Nexmark with Beam
Evaluating Big Data systems with Apache Beam

Etienne Chauchot, Ismaël Mejía. *Talend*
Who are we?
Agenda

1. Big Data Benchmarking
   a. State of the art
   b. NEXMark: A benchmark over continuous data streams
2. Nexmark on Apache Beam
   a. Introducing Beam
   b. Advantages of using Beam for benchmarking
   c. Implementation
   d. Nexmark + Beam: a win-win story
   e. Neutral benchmarking: a difficult issue
   f. Example: Running Nexmark on Spark
3. Current state and future work
Big Data benchmarking
Benchmarking

Why do we benchmark?
1. Performance
2. Correctness

Benchmark suites steps:
1. Generate data
2. Compute data
3. Measure performance
4. Validate results

Types of benchmarks
- Microbenchmarks
- Functional
- Business case
- Data Mining / Machine Learning
Issues of Benchmarking Suites for Big Data

- **No standard** suite: Terasort, TPCx-HS (Hadoop), HiBench, ...
- No common model/API: Strongly tied to each processing engine or SQL
- Too focused on **Hadoop** infrastructure
- Mixed Benchmarks for storage/processing
- Few Benchmarking suites support **streaming**: Yahoo Benchmark, HiBench
State of the art

Batch

- **Terasoft**: Sort random data
- **TPCx-HS**: Sort to measure Hadoop compatible distributions
- **TPC-DS on Spark**: TPC-DS business case with Spark SQL
- **Berkeley Big Data Benchmark**: SQL-like queries on Hive, Redshift, Impala
- **HiBench** and **BigBench**

Streaming

- **Yahoo Streaming Benchmark**

* HiBench includes also some streaming / windowing benchmarks
Nexmark

Benchmark for queries over Data Streams
Business case: **Online Auction System**
Research paper draft 2004

Example:
Query 4: What is the average selling price for each auction category?
Query 8: Who has entered the system and created an auction in the last period?
Nexmark on Google Dataflow

- Port of SQL style queries described in the NEXMark research paper to Google Cloud Dataflow by Mark Shields and others at Google.
- Enriched queries set with Google Cloud Dataflow client use cases
- Used as rich integration testing scenario on the Google Cloud Dataflow
Nexmark on Beam
1. The Beam Programming Model
2. SDKs for writing Beam pipelines -- Java/Python
3. Runners for existing distributed processing backends
The Beam Model: **What** is Being Computed?

**Event Time:** timestamp when the event happened

**Processing Time:** wall clock absolute program time
The Beam Model: **Where** in Event Time?

- Split infinite data into finite chunks
The Beam Model: *Where* in Event Time?
Apache Beam pipeline

Data processing pipeline
(executed via a Beam runner)

- Read (Source)
  - Input
    - PCollection
    - KafkaIO
- PTransform
  - Window per min
- PTransform
  - Count
- Write (sink)
  - Output
    - HDFS
Apache Beam - Programming Model

Element-wise

- ParDo -> DoFn
- MapElements
- FlatMapElements
- Filter
  - WithKeys
  - Keys
  - Values

Grouping

- GroupByKey
- CoGroupByKey
- Combine -> Reduce
  - Sum
  - Count
  - Min / Max
  - Mean
  - ...

Windowing/Triggers

- Windows
  - FixedWindows
  - GlobalWindows
  - SlidingWindows
  - Sessions
- Triggers
  - AfterWatermark
  - AfterProcessingTime
  - Repeatedly
Nexmark on Apache Beam

- Nexmark was ported from Dataflow to Beam 0.2.0 as an integration test case
- Refactored to most recent Beam version
- Made code more generic to support all the Beam runners
- Changed some queries to use new APIs
- Validated queries in all the runners to test their support of the Beam model
Advantages of using Beam for benchmarking

- Rich model: all use cases that we had could be expressed using Beam API
- Can test both batch and streaming modes with exactly the same code
- Multiple runners: queries can be executed on Beam supported runners (provided that the given runner supports the used features)
- Monitoring features (metrics)
Implementation
Components of Nexmark

- **Generator:**
  - generation of timestamped events (bids, persons, auctions) correlated between each other

- **NexmarkLauncher:**
  - creates sources that use the generator
  - queries pipelines launching, monitoring

- **Output metrics:**
  - Each query includes ParDos to update metrics
  - execution time, processing event rate, number of results, but also invalid auctions/bids, …

- **Modes:**
  - Batch mode: test data is finite and uses a BoundedSource
  - Streaming mode: test data is finite but uses an UnboundedSource to trigger streaming mode in runners
Some of the queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
<th>Use of Beam model</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Who is selling in particular US states?</td>
<td>Join, State, Timer</td>
</tr>
<tr>
<td>5</td>
<td>Which auctions have seen the most bids in the last period?</td>
<td>Sliding Window, Combiners</td>
</tr>
<tr>
<td>6</td>
<td>What is the average selling price per seller for their last 10 closed auctions?</td>
<td>Global Window, Custom Combiner</td>
</tr>
<tr>
<td>7</td>
<td>What are the highest bids per period?</td>
<td>Fixed Windows, Side Input</td>
</tr>
<tr>
<td>9</td>
<td>Winning bids</td>
<td>Custom Window</td>
</tr>
<tr>
<td>11</td>
<td>How many bids did a user make in each session he was active?</td>
<td>Session Window, Triggering</td>
</tr>
<tr>
<td>12</td>
<td>How many bids does a user make within a fixed processing time limit?</td>
<td>Global Window, working in Processing Time</td>
</tr>
</tbody>
</table>

*: not in original NexMark paper
Query structure

1. Get `PCollection<Event>` as input
2. Apply `ParDo + Filter` to extract object of interest: Bids, Auctions, Person(s)
3. Apply transforms: `Filter, Count, GroupByKey, Window`, etc.
4. Apply `ParDo` to output the final `PCollection`: collection of `AuctionPrice`, `AuctionCount` ...
Key point: **When** to compute data?

- **Windows**: divide data into event-time-based finite chunks.
  - Often required when doing aggregations over unbounded data

![Diagram showing Fixed, Sliding, Sessions, and GlobalWindow windows](image-url)
Key point: **When** to compute data?

- **Triggers**: condition to fire computation
- Default trigger: at the end of the window
- Required when working on unbounded data in Global Window
- Q11: trigger fires when 20 elements were received
Key point: **When** to compute data?

- Q12: Trigger fired when first element is received + delay (works in processing in global window time to create a duration)
- **Processing time**: wall clock absolute program time
- **Event time**: timestamp in which the event occurred
Key point: How to make a join?

- CoGroupByKey (in Q3, Q8, Q9): groups values of PCollections<KV> that share the same key
  - Join Auctions and Persons by their person id and tag them

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>personId = 1</td>
<td>{Mark, NYC}</td>
</tr>
<tr>
<td>personId = 2</td>
<td>{William, Paris}</td>
</tr>
<tr>
<td>personId = 3</td>
<td>{Jack, Miami}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sellerId = 1</td>
<td>{Car, 2017/05/05}</td>
</tr>
<tr>
<td>sellerId = 4</td>
<td>{Noisy dog, 2017/05/23}</td>
</tr>
<tr>
<td>sellerId = 2</td>
<td>{Bike, 2017/05/15}</td>
</tr>
</tbody>
</table>

Join

<table>
<thead>
<tr>
<th>Key</th>
<th>CoGroupByKeyResult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{Mark, NYC}, {Car, 2017/05/05}</td>
</tr>
<tr>
<td>2</td>
<td>{William, Paris}, {Bike, 2017/05/15}</td>
</tr>
</tbody>
</table>
Key point: How to temporarily group events?

- Custom window function (in Q9)
  - As CoGroupByKey is per window, need to put bids and auctions in the same window before joining them.
Key point: How to deal with **out of order** events?

- State and Timer APIs in an incremental join (Q3):
  - Memorize person event waiting for corresponding auctions and clear at timer
  - Memorize auction events waiting for corresponding person event
Key point: How to **tweak reduction phase**?

Custom combiner (in Q6) to be able to specify

1. how elements are added to accumulators
2. how accumulators merge
3. how to extract final data

...to calculate the average price of the last 3 closed auctions...
Conclusion on queries

- Wide coverage of the Beam API
  - Most of the API
  - Illustrates also working in processing time
- Realistic
  - Real use cases, valid queries for an end user auction system
  - Extra queries inspired by Google Cloud Dataflow client use cases
- Complex queries
  - Leverage all the runners capabilities
Beam + Nexmark = A win-win story

- Streaming test
- A/B testing of big data execution engines (regression and performance comparison between 2 versions of the same engine or of the same runner, ...)
- Integration testing (SDK with runners, runners with engines, ...)
- Validate Beam runners capability matrix
Benchmarking results
Neutral benchmarking: a difficult issue

- Different levels of support of features of the Beam model among runners
- All runners have different strengths: we would end up comparing things that are not always comparable
  - Some runners were designed to be batch oriented, others streaming oriented
  - Some are designed towards sub-second latency, others support auto-scaling
- Runners can have multiple knobs to tweak the options
- The nondeterministic part of distributed environments
- Benchmarking on the cloud (e.g. Messy neighbors)
### Execution Matrix

#### Batch

<table>
<thead>
<tr>
<th>Runner</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1035</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1037</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Streaming

<table>
<thead>
<tr>
<th>Runner</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1035</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2112</td>
<td></td>
</tr>
<tr>
<td>Apex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1037</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some workload configuration items

● Events generation
  ○ 100 000 events generated with 100 generator threads
  ○ Event rate in SIN curve
  ○ Initial event rate of 10 000
  ○ Event rate step of 10 000
  ○ 100 concurrent auctions
  ○ 1000 concurrent persons putting bids or creating auctions

● Pipelines
  ○ probabilities:
    ■ hot actions = ½
    ■ hot bidders = ¼
    ■ hot sellers = ¼

● Technical
  ○ No artificial CPU load
  ○ No artificial IO load

● Windows
  ○ size 10s
  ○ sliding period 5s
  ○ watermark hold for 0s
## Nexmark Output - Spark Runner (Batch)

<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>3.8</td>
<td>26267.4</td>
<td>100000</td>
</tr>
<tr>
<td>0001</td>
<td>3.5</td>
<td>28232.6</td>
<td>92000</td>
</tr>
<tr>
<td>0002</td>
<td>3.6</td>
<td>27964.2</td>
<td>713</td>
</tr>
<tr>
<td>0003</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>0004</td>
<td>10.0</td>
<td>10006.0</td>
<td>50</td>
</tr>
<tr>
<td>0005</td>
<td>5.8</td>
<td>17214.7</td>
<td>3</td>
</tr>
<tr>
<td>0006</td>
<td>9.4</td>
<td>10642.8</td>
<td>1631</td>
</tr>
<tr>
<td>0007</td>
<td>7.4</td>
<td>13539.1</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>7.2</td>
<td>13861.9</td>
<td>6000</td>
</tr>
<tr>
<td>0009</td>
<td>9.5</td>
<td>10517.5</td>
<td>5243</td>
</tr>
<tr>
<td>0010</td>
<td>5.9</td>
<td>16877.6</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>5.8</td>
<td>17388.3</td>
<td>1992</td>
</tr>
<tr>
<td>0012</td>
<td>5.5</td>
<td>18181.8</td>
<td>1992</td>
</tr>
</tbody>
</table>
## Nexmark Output - Spark Runner (Streaming)

<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>1.0</td>
<td>10256.1</td>
<td>100000</td>
</tr>
<tr>
<td>0001</td>
<td>1.3</td>
<td>7722.1</td>
<td>92000</td>
</tr>
<tr>
<td>0002</td>
<td>0.7</td>
<td>14705.8</td>
<td>713</td>
</tr>
<tr>
<td>0003</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>0004</td>
<td>17.3</td>
<td>5779.7</td>
<td>50</td>
</tr>
<tr>
<td>0005</td>
<td>16.6</td>
<td>6020.8</td>
<td>3</td>
</tr>
<tr>
<td><strong>0006</strong></td>
<td><strong>26.5</strong></td>
<td><strong>3773.4</strong></td>
<td><strong>1631</strong></td>
</tr>
<tr>
<td>0007</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>0008</td>
<td>12.3</td>
<td>8142.0</td>
<td>6000</td>
</tr>
<tr>
<td>0009</td>
<td>17.7</td>
<td>5650.0</td>
<td>5243</td>
</tr>
<tr>
<td>0010</td>
<td>13.1</td>
<td>768.8</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>10.0</td>
<td>9962.1</td>
<td>1992</td>
</tr>
<tr>
<td>0012</td>
<td>10.2</td>
<td>9783.8</td>
<td>1992</td>
</tr>
</tbody>
</table>
Comparing different versions of the Spark engine
Current Status

Future work
## Current state

<table>
<thead>
<tr>
<th>Issue</th>
<th>Title</th>
<th>Tags</th>
<th>Author</th>
<th>Opened</th>
<th>Assignee</th>
</tr>
</thead>
<tbody>
<tr>
<td>#53</td>
<td>query7: fails in spark in streaming mode, need support for PCollectionView in spark in streaming mode</td>
<td>bug, needs-upstream-fix, spark</td>
<td>echauchot</td>
<td>14 days ago</td>
<td></td>
</tr>
<tr>
<td>#50</td>
<td>query3: fails on apex runner (state/timer) support</td>
<td>apex, bug, needs-upstream-fix, queries</td>
<td>imejia</td>
<td>15 days ago</td>
<td></td>
</tr>
<tr>
<td>#49</td>
<td>all queries: flink never stops in streaming mode, timeout/waitUntilFinish not supported</td>
<td>flink, needs-upstream-fix, queries</td>
<td>echauchot</td>
<td>17 days ago</td>
<td></td>
</tr>
<tr>
<td>#44</td>
<td>query3: fails on spark runner (state/timer) support</td>
<td>bug, needs-upstream-fix, queries, spark</td>
<td>echauchot</td>
<td>22 days ago</td>
<td></td>
</tr>
<tr>
<td>#20</td>
<td>all queries: Support tests for all</td>
<td>enhancement, queries, tests</td>
<td>echauchot</td>
<td>17 Mar</td>
<td></td>
</tr>
</tbody>
</table>

- Manage Nexmark issues in a dedicated place
- 5 open issues / 46 closed issues
Nexmark helped discover bugs and missing features in Beam
9 open issues / 6 closed issues on Beam upstream
Nexmark PR is opened, expected to be merged in Beam master soon
Future work

- Resolve open NexMark and Beam issues
- More queries to evaluate corner cases
- Validate New runners: Gearpump / JStorm
- Streaming SQL-based queries (using the ongoing work on Calcite DSL)
Contribute

You are welcome to contribute!

- 5 open Github issues and 9 Beam Jiras that need to be taken care of
- Improve documentation + more refactoring
- New ideas, more queries, support for IOs, etc
Greetings

- **Mark Shields** (Google): Contributing Nexmark + answering our questions.
- **Thomas Groh, Kenneth Knowles** (Google): Direct runner + State/Timer API.
- **Aljoscha Krettek** (data Artisans), **Jinsong Lee** (Ali Baba): Flink Runner.
- **Alexiane, Thomas Fion, Abbass Marouni, Jean-Baptiste Onofre, Ryan Skraba** (Talend): General comments/ideas and help to run Nexmark in our YARN cluster.
- The rest of the **Beam** community in general for being awesome.
References

Apache Beam
NexMark
BEAM-160
NexMark on Beam Issues
Big Data Benchmarks
Questions?
Thanks
Addendum
Query 3: Who is selling in particular US states?

- Illustrates incremental join of the auctions and the persons collections
- uses global window and using per-key state and timer APIs
  - Apply **global window** to events with **trigger repeatedly after at least nbEvents in pane** => results will be materialized each time nbEvents are received.
  - **input1**: collection of **auctions** events **filtered** by category and **keyed** by seller id
  - **input2**: collection of **persons** events **filtered** by US state codes and **keyed** by person id
  - **CoGroupByKey** to group **auctions** and **persons** by personId/sellerId + tags to distinguish persons and auctions
  - **ParDo** to do the **incremental join**: auctions and person events can arrive out of order
    - **person** element **stored** in persistent state in order to match future auctions by that person. Set a **timer** to **clear** the person state after a TTL
    - **auction** elements **stored** in persistent state until we have seen the corresponding person record. Then, it can be **output** and **cleared**
  - **output** NameCityStateld(person.name, person.city, person.state, auction.id) objects
Query 5: Which auctions have seen the most bids in the last period?

- Illustrates sliding windows and combiners (i.e. reducers) to compare the elements in auctions Collection
  - **Input:** (sliding) window (to have a **result over** 1h period **updated** every 1 min) collection of bids events
  - **ParDo** to **replace** bid elements by their auction id
  - **Count.PerElement** to **count the occurrences** of each auction id
  - **Combine.globally** to **select** only the auctions with the **maximum number** of bids
    - BinaryCombineFn to **compare** one to one the elements of the collection (auction id occurrences, i.e. number of bids)
    - Return KV(auction id, max occurrences)
  - **output:** AuctionCount(auction id, max occurrences) objects
Query 6: What is the average selling price per seller for their last 10 closed auctions?

- Illustrates specialized combiner that allows specifying the combine process of the bids
  - **input:** winning-bids keyed by seller id
  - GlobalWindow + trigerring at each element (to have a continuous flow of updates at each new winning-bid)
  - Combine.perKey to calculate **average price** of last 10 winning bids for each seller.
    - create Arraylist accumulators for chunks of data
    - add all elements of the chunks to the accumulators, sort them by bid time stamp then price keeping last 10 elements
    - iteratively merge the accumulators until there is only one: just add all bids of all accumulators to a final accumulator and sort by time stamp then price keeping last 10 elements
    - extractOutput: sum all the prices of the bids and divide by accumulator size
  - output SellerPrice(sellerId, avgPrice) object
Query 7: What are the highest bids per period?

- Could have been implemented with a combiner like query5 but deliberately implemented using Max(prices) as a side input and illustrate fanout.
- Fanout is a redistribution using an intermediate implicit combine step to reduce the load in the final step of the Max transform
  - **input**: (fixed) windowed collection of bids events
  - **ParDo** to replace bids by their price
  - **Max.withFanout** to get the max per window and use it as a side input for next step. Fanout is useful if there are many events to be computed in a window using the Max transform.
  - **ParDo** on the bids with side input to output the bid if bid.price equals maxPrice (that comes from side input)
Query 9 Winning-bids (not part of original NexMark): extract the most recent of the highest bids

- Illustrates custom window function to reconcile auctions and bids + join them
  - **input**: collection of events
  - Apply custom windowing function to **temporarily reconcile** auctions and bids events in the same custom window (AuctionOrBidWindow)
    - assign auctions to window [auction.timestamp, auction.expiring]
    - assign bids to window [bid.timestamp, bid.timestamp + expectedAuctionDuration (generator configuration parameter)]
    - merge all 'bid' windows into their corresponding 'auction' window, provided the auction has not expired.
  - Filter + ParDo to **extract** auctions out of events and key them by auction id
  - Filter + ParDo to **extract** bids out of events and key them by auction id
  - CogroupByKey (groups values of PCollections<KV> that share the same key) to **group** auctions and bids by auction id + tags to distinguish auctions and bids
  - ParDo to
    - determine best bid price: verification of valid bid, sort prices by price ASC then time DESC and keep the max price
    - and output AuctionBid(auction, bestBid) objects
Query 11 (not part of original NexMark): How many bids did a user make in each session he was active?

- Illustrates session windows + triggering on the bids collection
  - **input**: collection of bids events
  - **ParDo** to replace bids with their bidder id
  - Apply session windows with gap duration = windowDuration (configuration item) and trigger repeatedly after at least nbEvents in pane => each window (i.e. session) will contain bid ids received since last windowDuration period of inactivity and materialized every nbEvents bids
  - **Count.perElement** to count bids per bidder id (number of occurrences of bidder id)
  - **output** idsPerSession(bidder, bidsCount) objects
Query 12 (not part of original NexMark): How many bids does a user make within a fixed processing time limit?

- Illustrates working in processing time in the Global window to count occurrences of bidder
  - **input**: collection of bid events
  - **ParDo** to replace bids by their bidder id
  - Apply global window with trigger repeatedly after processingTime pass the **first** element in pane + windowDuration (configuration item) => each pane will contain elements processed within windowDuration time
  - **Count.perElement** to count bids per bidder id (occurrences of bidder id)
  - **output** BidsPerWindow(bidder, bidsCount) objects