Introduction to performance testing in Apache Beam

Alexey Romanenko
Principal Software Engineer, Talend
Apache Beam PMC Member
Intro
Performance testing vs Benchmarking

**Performance testing** is in general a testing practice performed to determine how a system performs in terms of responsiveness and stability under a particular workload.

It can also serve to investigate, measure, validate or verify other quality attributes of the system, such as scalability, reliability and resource usage.

**Benchmark** is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

https://en.wikipedia.org/wiki/Benchmark_(computing)
Why do we need performance testing in Beam?

- Measure a runner **performance** and detect performance degradation (if any)
  - e.g. between two Beam releases or periodically
- Test how Beam pipelines run **under the load**
- Compare the performance for **different runners** and **SDK** in Beam
  - Same test suite, same datasets, same environment (well, we do our best...)
- Compare the performance between Beam **runners** and **native engines**
  - Sensible topic =)
Performance testing in Beam

- IO transform integration tests
- Core Beam Operations tests
- Nexmark suites
- TPC-DS suites
IOIT
IOIT (IO Integration Tests)

- “2-in-1”: integration and performance tests (depending on input data size)
- Intended to be implemented for every IO connector
  - Some IOs are still missing
- Only batch mode
  - `BoundedSource` has to be used for streaming pipelines
- For now, implemented only for Java SDK
- Supported runners:
  - Any runner that supports Java SDK
- Run manually / on Jenkins
- Grafana dashboard integration
IOIT: Common scenario

Write pipeline

N records \(\rightarrow\) IO.write() \(\rightarrow\) IO Sink \(\rightarrow\) Collect Write metrics

Read pipeline

IO Source \(\rightarrow\) IO.read() \(\rightarrow\) Count records \(\rightarrow\) hash(input) \(\rightarrow\) Collect Read metrics
IOIT (IO Integration Tests)

Collected metrics:

- Read time
- Write time
IOIT: Pros/Cons

Pros:

- Leverage the **same code** as for ITs
  - Most Java IOs already has them
- **Easy** to implement for new IO
- Runs against **real** (or **k8s**) data backends

Cons:

- Only **Java** SDK and Batch mode
- Very **few** metrics
- **Limited** number of predefined input records (N)
Core Beam Operations
Core Beam Operations Load Tests

- Test performance of the core beam operations from Apache Beam model on different runners:
  - ParDo, ParDo with SideInput, GroupByKey, CoGroupByKey, Combine
- Uses *Synthetic Source* and *Synthetic Step*
- Supports **Batch** and **Streaming**
- **SDK** supported:
  - Java SDK, Python SDK and Go SDK
- **Runners** supported:
  - Dataflow, Flink, Spark (Dataset)
- Runs on **Jenkins**
- **Grafana** dashboard integration
**Synthetic Source & Step**

*Synthetic Source* is a highly parameterizable *Source* that provides deterministic data *(KV<byte[], byte[]>)*.

**Provided options:**
- Seed
- Key and value size
- Hot keys
- Delay between consequent data emissions
- Number of generated records
- ... and others

*Synthetic Step* is a highly parameterizable *DoFn* that consumes *(KV<byte[], byte[]>)* and emits *(KV<byte[], byte[]>)*.

**Provided options:**
- Actions between data emissions
- Delay per bundle
- Upper throughput limit
- ... and others

+ iterations
+ fanout
Core Beam Operations Load Tests

Gathered metrics:

- Run time
- Consumed bytes
- Memory usage
- Split/bundle count
- Throughput / lag (for streaming scenarios)
Example: ParDoLoadTest

```java
PCollection<KV<byte[], byte[]>> input =
    pipeline
    .apply("Read input", readFromSource(sourceOptions)) // Synthetic Source
    .apply(ParDo.of(runtimeMonitor))
    .apply(ParDo.of(new ByteMonitor(METRICS_NAMESPACE, "totalBytes.count")));

for (int i = 0; i < options.getIterations(); i++) {
    input =
    input.apply(
        String.format("Step: %d", i),
        ParDo.of(
            new CounterOperation<*>(
                options.getNumberOfCounters(), options.getNumberOfCounterOperations())));
}

input.apply(ParDo.of(runtimeMonitor));
```
Example: ParDoLoadTest

<table>
<thead>
<tr>
<th>Data processing type</th>
<th>java</th>
<th>ParDo</th>
<th>2GB, 100 byte records, 10 iterations</th>
</tr>
</thead>
</table>

- dataflow_runtime_sec: 18.6 s
- dataflow_v2_java11_runtime_sec: 17.8 s
- dataflow_v2_java17_runtime_sec: 13.2 s
- sparkstructuredstreaming_runtime_sec: 1.11 min
Nexmark
Nexmark benchmark suite

Nexmark is a suite of pipelines inspired by the ‘continuous data stream’ queries in Nexmark research paper.

These are multiple queries over a three entities model representing an online auction system:

- **Person** represents a person submitting an item for auction and/or making a bid on an auction.
- **Auction** represents an item under auction.
- **Bid** represents a bid for an item under auction.

Example:
Query 4: What is the average selling price for each auction category?
Nexmark

9 (+6) benchmark queries of a continuous processing system

- Continuous queries is a good match for the Beam Model
- Run regularly for a long time on Beam and helped find MANY issues + regressions

**but**

- Not running at big scale
- Not Industry standard
- We can’t compare results with other systems (only inside Beam)
Nexmark in Beam

- Supports **batch** and **streaming** pipelines
- Implemented only for **Java SDK**
  - non-SQL
  - SQL
- Running on:
  - Dataflow runner
  - Spark (RDD and Dataset) runner
  - Flink runner
- Used to detect **performance regression** for Beam releases
Nexmark: Default configuration

Events generation
- 100,000 events generated
- 100 generator threads
- Event rate in SIN curve
- Initial event rate of 10,000
- Event rate step of 10,000
- 100 concurrent auctions
- 1,000 concurrent persons bidding / creating auctions

Windows
- size 10s
- sliding period 5s
- watermark hold for 0s

Events Proportions
- Hot Auctions = ½
- Hot Bidders = ¼
- Hot Sellers = ¼

Technical
- Artificial CPU load
- Artificial IO load
# Nexmark: Output

## Performance:

<table>
<thead>
<tr>
<th>Conf</th>
<th>Runtime (sec)</th>
<th>Events (/sec)</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>5.5</td>
<td>18138.9</td>
<td>100000</td>
</tr>
<tr>
<td>0001</td>
<td>4.2</td>
<td>23657.4</td>
<td>92000</td>
</tr>
<tr>
<td>0002</td>
<td>2.2</td>
<td>45683.0</td>
<td>351</td>
</tr>
<tr>
<td>0003</td>
<td>3.9</td>
<td>25348.5</td>
<td>444</td>
</tr>
<tr>
<td>0004</td>
<td>1.6</td>
<td>6207.3</td>
<td>40</td>
</tr>
<tr>
<td>0005</td>
<td>5.0</td>
<td>20173.5</td>
<td>12</td>
</tr>
<tr>
<td>0006</td>
<td>0.9</td>
<td>11376.6</td>
<td>401</td>
</tr>
<tr>
<td>0007</td>
<td>121.4</td>
<td>823.5</td>
<td>1</td>
</tr>
<tr>
<td>0008</td>
<td>2.5</td>
<td>40273.9</td>
<td>6000</td>
</tr>
<tr>
<td>0009</td>
<td>0.9</td>
<td>10695.2</td>
<td>298</td>
</tr>
<tr>
<td>0010</td>
<td>4.0</td>
<td>25025.0</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>4.4</td>
<td>22655.2</td>
<td>1919</td>
</tr>
<tr>
<td>0012</td>
<td>3.5</td>
<td>28208.7</td>
<td>1919</td>
</tr>
</tbody>
</table>
Query1 or CURRENCY_CONVERSION:
What are the bid values in Euro’s? Illustrates a simple map.

SparkRunner (RDD)

SparkRunner (Dataset)
TPC-DS
TPC-DS Benchmark

TPC-DS is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance.

- **Industry standard** benchmark (OLAP/Data Warehouse)
  - [http://www.tpc.org/tpcds/](http://www.tpc.org/tpcds/)

- Implemented for many analytical processing systems
  - RDBMS, Apache Spark, Apache Flink, etc

- **Wide range** of different queries (SQL)

- Existing tools to generate input data of different sizes
TPC-DS: Basic tables schema
TPC-DS: Input Data

Data source:

- Input files are generated with CLI tool (CSV)
- The tool constrains the minimum amount of data to be generated to 1GB.
- TPC-DS dsdgen tool for text (CSV) generation.
  - 3rd-party tools to generate input in different formats (Parquet)

Generated datasets:

- Data size scale factors:
  - 1GB / 10GB / 100GB / 1000GB
TPC-DS: Queries

- **99 distinct** SQL-99 queries (including OLAP extensions)
- Each query answers a **business question**, which illustrates the business context in which the query could be used
- All queries are “*templated*” with random **input parameters**.
- Used to **compare SQL implementation** of completeness and performance
TPC-DS: Query example

Query3 is a good example that contains all main data processing primitives (filtering, aggregation, sorting, selecting, etc)

Report the total extended sales price per item brand of a specific manufacturer for all sales in a specific month of the year.

```
SELECT dt.d_year, item.i_brand_id brand_id, item.i_brand brand,
       SUM(ss_ext_sales_price) sum_agg
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
  AND store_sales.ss_item_sk = item.i_item_sk
  AND item.i_manufact_id = 128
  AND dt.d_moy=11
GROUP BY dt.d_year, item.i_brand, item.i_brand_id
ORDER BY dt.d_year, sum_agg desc, brand_id
LIMIT 100
```
TPC-DS extension in Beam

- **It can be used to:**
  - Compare the performance of Beam SQL for **different runners** and their different versions
  - Run Beam SQL on **different environments**
  - Detect **missing** Beam SQL **features / incompatibilities**
  - Find **performance issues** in Beam

- **Data sources** supported:
  - CSV and Parquet

- **Runners** supported:
  - Dataflow, Spark (RDD and Dataset), Flink

- **25 (of 103) queries** are passing
  - Many queries are not supported by Beam SQL
TPC-DS: Pros/Cons

Pros:

● **Industry** standard benchmark
● Helped to **find** a bunch of Beam **issues** while running on scale
  ○ See a talk: “TPC-DS and Apache Beam - the time has come!”
    (Ismael Mejía/Alexey Romanenko)

Cons:

● Still **under development**
  ○ Requires more attention from Beam community
● Many SQL queries are **not supported** by Beam SQL
  ○ Can't run the whole benchmark
● Only **batch** mode is supported

Infra
Collect runtime metrics

- **Collect metrics**
  - Use Metrics API
    - *TimeMonitor* (Java), *MetricsReader* (Python)
  - Custom collector
    - Nexmark, TPC-DS
- **Store metrics**
  - *BigQuery, InfluxDB*
- **Visualisation**
  - *PerfKit* (past), *Grafana*
Automation: Jenkins

```json
{
  name: 'beam_PerformanceTests_AvroIOIT',
  description: 'Runs performance tests for AvroIOIT',
  githubTitle: 'Java AvroIO Performance Test',
  githubTriggerPhrase: 'Run Java AvroIO Performance Test',
  pipelineOptions: [
    {numberOfRecords: '225000000'},
    {expectedHash: '2f9f5ca33ea464b25109c0297eb6ae6b'},
    {datasetSize: '1089730000'},
    {bigQueryDataset: 'beam_performance'},
    {bigQueryTable: 'avroioit_results'},
    {influxMeasurement: 'avroioit_results'},
    {numWorkers: '5'},
    {autoscalingAlgorithm: 'NONE'}
  ],
}
```

https://ci-beam.apache.org/
Dashboards: Grafana

http://metrics.beam.apache.org/
# Beam Metrics Report

**Possible regression**

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Metric</th>
<th>Runner</th>
<th>Mean previous week</th>
<th>Mean last week</th>
<th>Diff %</th>
<th>Dashboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>go_batch_cogbk_1</td>
<td>dataflow_runtime</td>
<td>-</td>
<td>243.78</td>
<td>274.01</td>
<td>12.4</td>
<td></td>
</tr>
<tr>
<td>go_batch_combine_1</td>
<td>dataflow_runtime</td>
<td>-</td>
<td>408.0</td>
<td>683.05</td>
<td>72.8%</td>
<td></td>
</tr>
<tr>
<td>go_batch_gbk_7</td>
<td>dataflow_runtime</td>
<td>-</td>
<td>107.65</td>
<td>133.68</td>
<td>22.1%</td>
<td></td>
</tr>
<tr>
<td>java_batch_cogbk_1</td>
<td>dataflow_runtime_scc</td>
<td>-</td>
<td>38.54</td>
<td>45.26</td>
<td>17.4%</td>
<td></td>
</tr>
<tr>
<td>java_batch_gbk_7</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>40.36</td>
<td>53.18</td>
<td>31.7%</td>
<td></td>
</tr>
<tr>
<td>java_batch_pardo_1</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>14.63</td>
<td>16.85</td>
<td>15.2%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_gbk_3</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>66.34</td>
<td>85.2</td>
<td>28.5%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_gbk_5</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>594.44</td>
<td>663.12</td>
<td>11.5%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_gbk_6</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>143.59</td>
<td>168.9</td>
<td>17.6%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_gbk_7</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>157.25</td>
<td>189.98</td>
<td>20.6%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_pardo_1</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>18.16</td>
<td>20.29</td>
<td>11.7%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_pardo_2</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>29.1</td>
<td>34.64</td>
<td>19.0%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_pardo_3</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>146.98</td>
<td>169.42</td>
<td>15.2%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_pardo_4</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>24.17</td>
<td>26.76</td>
<td>10.6%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_pardo_2</td>
<td>python_dataflow_streaming_pardo_2_runtime</td>
<td>-</td>
<td>2921.57</td>
<td>3254.8</td>
<td>11.4%</td>
<td></td>
</tr>
<tr>
<td>tcrecordit_results</td>
<td>read_time</td>
<td>-</td>
<td>15.5</td>
<td>18.2</td>
<td>17.8%</td>
<td></td>
</tr>
<tr>
<td>tcrecordit_results</td>
<td>write_time</td>
<td>-</td>
<td>24.96</td>
<td>28.79</td>
<td>15.3%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_11_batch</td>
<td>RuntimeMs</td>
<td>FlinkRunner</td>
<td>287.89</td>
<td>319.59</td>
<td>11.0%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_11_sql_streaming</td>
<td>RuntimeMs</td>
<td>DataflowRunner</td>
<td>36914.86</td>
<td>42544.73</td>
<td>15.2%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_14_batch</td>
<td>RuntimeMs</td>
<td>FlinkRunner</td>
<td>538.61</td>
<td>617.91</td>
<td>14.7%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_14_streaming</td>
<td>RuntimeMs</td>
<td>DataflowRunner</td>
<td>109147.82</td>
<td>124644.5</td>
<td>14.2%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_15_batch</td>
<td>RuntimeMs</td>
<td>FlinkRunner</td>
<td>474.89</td>
<td>563.91</td>
<td>18.7%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_5_streaming</td>
<td>RuntimeMs</td>
<td>DataflowRunner</td>
<td>203490.09</td>
<td>236982.9</td>
<td>16.4%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_9_batch</td>
<td>RuntimeMs</td>
<td>FlinkRunner</td>
<td>303.61</td>
<td>347.77</td>
<td>14.5%</td>
<td></td>
</tr>
</tbody>
</table>

**Possible improvement**

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Metric</th>
<th>Runner</th>
<th>Mean previous week</th>
<th>Mean last week</th>
<th>Diff %</th>
<th>Dashboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>go_batch_sideinput_3</td>
<td>dataflow_runtime</td>
<td>-</td>
<td>2.93</td>
<td>2.44</td>
<td>14.7%</td>
<td></td>
</tr>
<tr>
<td>java_batch_cogbk_3</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>16.07</td>
<td>13.47</td>
<td>19.7%</td>
<td></td>
</tr>
<tr>
<td>java_batch_cogbk_3</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>15.92</td>
<td>13.11</td>
<td>18.9%</td>
<td></td>
</tr>
<tr>
<td>java_batch_gbk_3</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>17.1</td>
<td>14.73</td>
<td>19.7%</td>
<td></td>
</tr>
<tr>
<td>java_batch_gbk_4</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>26.6</td>
<td>23.23</td>
<td>13.7%</td>
<td></td>
</tr>
<tr>
<td>java_batch_gbk_5</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>20.92</td>
<td>17.87</td>
<td>14.7%</td>
<td></td>
</tr>
<tr>
<td>java_batch_gbk_6</td>
<td>dataflow_runtime_scc</td>
<td>-</td>
<td>42.51</td>
<td>34.2</td>
<td>19.8%</td>
<td></td>
</tr>
<tr>
<td>java_batch_pardo_3</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>13.72</td>
<td>11.58</td>
<td>16.5%</td>
<td></td>
</tr>
<tr>
<td>java_batch_pardo_3</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>16.6</td>
<td>14.2</td>
<td>14.4%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_gbk_1</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>3396.29</td>
<td>1461.72</td>
<td>133.7%</td>
<td></td>
</tr>
<tr>
<td>java_streaming_gbk_1</td>
<td>dataflow_v2_javal1_runtime_sec</td>
<td>-</td>
<td>3338.54</td>
<td>1258.08</td>
<td>153.3%</td>
<td></td>
</tr>
<tr>
<td>python_batch_combine_5</td>
<td>python_dataflow_batch_combine_5_runtime</td>
<td>-</td>
<td>53.43</td>
<td>36.4</td>
<td>48.9%</td>
<td></td>
</tr>
<tr>
<td>python_batch_gbk_3</td>
<td>python_dataflow_batch_gbk_3_runtime</td>
<td>-</td>
<td>27.71</td>
<td>24.0</td>
<td>15.4%</td>
<td></td>
</tr>
<tr>
<td>python_batch_sidemput_5</td>
<td>python_dataflow_batch_sideinput_5_runtime</td>
<td>-</td>
<td>46.14</td>
<td>40.8</td>
<td>14.6%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_cogbk_1</td>
<td>python_dataflow_streaming_cogbk_1_runtime</td>
<td>-</td>
<td>5931.0</td>
<td>1449.2</td>
<td>27.9%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_cogbk_2</td>
<td>python_dataflow_streaming_cogbk_2_runtime</td>
<td>-</td>
<td>1164.0</td>
<td>414.6</td>
<td>176.3%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_cogbk_3</td>
<td>python_dataflow_streaming_cogbk_3_runtime</td>
<td>-</td>
<td>10235.0</td>
<td>544.4</td>
<td>189.8%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_gbk_3</td>
<td>python_dataflow_streaming_gbk_3_runtime</td>
<td>-</td>
<td>147.86</td>
<td>69.8</td>
<td>115.7%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_gbk_6</td>
<td>python_dataflow_streaming_gbk_6_runtime</td>
<td>-</td>
<td>2378.33</td>
<td>649.6</td>
<td>267.2%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_gbk_7</td>
<td>python_dataflow_streaming_gbk_7_runtime</td>
<td>-</td>
<td>2211.67</td>
<td>530.5</td>
<td>269.3%</td>
<td></td>
</tr>
<tr>
<td>python_streaming_pardo_1</td>
<td>python_dataflow_streaming_pardo_1_runtime</td>
<td>-</td>
<td>251.43</td>
<td>219.0</td>
<td>12.9%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_12_batch</td>
<td>RuntimeMs</td>
<td>FlinkRunner</td>
<td>204.25</td>
<td>176.86</td>
<td>13.8%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_16_batch</td>
<td>RuntimeMs</td>
<td>FlinkRunner</td>
<td>250.21</td>
<td>214.0</td>
<td>14.4%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_6_streaming</td>
<td>RuntimeMs</td>
<td>FlinkRunner</td>
<td>734.86</td>
<td>607.18</td>
<td>17.3%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_7_sql_batch</td>
<td>RuntimeMs</td>
<td>DirectRunner</td>
<td>504902.43</td>
<td>371615.77</td>
<td>122.4%</td>
<td></td>
</tr>
<tr>
<td>nextraxk_7_sql_streaming</td>
<td>RuntimeMs</td>
<td>DirectRunner</td>
<td>2184798.29</td>
<td>1417686.68</td>
<td>53.4%</td>
<td></td>
</tr>
</tbody>
</table>
Some conclusions

- Performance measuring is **CRUCIAL** important!
- **Java SDK** is pretty well covered by different performance testing suites and benchmarks
- **Python SDK, Go SDK** and **Cross-Language** pipelines are missing the benchmarks
- We don’t run regularly the performance tests on **large datasets** and at **real scale**
  - It helps to find the specific issues
- Beam is in a good shape on this but...
Want to contribute?

Examples of things to do:

- Add perf tests / benchmarks for Python and Go SDKs
- Add more runners to run regularly
  - Portable runners including!
- Automate perf regressions with “git bisect”
  - Grafana alerts
  - Add to release testing routine
- Make TPC-DS in Beam more mature and part of release testing
- Add benchmarks/tests of your choice
- ... etc
References

Nexmark:

- Main doc: https://datalab.cs.pdx.edu/niagara/NEXMark/
- Beam: https://beam.apache.org/documentation/sdks/java/testing/nexmark/
- Wiki: https://cwiki.apache.org/confluence/display/BEAM/Nexmark

TPC-DS:

- Website: https://www.tpc.org/tpcds/default5.asp
- Beam: https://beam.apache.org/documentation/sdks/java/testing/tpcds/
Thanks!