Embedded Databases MicroBenchmark

Team CodeBlooded

Refinement of goals

- Try and understand the usefulness of a DB in terms of a particular application domain
- Goal : Build a DB calculator that works with the micro-benchmark
  - Embedded devices and the internet of things
  - Version control
  - Websites
  - Data analysis
  - Internal or temporary databases (Joins etc)

Understanding the Databases

- Each claims to be the fastest
- Each claims to have the smallest footprint
- Each claim to be cross platform
- No one compares themselves to every other databse
- How do they differ in terms of features?

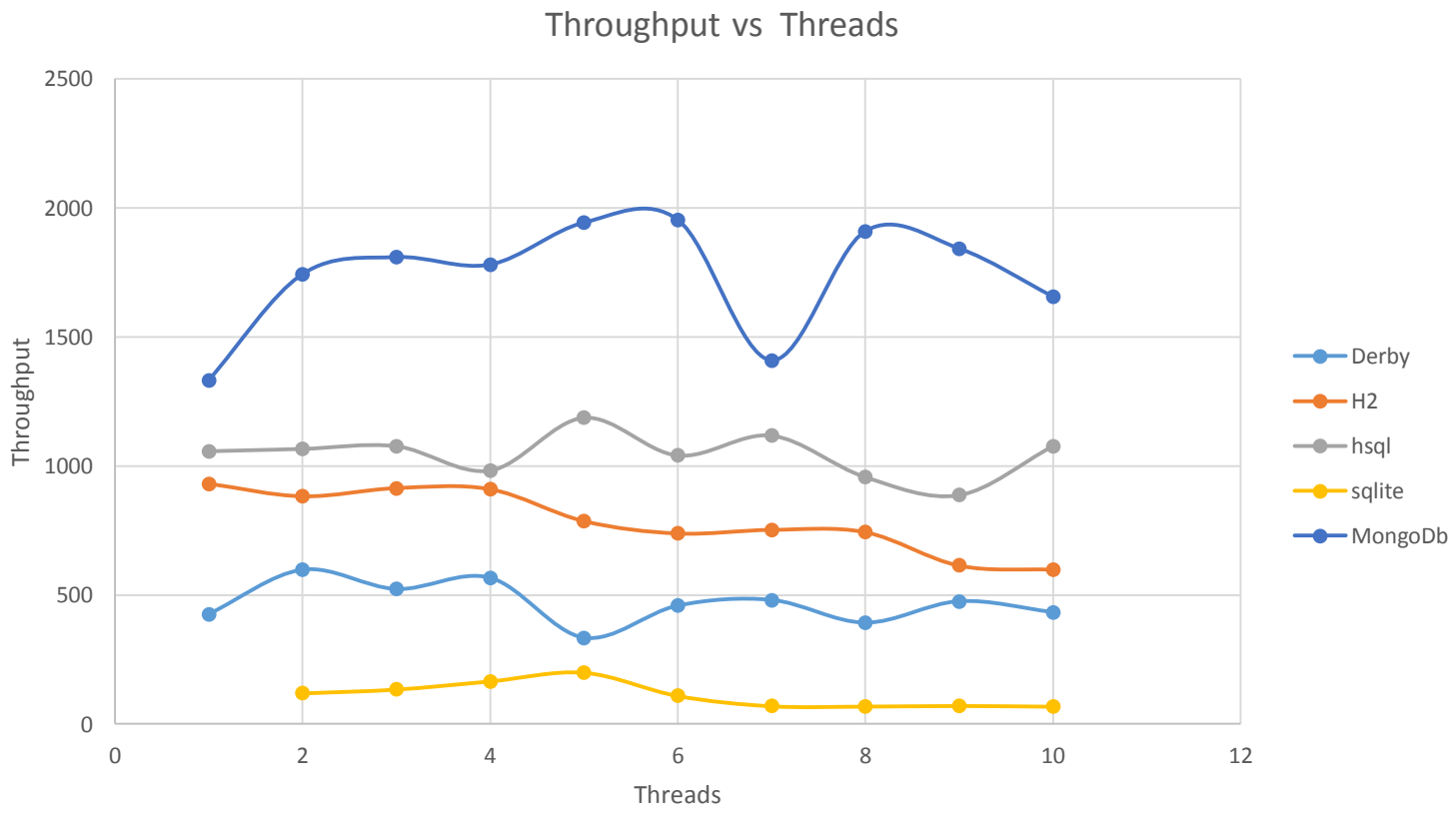
Feature comparison

- -SqlLite and H2 are the only ones that support any other indexing other than B-/B+
- Derby, Berkeley and HSQLDB do not
- -HSQLDB does support merge joins others do not (Derby claims it does)
- -SQLLite only Left outer join
- -Derby supports cursors others do not

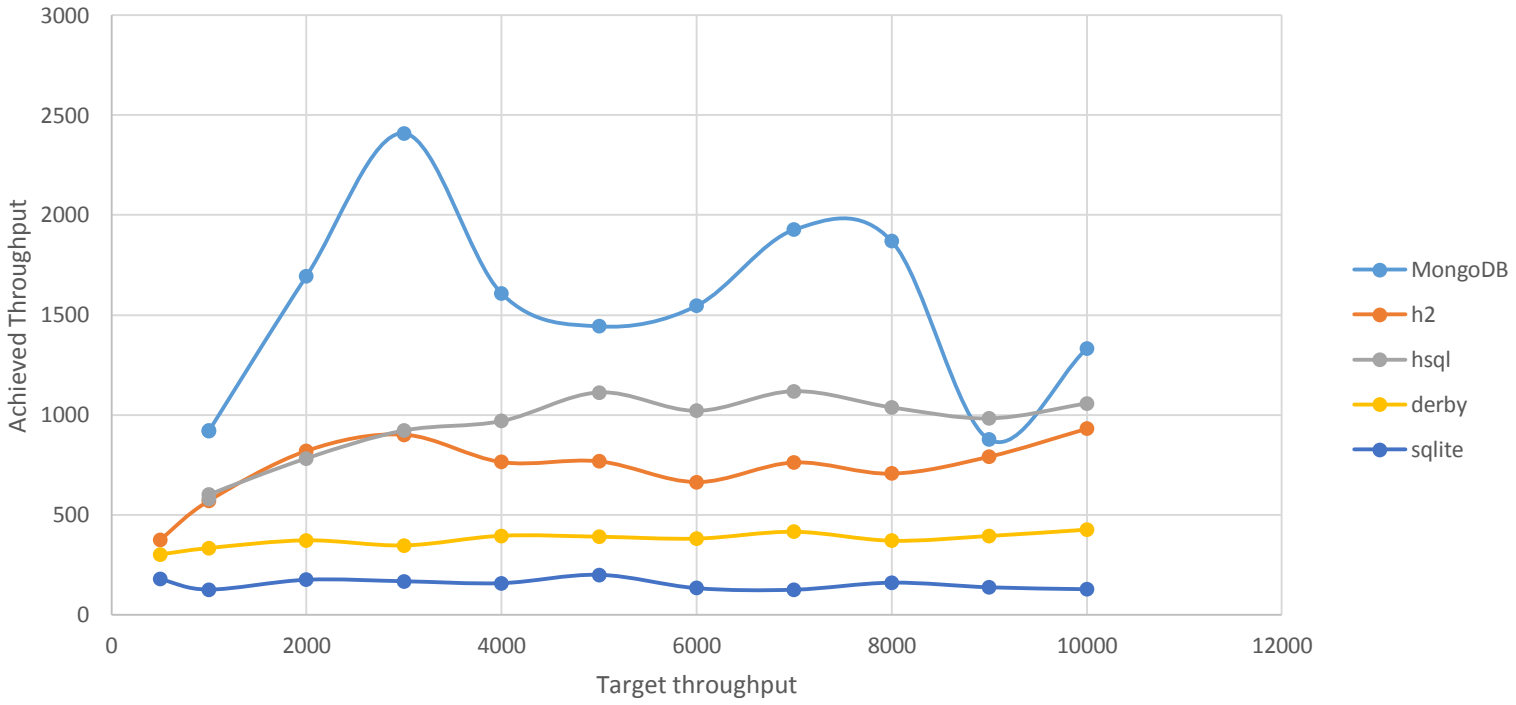
Future tests (Things to include in our Benchmark)

- Complex queries : Inner and outer joins, subqueries, read only views and inline views (Performance)
- ORDER BY, GROUP BY, HAVING, UNION, LIMIT, TOP
- Concurrency Test
- Uses a small number of database files (Comparison)
- Idle time resource use/usefulness
- Performance when a database is NOT at capacity (Metrics : Time, CPU use)

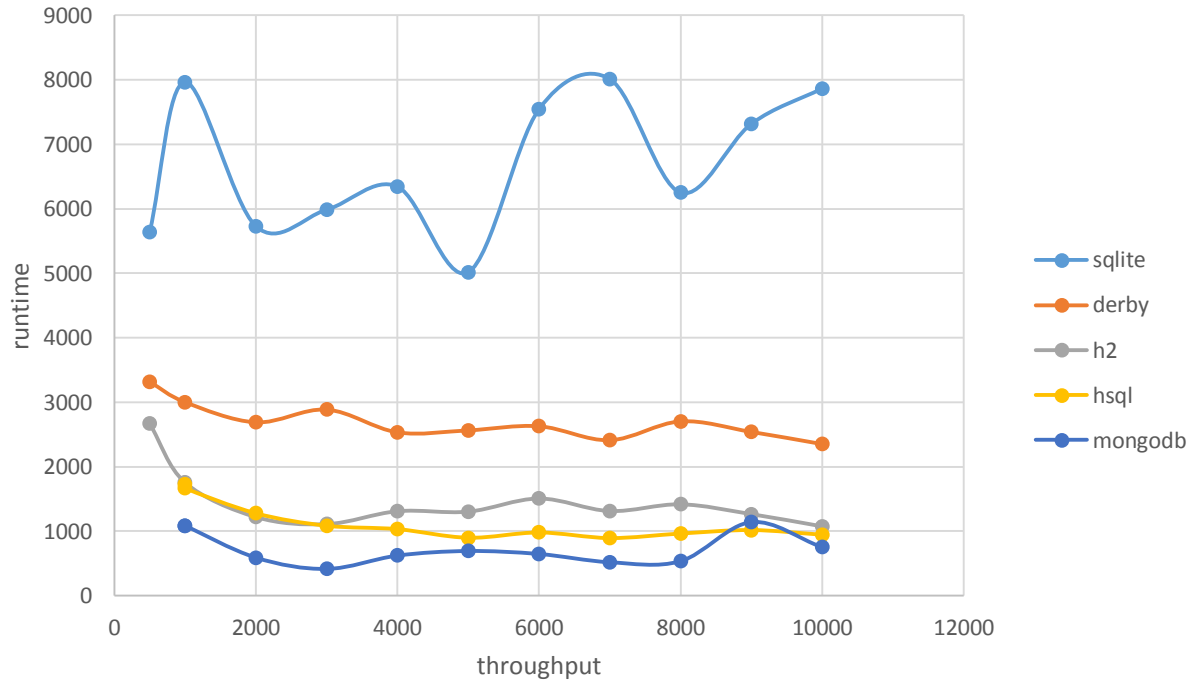
Throughput vs threads



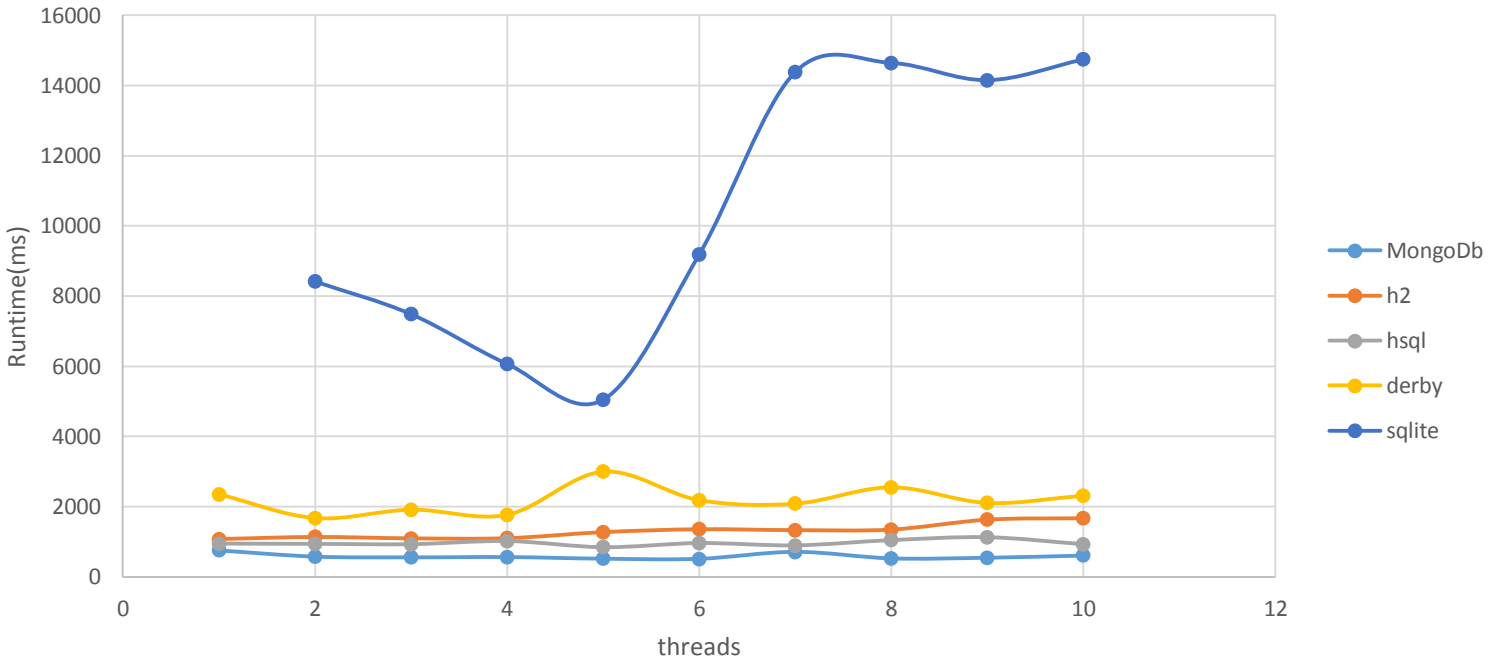
Achieved vs target throughput



Overall Runtime vs throughput



Overall Runtime vs threads



Analysis

SqlLite closes and opens database files and invalidates cache for each transaction.
Running multiple sql in single transaction is faster.
Fsync is called after every transaction to put changes on disk, where H2 batches the changes.
SqlLite waits idle for disk I/O to complete.

Lightweight Runtimes

Team Sparkle

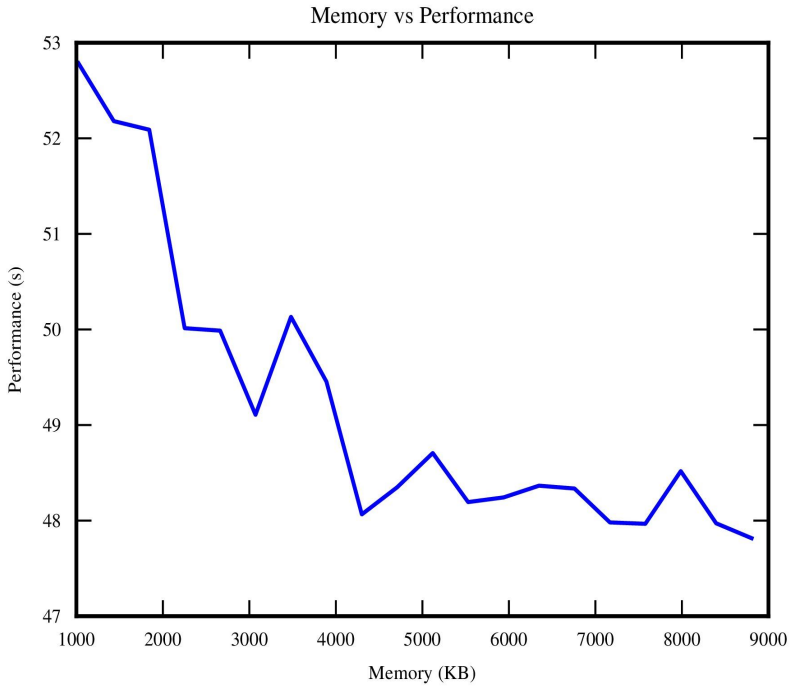
Dhinesh
Shiva
Keno
Guru

Next Steps

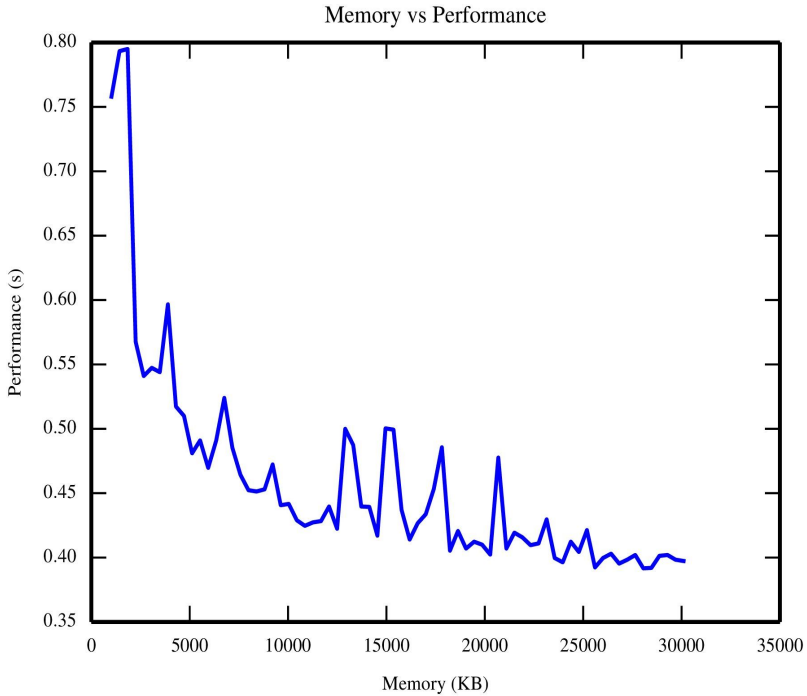
- Study behaviour under memory pressure
- Java and GC effects on Galileo
- Characterize specific workloads we want to support
  - rapid inserts
  - rapid queries
  - range queries
- Find bottlenecks in current implementations
- Benchmark streaming data eg. light sensor from phone

32-bit JVM Comparison

Galileo



Host



TPCH-1 on a 10MB dataset

32-bit JVM Comparison

With Increasing Dataset Sizes

- Working on it
- Have plots from host
- Been running since Wednesday
- Galileo - 1hr/query (`--repeat 10`)

Simulating Intended Workload

- Set up server to collect data
- Writing app to stream light sensor data to Galileo
- Plans to add 1-2 more sensors for rich time-range queries

Interpretation of the hprof info

For the TPCH-3 query on a 30MB dataset, we found the top entries to be:

- 5.39% 5.39% 3404 300603 sun.util.calendar.BaseCalendar.getCalendarDateFromFixedDate
- 2.82% 8.21% 1780 300460 java.lang.Character.digit
- 2.79% 11.00% 1761 300383 java.lang.AbstractStringBuilder.append
- 2.43% 13.44% 1536 300593 java.lang.Character.digit
- 2.41% 15.85% 1521 300438 java.util.HashMap.hash
- 2.21% 18.06% 1393 300475 java.lang.String.<init>
- 2.19% 20.25% 1385 300477 java.util.Arrays.copyOf
- 2.19% 22.44% 1379 300606 sun.util.calendar.BaseCalendar.getFixedDate
- 2.17% 24.60% 1367 300479 edu.buffalo.cse562.DTO.Datum.<init>
- 2.17% 26.77% 1367 300627 sun.util.calendar.Gregorian$Date.<init>
- 1.89% 28.66% 1192 300667 sun.util.calendar.BaseCalendar.normalizeMonth
- 1.78% 30.44% 1123 300480 java.io.BufferedReader.readLine
- 1.75% 32.19% 1104 300490 net.sf.jsqlparser.schema.Column.getWholeColumnName
- 1.69% 33.88% 1068 300451 sun.nio.cs.US_ASCII$Decoder.decodeArrayLoop
- 1.68% 35.56% 1062 300441 java.lang.String.<init>

o can be traced directly to our code o external activities

Next Steps

- Getting a full-fledged dataset (from our service)
- Characterizing insertion patterns
- Managing indexes (if any)
- Storage - Memory and Disk
- Characterizing query patterns

LLVM Query Runtime

VALKyrie

Arindam
Kaushik

Ladan
Vinayak

Progress

- Pros and Cons of different approaches of generating IR
- More on benchmarking (if we have time)

Ways to generate IR

- LLVM-J
  - Pros:
    - It is in Java
  - Cons:
    - We couldn’t get it running
    - Lack of documentation
    - Based on older version of LLVM
    - Abandoned about a year ago

Ways to generate IR

- Java IR builder
  - Pros:
    - It is in Java
    - Simple to use API
    - We could generate IR
  - Cons:
    - We could not run the generated IR
    - Based on older version of LLVM (3 years ago)
    - No documentation at all.

Ways to generate IR

- Python bindings - llvmpy / llvmlite
  - Pros:
    - Pythonic!!!
    - llvmlite pretty simple
  - Cons:
    - llvmpy too bloated, bad performance and abandoned
    - llvmlite unstable API, development driven by Numba

Ways to generate IR

- C++ IR builder
  - Pros:
    - Enough documentation for start
    - The most complete approach
    - Based on the latest version of LLVM
    - We could generate IR and run it
  - Cons:
    - It is C++!!!

Our Challenges this week

- Integrating our CSE562 code with C++
- Integrating C++ and LLVM generator code.
  - So we can read the tuples by C++ and bring them to memory
  - Use LLVM just for optimization

Design Decisions

- Parsing and optimizing using existing Java Implementation
- Dump the results in an easy to parse format (e.g. Json)
- Use C++ IR-Builder for generating IR

Benchmark discussion

PocketData Benchmark

Naveen, Sankar, Saravanan, Sathish

What did we do last week?

- Started extracting data from phone log.

- 93b46e40acbf1167ab0ad421b73761f16b74b562 1426486358667 1426486358667.0 2015-03-16 06:12:38.667574 13274 13274 I SQLite-Query-PhoneLab
  {"Counter":0,"LogFormat":"1.0",**"AppName":"Google Play Books"**,"Action":"**APP_NAME**","PackageName":"com.google.android.apps.books"}

- 93b46e40acbf1167ab0ad421b73761f16b74b562 1426477052477 1426477052477.1 2015-03-16 03:37:32.477583 13274 13274 I SQLite-Query-PhoneLab
  {"Counter":561,"LogFormat":"1.0","Time":243698,"Arguments":"null","Results":"SELECT value FROM ScheduledTaskProto ORDER BY sortingValue ASC, insertionOrder ASC","Action":"SELECT","Rows returned":3}

- {"Counter":27,"LogFormat":"1.0","Time":129375,"Results":"SQLiteProgram: INSERT INTO carriers(bearer,authtype,carrier_enabled,protocol,mmsproxy,roaming_protocol,numeric,mcc,type,mmsc,password,mvno_match_data,mvno_type,name,server,mnc,apn,user,mmsport) VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)","Action":"INSERT",**"Arguments(hashCoded)"**:"0,-1,1231,2343,-1630285132,2343,47743056,49679,-697368471,155249537,117478,0,0,1755004508,42,1537,1954370557,117478,1784,"}

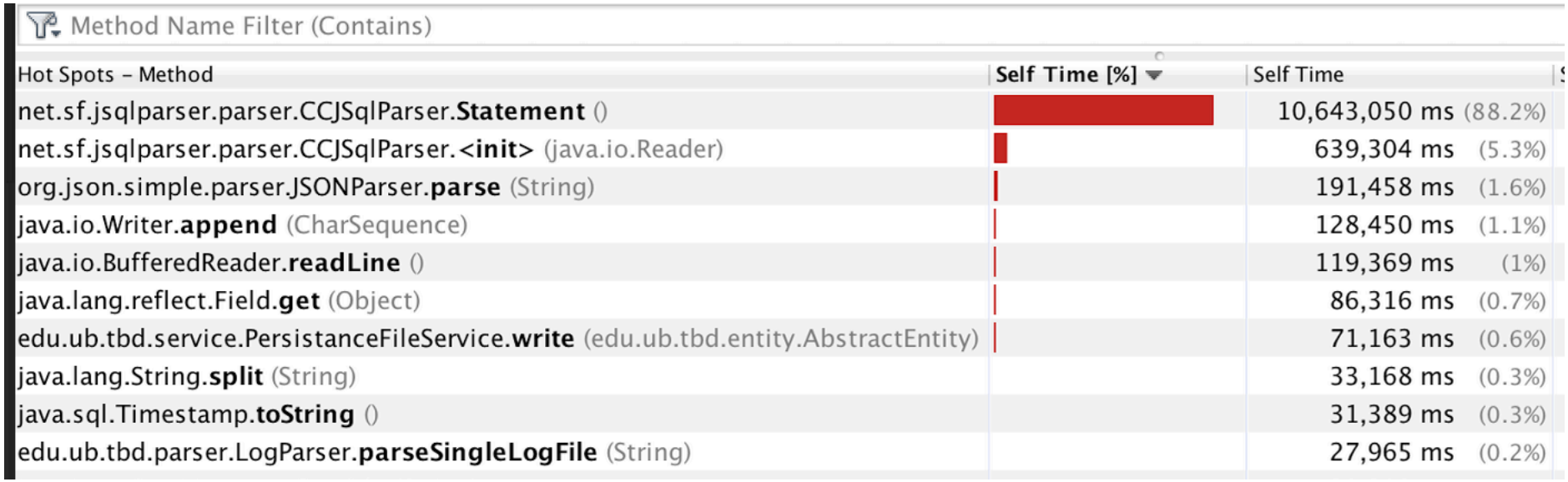
- {"Counter":5,"LogFormat":"1.0","Time":149843,"Arguments":"null","Results":"**PRAGMA** table_info(name)","Action":"SELECT","Rows returned":0}

How did we parse it?

- Switched to Java from Python
  - Java is faster. Need to parse more than a GB
  - Better support with jsqlparser
- Extracted features like number of projected columns, table count etc.,
  - Made use of jsqlparser visitor pattern
- Modified jsqlparser to allow PRAGMA and INSERT or REPLACE queries (for future).

Challenges faced

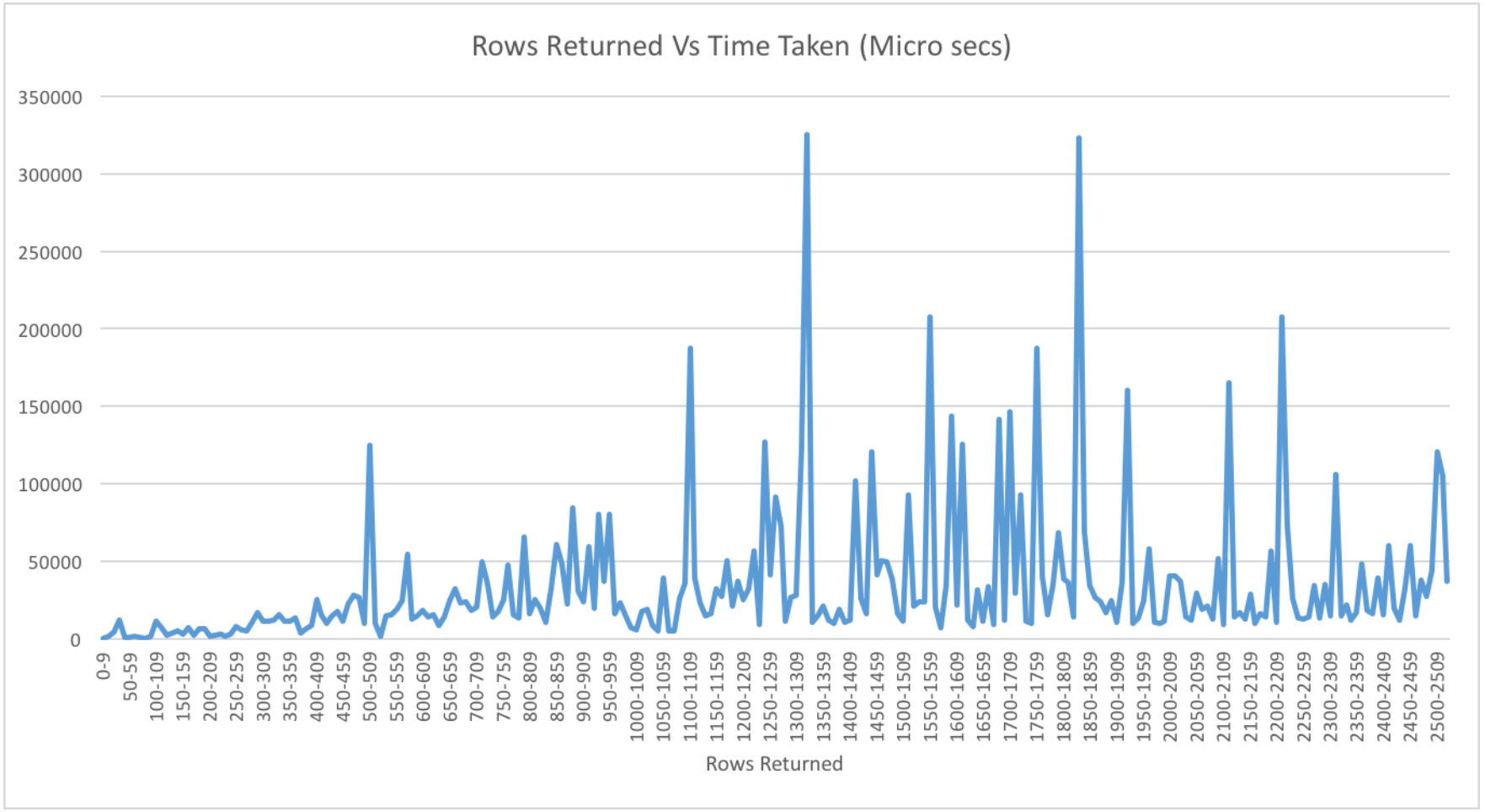
- It took 3.5 hours to parse and extract log files.
- Future : To run parallel for users.



Challenges [contd…]

- Unable to find app names for 32 million queries.
  - Something wrong with our logic
- Coming up with features without data available.

Results


Rows Returned Vs Time Taken (Micro secs)
350000
300000
250000
200000
150000
100000
50000
0
0-9
50-59
100-109
150-159
200-209
250-259
300-309
350-359
400-409
450-459
500-509
550-559
600-609
650-659
700-709
750-759
800-809
850-859
900-909
950-959
1000-1009
1050-1059
1100-1109
1150-1159
1200-1209
1250-1259
1300-1309
1350-1359
1400-1409
1450-1459
1500-1509
1550-1559
1600-1609
1650-1659
1700-1709
1750-1759
1800-1809
1850-1859
1900-1909
1950-1959
2000-2009
2050-2059
2100-2109
2150-2159
2200-2209
2250-2259
2300-2309
2350-2359
2400-2409
2450-2459
2500-2509
Rows Returned

Next week plan

- Play around with the extracted data to come up with interesting features.
- Extract more features from the log files.