   a. dump()

SSA

$$\begin{array}{l} y := 1 \\ y := 2 \\ x := y \end{array} \Rightarrow \begin{array}{l} y_1 := 1 \\ y_2 := 2 \\ x_1 := y_2 \end{array}$$

```
errs() << "Function body:\n"
F.dump();

for (auto& B : F) {
  errs() << "Basic block:\n";
  B.dump();

  for (auto& I : B) {
    errs() << "Instruction: ";
    I.dump();
  }
}
```

Program to Generate IR

1. Lexer
2. Parser
3. Abstract Syntax Tree (Parse Tree) - Expression AST Base
    a. Number : Expression
    b. Variable : Expression
    c. Binary Operator : Expression
    d. Function Calls : Expression
    e. Function Definition : Prototype
4. *codegen()
5. ‘Builder’ object of ‘IRBuilder’ class

Design Decisions

1. Physical data layout
2. Abstractions over IRBuilder
   a. Conditionals
   b. Mutable variables
   c. Looping constructs
3. Granularity of control between the Java plan generator and the C++/LLVM execution strategy
   a. Export as JSON/XML tree of operators?
   b. Or, a minimal pseudo-language?

Next Step

Create a simple file scan operator in LLVM

Challenges

```
void RTFM() {
    RTFM();
}
```

Lightweight Runtimes

Team Sparkle

Dhinesh
Shiva
Keno
Guru

Next Steps

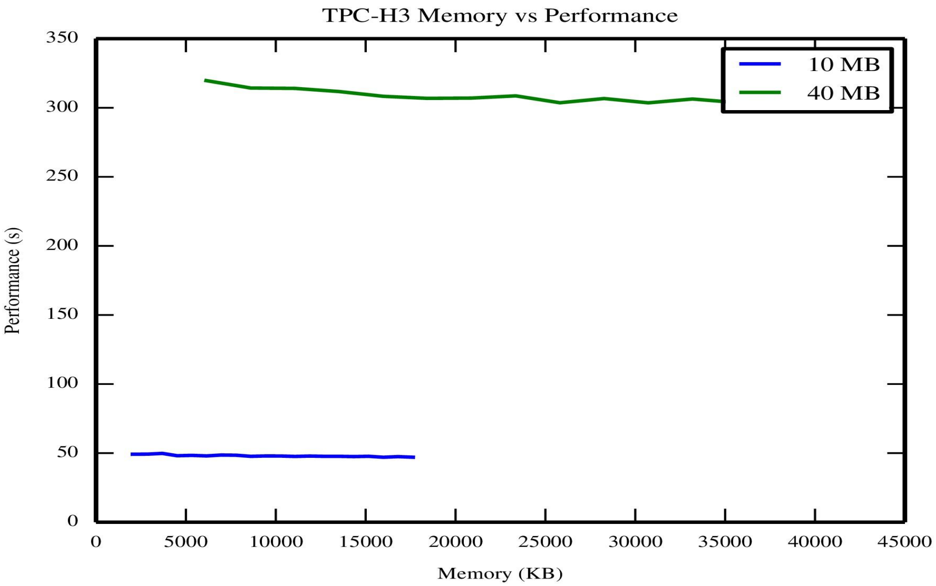
- Study behaviour under memory pressure
- Java and GC effects on Galileo <- Still stuck here
- Characterize specific workloads we want to support
    - rapid inserts
    - rapid queries
    - range queries
- Find bottlenecks in current implementations
- Benchmark streaming data eg. light sensor from phone

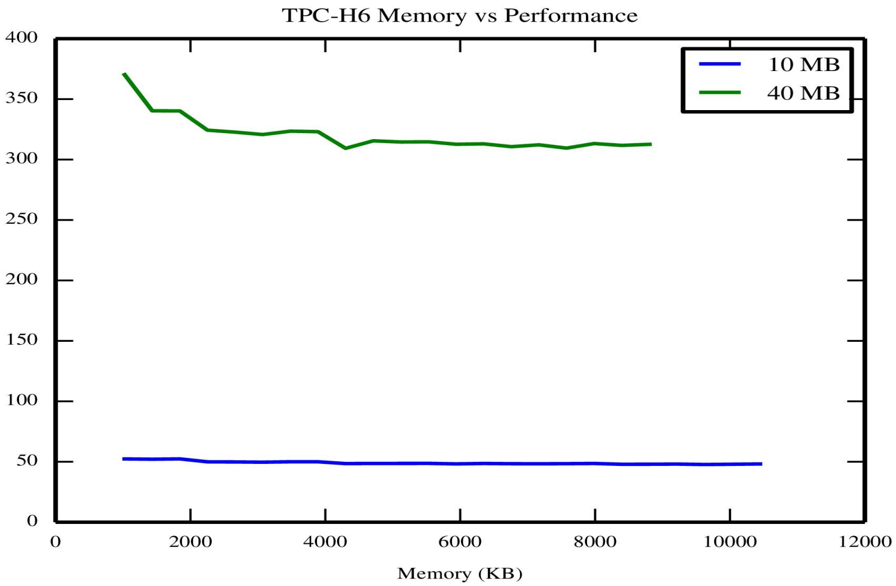
Java and GC Effects

- Currently processing 70MB dataset
  - Elapsed time > 72 hours
- Guru: Looking into GC Trigger

Java and GC Effects

- Currently processing 70MB dataset
  - Elapsed time > 72 hours
- Need more data points!





Java and GC Effects

- Currently processing 70MB dataset
    - Elapsed time > 72 hours
- Guru: Looking into GC Trigger

GC Causes

`hotspot/src/share/vm/gc_interface/gccause.cpp`

- java_lang_system_gc
- full_gc_alot
- scavenge_alot
- allocation_profiler
- jvmti_force_gc
- gc_locker
- heap_inspection
- heap_dump
- no_gc
- allocation_failure
- tenured_generation_full
- metadata_GC_threshold
- cms_generation_full
- cms_initial_mark
- cms_final_remark
- cms_concurrent_mark
- old_generation_expanded_on_last_scavenge
- old_generation_too_full_to_scavenge
- adaptive_size_policy
- g1_inc_collection_pause
- g1_humongous_allocation
- last_ditch_collection
- last_gc_cause

Java and GC Effects

- Currently processing 40MB dataset
    - Elapsed time > 72 hours
- Guru: Looking into GC Trigger
- Obtained openjdk-8
- Compiled openjdk-8
- Yet to identify GC trigger threshold

Characterization of Workload - 1

Given n streams at frequencies <f1, f2, f3…. fn> to the galileo, and a windowed query containing joins over the streams, what percentage of the expected results can our query engine produce? A simplistic example:

stream_1 = screen brightness readings of the form <user, time, value> from mobile phones

stream_2 = heartbeats of the form <user, time> from PCs

Query: What times did user a use both her phone and PC at the same time in the past day? How about with stream frequencies, window size, and number of stream sources

Characterization of Workload - 2

The challenges involved:

- Selecting the optimal join algorithm
- Handling windows that won’t fit into memory (we have < 256MB)
- Possibly taking advantage of varying frequencies for each stream
- Handling increasing frequencies

Next Steps

- Getting a full-fledged dataset (from our service)
  - Currently very slow. Acquired 280KB
  - Increased data collection speed to 1 reading/s (from 1 reading/5s)
- Raise GC Threshold and re-run experiments
- Setting up a simple benchmark based on the described workload
- Figuring out storage and indexing strategies
- Cost Estimation for different join strategies

PocketData Benchmark

Naveen, Sankar, Saravanan, Sathish

Resolved Challenges from Last Week

Parallelization of Parsing and Extraction of Log Files – User based.

Identification of the application names for 32 million queries – removed *thread_id* as the mapping criteria.

Added PRAGMA support to latest JSQLParser provided – still unable to parse 2266 queries of the form PRAGMA <name>('<value>') & PRAGMA <name> = '<value>'.

Extrated few more features of SQL - # of joins (outer, left ...) , # of union...

Challenges Faced/Facing…

JSQLParser still doesn't parse 4,207,615 queries:

❖key (column name): 1,973,090

SELECT key, value FROM CalendarCache WHERE key=?

❖(()) double parenthesis: 339,575

SELECT _id, contact_last_updated_timestamp FROM view_contacts WHERE ((_id IN default_directory))

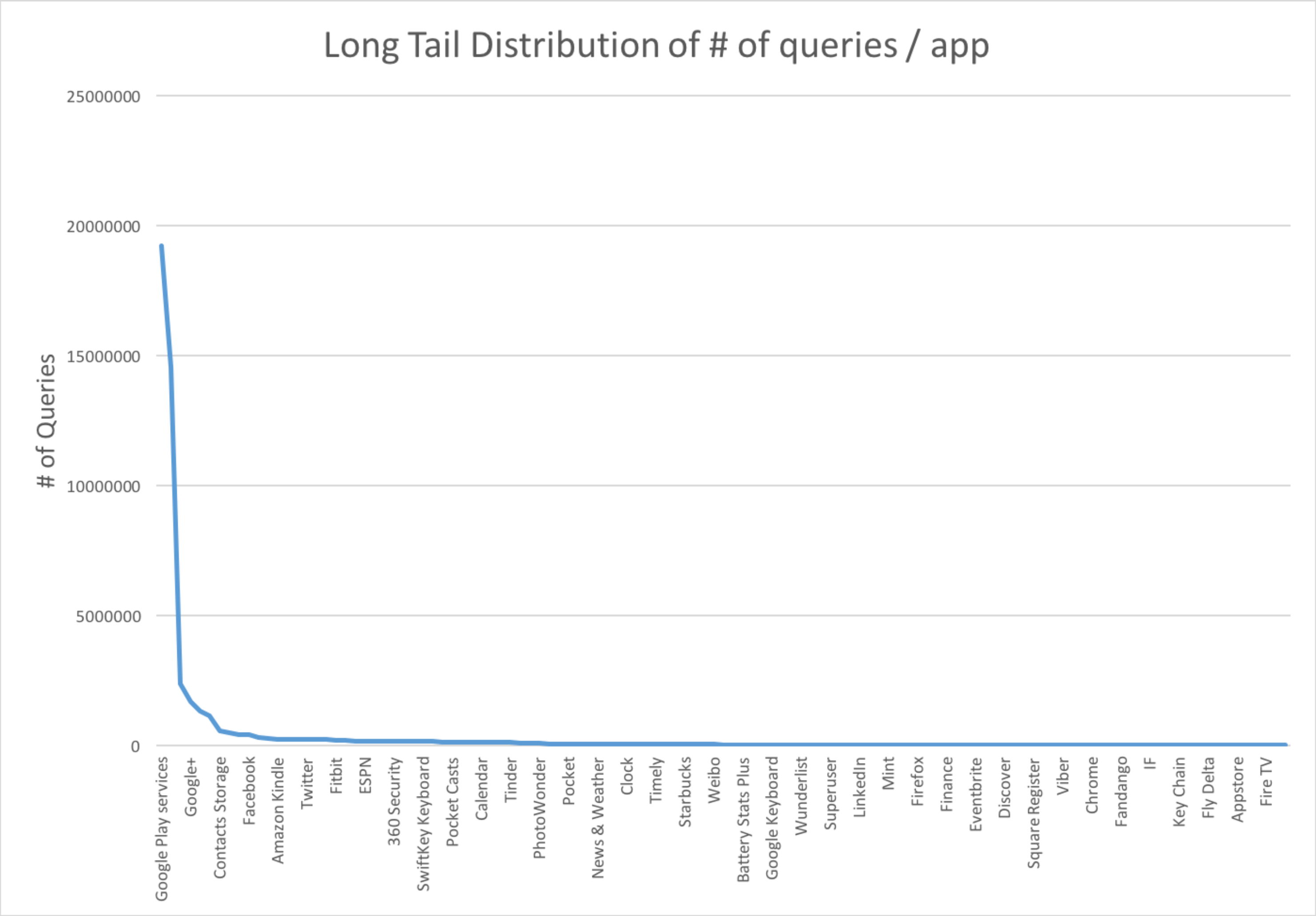
❖PRAGMA: 2,266

PRAGMA table_info('nfcTapEvent')

PRAGMA secure_delete = ON;

More to be analyzed…

Long Tail distribution - # of queries / app



Maximum # of queries in 10 ms range

user_id	no_of_queries	avg_response_time (µs)
4	80	57673.125
7	69	56148
10	66	82759.5909
2	60	53961.0167
5	55	54337.0727
9	50	121999
8	47	145619.4681
6	39	98032.8718
11	38	89813.5263
3	37	124932.4324
1	21	87041.381

Roadmap

- ❖ Parse most of the SQLs.
- ❖ Extract more features.
- ❖ Prepare Report for check-point.
- ❖ Start building the model.

Embedded Database Benchmark

Team CodeBlooded

What makes a good embedded database?

- Small footprint
  - Installation size
- Less memory consumption
  - Heap size variations
- Self managed
  - No DBA involvement, logging, recovery
- Portability
  - Single file databases

What makes a good embedded database?

- May or may not require persistence
  - Only in-memory mode
- Support for mobile devices
  - Power consumption

Embedded Applications

- Key-Value stores
  - mobile apps, browser cookies/bookmarks
  - multiple inserts and reads
- Internet of Things
  - sensors, cameras, id scanners
  - heavy inserts, aggregate queries, joins
- Read-only persistence
  - programmed devices, cache
  - heavy reads

Embedded Applications

- Version/Source Control
  - Fossil
  - mixed load, join queries
- Desktop media applications
  - iTunes, photos
  - moderate inserts, heavy reads

Next Steps

- YCSB code for generating workloads
- Poleposition benchmark test suite to compare database engines and object-relational mapping technology
- Comparing available results to find meaningful data
- Using information from these benchmarks to come up with initial workloads