Lightweight Primitives for Online Tuning

by Tomer Kaftan (UW), Magdalena Balazinska (UW), Alvin Cheung (UW), Johannes Gehrke (Microsoft)
Data processing workloads today are complicated.
Motivating Workload
Motivating Workload

“A Cuttlefish pretending to be a rock”

*Image Sourced from https://www.flickr.com/photos/silkebaron/32001215104
Motivating Workload

“A Cuttlefish pretending to be a rock”

*Image Sourced from https://www.flickr.com/photos/silkebaron/32001215104

Generate Training Data from:

- flickr
- imgur
- etc.
Motivating Workload

*caption-generating model portion of the logical plan inspired by: Xu et al. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. ICML 2015*
Motivating Workload

Diverse, sophisticated operators, with multiple implementations!

*caption-generating model portion of the logical plan inspired by: Xu et al. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. ICML 2015*
Example Operator: Convolution
Example Operator: Convolution

Tested 3 convolution algorithms on 8000 Flickr images

Relative throughput normalized against the highest-throughput algorithm.
Traditionally:

Use a Query Optimizer
Traditionally:

Use a Query Optimizer

(Collect Dataset Statistics, Apply Heuristics & Cost Models)
These work great, BUT...
These work great, BUT…

• Designing good query optimizers takes time!
These work great, BUT...

- Designing good query optimizers takes time!

- Requires deep knowledge of the operators and significant development effort.
These work great, BUT...

• Designing good query optimizers takes time!

• Requires deep knowledge of the operators and significant development effort.

• Spark SQL took 2 years to go from heuristics-based optimization to cost-based optimization! [1]

These work great, BUT...

- Designing good query optimizers takes time!

- Requires deep knowledge of the operators and significant development effort.

- Spark SQL took 2 years to go from heuristics-based optimization to cost-based optimization! [1]


- Modern data processing applications involve more than just relational operators!
Can we optimize without a full-fledged optimizer?
Workload developer (or the query optimizer) inserts calls to Cuttlefish’s API to insert tuners that select implementations *during execution*. 
Workload developer (or the query optimizer) inserts calls to Cuttlefish’s API to insert tuners that select implementations during execution.
Workload developer (or the query optimizer) inserts calls to Cuttlefish’s API to insert tuners that select implementations *during execution*
Workload developer (or the query optimizer) inserts calls to Cuttlefish’s API to insert tuners that select implementations \textit{during execution}
Cuttlefish: A Lightweight Primitive for Online Tuning

The user maps tuning rounds to the execution model of each operator:
The user maps tuning rounds to the execution model of each operator:

- **Regex**: One round per HTML Doc
Workload developer (or the query optimizer) inserts calls to Cuttlefish’s API to insert tuners that select implementations during execution.

The user maps tuning rounds to the execution model of each operator:

- **Regex**: One round per HTML Doc
- **Convolve**: One round per image
The user maps tuning rounds to the execution model of each operator:

- **Regex**: One round per HTML Doc
- **Convolve**: One round per image
- **Parallel Distributed Join**: One round per partition
I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion
The Cuttlefish Primitive
1. Construct a tuner (from a set of choices)
1. Construct a tuner (from a set of choices)

2. Tuner.choose (pick one of the choices)
The Cuttlefish Primitive

1. Construct a tuner (from a set of choices)
2. Tuner.choose (pick one of the choices)
3. Tuner.observe (observe a reward for a choice)
The Cuttlefish Primitive

1. Construct a tuner (from a set of choices)

2. Tuner.choose (pick one of the choices)

3. Tuner.observe (observe a reward for a choice)

Cuttlefish tuners maximize the total reward after multiple choose-observe tuning rounds
Tuning Convolution with Cuttlefish
Tuning Convolution with Cuttlefish

def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
def loopConvolve(image, filters): ...
def fftConvolve(image, filters): ...
def mmConvolve(image, filters): ...
for image, filters in convolutions:
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    reward = computeReward(elapsedTime)
    output result

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
Tuning Convolution with Cuttlefish

def loopConvolve(image, filters): ...
def fftConvolve(image, filters): ...
def mmConvolve(image, filters): ...
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])

for image, filters in convolutions:

Tuning Convolution with Cuttlefish

def loopConvolve(image, filters): 

def fftConvolve(image, filters): 

def mmConvolve(image, filters): 

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])

for image, filters in convolutions:
    convolve, token = tuner.choose()
Tuning Convolution with Cuttlefish

def loopConvolve(image, filters): ...
def fftConvolve(image, filters): ...
def mmConvolve(image, filters): ...
for image, filters in convolutions:
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    output result

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
for image, filters in convolutions:
    convolve, token = tuner.choose()
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
Tuning Convolution with Cuttlefish

def loopConvolve(image, filters): ...
def fftConvolve(image, filters): ...
def mmConvolve(image, filters): ...

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])

for image, filters in convolutions:
    convolve, token = tuner.choose()
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    reward = computeReward(elapsedTime)
Tuning Convolution with Cuttlefish

def loopConvolve(image, filters): ...
def fftConvolve(image, filters): ...
def mmConvolve(image, filters): ...
for image, filters in convolutions:
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    reward = computeReward(elapsedTime)
    output result

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])

for image, filters in convolutions:
    convolve, token = tuner.choose()
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    reward = computeReward(elapsedTime)
    tuner.observe(token, reward)
Tuning Convolution with Cuttlefish

def loopConvolve(image, filters):

def fftConvolve(image, filters):

def mmConvolve(image, filters):

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])

for image, filters in convolutions:
    convolve, token = tuner.choose()
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    reward = computeReward(elapsedTime)
    tuner.observe(token, reward)

output result
I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion
Approach: Tuning
Approach: Tuning

Multi-armed Bandit Problem
Approach: Tuning

Multi-armed Bandit Problem

- K possible choices (called arms)
Multi-armed Bandit Problem

- $K$ possible choices (called arms)
- Arms have unknown reward distributions
Multi-armed Bandit Problem

- K possible choices (called arms)
- Arms have unknown reward distributions
- At each round: select an Arm and observe a reward
Multi-armed Bandit Problem

- K possible choices (called arms)
- Arms have unknown reward distributions
- At each round: select an Arm and observe a reward

Goal: Maximize Cumulative Reward (by balancing exploration & exploitation)
Thompson Sampling
Thompson Sampling

Belief distributions about expected reward

Reward

Arm 1
Arm 2
Arm 3
Arm 4
Thompson Sampling

Reward

Arm 1  Arm 2  Arm 3  Arm 4
Thompson Sampling

Reward

Arm 1  Arm 2  Arm 3  Arm 4
Thompson Sampling

Arm 1  Arm 2  Arm 3  Arm 4

Reward
Thompson Sampling

Reward

Arm 1  Arm 2  Arm 3  Arm 4
Thompson Sampling

- Better arms chosen more often

Reward

Arm 1

Arm 2

Arm 3

Arm 4

20
Thompson Sampling
Thompson Sampling

- Gaussian runtimes with initially unknown means and variances
Thompson Sampling

• Gaussian runtimes with initially unknown means and variances
• Belief distributions form t-distributions
  • Depend only on sample mean, variance, count
Thompson Sampling

- Gaussian runtimes with initially unknown means and variances
- Belief distributions form t-distributions
  - Depend only on sample mean, variance, count
- No meta-parameters, yet works well for diverse operators
Thompson Sampling

- Gaussian runtimes with initially unknown means and variances
- Belief distributions form t-distributions
  - Depend only on sample mean, variance, count
- No meta-parameters, yet works well for diverse operators
- Constant memory overhead, 0.03 ms per tuning round
Convolution Evaluation

- Prototype in Apache Spark
Convolution Evaluation

- Prototype in Apache Spark

- Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
  - Reward: -1*elapsedTime (maximizes throughput)
Convolution Evaluation

• Prototype in Apache Spark

• Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
  • Reward: -1*elapsedTime (maximizes throughput)

• Convolve 8000 Flickr images with sets of filters (~32gb)
  • Vary number & size of filters
Convolution Evaluation

- Prototype in Apache Spark

- Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
  - Reward: \(-1 \times \text{elapsedTime}\) (maximizes throughput)

- Convolve 8000 Flickr images with sets of filters (~32gb)
  - Vary number & size of filters

- Compute intensive
  - (Some configs up to 45 min on a single node)
Convolution Evaluation

• Prototype in Apache Spark

• Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
  • Reward: -1*elapsedTime (maximizes throughput)

• Convolve 8000 Flickr images with sets of filters (~32gb)
  • Vary number & size of filters

• Compute intensive
  • (Some configs up to 45 min on a single node)

• Run on an 8-node (AWS EC2 4-core r3.xlarge) cluster.
  • 32 total cores, ~252 images per core
Convolution Results

Relative throughput normalized against the highest-throughput algorithm
Convolution Results

Relative throughput normalized against the highest-throughput algorithm
Convolution Results

Relative throughput normalized against the highest-throughput algorithm.
I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion
Challenges in Distributed Tuning
Challenges in Distributed Tuning

1. Choosing and observing occur throughout a cluster
   • To maximize learning, need to communicate
Challenges in Distributed Tuning

1. Choosing and observing occur throughout a cluster
   • To maximize learning, need to communicate

2. Synchronization & communication overheads
Challenges in Distributed Tuning

1. Choosing and observing occur throughout a cluster
   • To maximize learning, need to communicate

2. Synchronization & communication overheads

3. Feedback delay
   • How many times is `choose` called before an earlier reward is observed?
   • Fortunately, theoretically sound to have delays
Distributed Tuning Approach
Distributed Tuning Approach

Centralized Tuner

Machine 1
Choose/Observe

Machine 2

Machine 3
Distributed Tuning Approach

Centralized Tuner

Machine 1

Choose/Observed

Machine 2

Machine 3

Independent Tuners, Centralized Store

Machine 1

Push Local / Pull Global

Machine 2

Global Model Store

Machine 3
Distributed Tuning Approach

Centralized Tuner

Machine 1

Machine 2

Machine 3

Choose/Observe

Independent Tuners, Centralized Store

Machine 1

Machine 2

Machine 3

Push Local / Pull Global

Global Model Store
Distributed Tuning Approach

Centralized Tuner

Machine 1

Machine 2

Machine 3

Choose/Observed

Independent Tuners, Centralized Store

Machine 1

Machine 2

Machine 3

Push Local / Pull Global

Global Model Store

Peer-to-Peer is also a possibility, but requires more communication
Distributed Tuning Approach

Worker 1
- Local State
- Non-local State
  - Thread 1
  - Thread 2
  - Thread 3

Worker 2
- Local State
- Non-local State
  - Thread 1
  - Thread 2
  - Thread 3

Model Store
- Local State
- Non-local State
  - Worker 1:
    - Local State
  - Worker 2:
    - Local State

*On Master or a Parameter Server*
Distributed Tuning Approach

Worker 1

Local State
Non-local State

Thread 1 Thread 2 Thread 3

Worker 2

Local State
Non-local State

Thread 1 Thread 2 Thread 3

Model Store

Worker 1:
Local State

Worker 2:
Local State

*On Master or a Parameter Server*
Distributed Tuning Approach

• When choosing: aggregate local & non-local state
Distributed Tuning Approach

- When choosing: aggregate local & non-local state
- When observing: update the local state
Distributed Tuning Approach

- When choosing: aggregate local & non-local state
- When observing: update the local state
- Model store aggregates non-local state
Results with Distributed Approach

Relative throughput normalized against the highest-throughput algorithm
Results with Distributed Approach

Throughput normalized against an ideal oracle that always picks the fastest option at each round
Results with Distributed Approach

Throughput normalized against an ideal oracle that always picks the fastest option at each round.
Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning (by learning cost models)

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion
Contextual Tuning
Contextual Tuning

- Best physical operator for each round may depend on current (easy to compute) context
- e.g. convolution performance depends on the image & filter dimensions
• Best physical operator for each round may depend on current (easy to compute) context
  • e.g. convolution performance depends on the image & filter dimensions

• Users may know important context features
  • e.g. from the asymptotic algorithmic complexity
Contextual Tuning

• Best physical operator for each round may depend on current (easy to compute) context
  • e.g. convolution performance depends on the image & filter dimensions

• Users may know important context features
  • e.g. from the asymptotic algorithmic complexity

• Users can specify context in Tuner.choose
Contextual Tuning Algorithm
Contextual Tuning Algorithm

• Linear contextual Thompson sampling learns a linear model that maps features to rewards
Contextual Tuning Algorithm

• Linear contextual Thompson sampling learns a linear model that maps features to rewards

• Feature Normalization & Regularization
  • Increased robustness towards feature choices
Contextual Tuning Algorithm

• Linear contextual Thompson sampling learns a linear model that maps features to rewards

• Feature Normalization & Regularization
  • Increased robustness towards feature choices

• Effectively learns a cost model
Tuning Convolution with Cuttlefish

def loopConvolve(image, filters): ...
def fftConvolve(image, filters): ...
def mmConvolve(image, filters): ...

for image, filters in convolutions:
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    reward = computeReward(elapsedTime)
    output result

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])

convolve, token = tuner.choose()
start = now()
result = convolve(image, filters)
elapsedTime = now() - start
reward = computeReward(elapsedTime)
tuner.observe(token, reward)
def loopConvolve(image, filters): ...
def fftConvolve(image, filters): ...
def mmConvolve(image, filters): ...

def getDimensions(image, filters): ...

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])

for image, filters in convolutions:
    context = getDimensions(image, filters)
    convolve, token = tuner.choose(context)
    start = now()
    result = convolve(image, filters)
    elapsedTime = now() - start
    reward = computeReward(elapsedTime)
    tuner.observe(token, reward)

output result
Contextual Convolution Results

Throughput normalized against an ideal oracle that always picks the fastest algorithm
Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion
Nonstationary Settings
Nonstationary Settings

- Runtimes may drift over time, or differ across nodes
  - heterogeneous cluster, changing resource availabilities, data properties varying throughout the workload, etc.
  - E.g. web crawl data and images may be stored sorted by website. This could correlate with performance
Nonstationary Settings

- Runtimes may drift over time, or differ across nodes
- Heterogenous cluster, changing resource availabilities, data properties varying throughout the workload, etc.
- E.g. web crawl data and images may be stored sorted by website. This could correlate with performance
- We might not be capturing sufficient context!
**Nonstationary Settings**

- Runtimes may drift over time, or differ across nodes
- Heterogenous cluster, changing resource availabilities, data properties varying throughout the workload, etc.
- E.g. web crawl data and images may be stored sorted by website. This could correlate with performance

- We might not be capturing sufficient context!

- Standard multi-armed bandit techniques fail
Nonstationary Settings

• Runtimes may drift over time, or differ across nodes
• heterogenous cluster, changing resource availabilities, data properties varying throughout the workload, etc.
• E.g. web crawl data and images may be stored sorted by website. This could correlate with performance
• We might not be capturing sufficient context!
• Standard multi-armed bandit techniques fail
• Solution: only tune using observations from nodes & times with statistically similar data
### Possible Solution

<table>
<thead>
<tr>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agents (core or machine)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Agents (core or machine):**

- Row 1
- Row 2
- Row 3
- Row 4
- Row 5

**Observations:**

- Column 1
- Column 2
- Column 3
- Column 4
- Column 5
### Possible Solution

<table>
<thead>
<tr>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Agents (core or machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

39
**Possible Solution**

Observations

<table>
<thead>
<tr>
<th>Agents (core or machine)</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Use all epochs that pass a statistical similarity test
Possible Solution

Use all epochs that pass a statistical similarity test.
To Lower Overheads

Observations

Store only one ‘aggregated old state’ per epoch
To Lower Overheads

Observations

Store only one ‘aggregated old state’ per epoch

At epoch end: If similar to old, merge into ‘old state’. Otherwise, replace ‘old state’
To Lower Overheads

Observations

Store only one ‘aggregated old state’ per epoch

At epoch end: If similar to old, merge into ‘old state’. Otherwise, replace ‘old state’

Identify (& merge) similar non-local states only at communication rounds, in the centralized model store
Nonstationary Results

Throughput normalized against an ideal oracle that always picks the fastest algorithm
I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion
Regex Operator
Regex Operator

- Tune between four regular expression searching libraries
  - Built-in Java Regex and 3 third-party libraries
Regex Operator

• Tune between four regular expression searching libraries
  • Built-in Java Regex and 3 third-party libraries

• Search through 256k Common Crawl docs (~30gb uncompressed)
  • one tuning round per doc
Regex Operator

• Tune between four regular expression searching libraries
  • Built-in Java Regex and 3 third-party libraries

• Search through 256k Common Crawl docs (~30gb uncompressed)
  • one tuning round per doc

• Test 8 Regexes sourced from regex-sharing website RegExr
  • Match hyperlinks, trigrams, valid emails, color codes, etc.
Regex Operator

• Tune between four regular expression searching libraries
  • Built-in Java Regex and 3 third-party libraries

• Search through 256k Common Crawl docs (~30gb uncompressed)
  • one tuning round per doc

• Test 8 Regexes sourced from regex-sharing website RegExr
  • Match hyperlinks, trigrams, valid emails, color codes, etc.

• Multiple of orders of magnitude variation in performance
  • Email validation regex w/ built-in java utilities takes 33μs to process the fastest document, but over 1000s for the slowest document
Regex Operator

- Tune between four regular expression searching libraries
  - Built-in Java Regex and 3 third-party libraries
- Search through 256k Common Crawl docs (~30gb uncompressed)
  - one tuning round per doc
- Test 8 Regexes sourced from regex-sharing website RegExr
  - Match hyperlinks, trigrams, valid emails, color codes, etc.
- Multiple of orders of magnitude variation in performance
  - Email validation regex w/ built-in java utilities takes 33μs to process the fastest document, but over 1000s for the slowest document
- 8-node (AWS EC2 4-core r3.xlarge) cluster
Note: Y-axis is Log-scale
Distributed Parallel Join Operator
Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
- On each partition, pick a local hash join or a local sort-merge join
Distributed Parallel Join Operator

• Hash-partition relations according to join attributes

• On each partition, pick a local hash join or a local sort-merge join

• Rewards capture total join time
  • measure from when joins begin until result iterators are fully consumed
Distributed Parallel Join Operator

• Hash-partition relations according to join attributes

• On each partition, pick a local hash join or a local sort-merge join

• Rewards capture total join time
  • measure from when joins begin until result iterators are fully consumed

• Set as Spark SQL 2.2’s join for all equijoins too large to broadcast
  • No heuristics and cost models in the query optimizer, falls back on explicit configurations (defaults to global sort-merge join)
Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
- On each partition, pick a local hash join or a local sort-merge join
- Rewards capture total join time
  - measure from when joins begin until result iterators are fully consumed
- Set as Spark SQL 2.2’s join for all equijoins too large to broadcast
  - No heuristics and cost models in the query optimizer, falls back on explicit configurations (defaults to global sort-merge join)
- Test on TPC-DS benchmark (scale factor 200)
Distributed Parallel Join Operator

• Hash-partition relations according to join attributes

• On each partition, pick a local hash join or a local sort-merge join

• Rewards capture total join time
  • measure from when joins begin until result iterators are fully consumed

• Set as Spark SQL 2.2’s join for all equijoins too large to broadcast
  • No heuristics and cost models in the query optimizer, falls back on explicit configurations (defaults to global sort-merge join)

• Test on TPC-DS benchmark (scale factor 200)

• Configure queries to use 512 shuffle / join partitions
Join Results (Query Throughput)
Join Results (Query Throughput)

Cuttlefish join usually faster or very comparable
(Join throughput graphs even more dramatic)
Join Results (Query Throughput)

Cuttlefish join usually faster or very comparable (Join throughput graphs even more dramatic)

But, requires exploration & doesn’t always provide ‘special ordering’ benefits
I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion
**Cuttlefish**

uwdb.io/projects/cuttlefish

- A simple, flexible API for online tuning
- Thompson-sampling based tuning algorithms
- Supports contextual tuning (learns cost models)
- Distributed learning between workers
- Adapts to nonstationary workloads
- Prototyped in Apache Spark & successfully tunes convolution, regex, and join operators