CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics

Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, Ming Zhang
Recurring jobs are popular

Microsoft reports 40% of key jobs at Bing are rerun periodically.
<table>
<thead>
<tr>
<th>Providers</th>
<th>Machine Type</th>
<th>Cluster Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>r3.8xlarge, i2.8xlarge, m4.8xlarge, c4.8xlarge, ...</td>
<td>+ configurable VMs</td>
</tr>
<tr>
<td>Azure</td>
<td>A12, D1, D2, D3, L4s, ...</td>
<td>n1-standard-4, n1-highmem-2, n1-highcpu-4, f1-micro, ...</td>
</tr>
</tbody>
</table>

Hundreds of instance types and instance count combinations
Choosing a good configuration is important

Better performance:
- For the same cost: best/worst running time is up to 3x
  - Worst case has good CPUs whereas the memory is bottlenecked.

Lower cost:
- For the same performance: best/worst cost is up to 12x
  - No need for expensive dedicated disks in the worst config.
  - 2$ vs. 24$ per job with 100s of monthly runs
How to find the best cloud configuration
One that minimizes the cost given a performance constraint
for a recurring job, given its representative workload?
High Accuracy
Close the optimal configurations

Adaptivity
Works across all big-data apps

Key Challenges

Low Overhead
Only runs a few configuration

Modeling
Searching
Existing solution: searching

- Systematically search each dimension (Coordinate descent)
  - On each resources: RAM, CPU, disk, cluster sizes

- Problem: not accurate
  - Non-convex performance/cost curves across many resources
  - If you search one dimension, drop early, it would mislead you later
Existing solution: modeling

- Modeling the resource-perf/cost tradeoffs
  - Ernest [NSDI 16] models machine learning apps for each machine type

- Problem: not adaptive
  - Frameworks (e.g., MapReduce or Spark)
  - Applications (e.g., machine learning or database)
  - Machine type (e.g., memory vs CPU intensive)
High Accuracy

High Accuracy

Adaptivity

Low Overhead

Exhaustive

High overhead

CherryPick

Modeling
Ernest
Not general

Searching
Coord. Descent
Not accurate

Adaptivity
Key idea of CherryPick

- **Adaptivity: black-box modeling**
  - Without knowing the structure of each application

- **Accuracy: modeling for ranking configurations**
  - No need to be accurate everywhere

- **Low overhead: interactive searching**
  - Smartly select next run based on existing runs
Workflow of CherryPick

Start with any config

1. Run the config
2. Black-box Modeling
3. Rank and choose next config
4. Return config

Interactive search
**Workflow of CherryPick**

1. Run the config
2. Black-box Modeling
3. Rank and choose next config
4. Return config

Interactive search
Workflow of CherryPick

1. Run the config
2. Black-box Modeling
3. Rank and choose next config
4. Return config

Interactive search
Workflow of CherryPick

1. Run the config
2. Black-box Modeling
3. Rank and choose next config
4. Return config

Interactive search
Workflow of CherryPick

Run the config → Black-box Modeling → Rank and choose next config → Return config

Interactive search
Workflow of CherryPick

Bayesian Optimization

Run the config ➔ Black-box Modeling ➔ Rank and choose next config ➔ Return config

Interactive search
Workflow of CherryPick

1. Prior function
2. Run the config
3. Black-box Modeling
4. Rank and choose next config
5. Return config

Bayesian Optimization

Interactive search
Workflow of CherryPick

Bayesian Optimization

Run the config → Black-box Modeling → Rank and choose next config → Return config

Prior function
Interactive search
Acquisition function
Workflow of CherryPick

Bayesian Optimization

Run the config → Black-box Modeling → Rank and choose next config → Return config

Prior function → Interactive search
Prior function for black box modeling

This is the actual cost function curve across all configurations.

Feature vector:
- CPU
- RAM
- Machine count
- ...
Challenge: what can we infer about configurations with two runs?
Challenge: what can we infer about configurations with two runs? There are many valid functions passing through two points
Challenge: what can we infer about configurations with two runs?
Solution: confidence intervals capture where the function likely lies
Workflow of CherryPick

- Run the config
- Black-box Modeling
- Rank and choose next config
- Return config

Prior function
Interactive search
Acquisition function
Acquisition function for choosing the next config

Build an acquisition function based on the prior function
Calculate the expected improvement in comparison to the current best configuration
Acquisition function for choosing the next config
Acquisition function for choosing the next config
Iterative searching
CherryPick searches in the area that matters

A database application with 66 configurations
CherryPick searches in the area that matters

A database application with 66 configurations
CherryPick searches in the area that matters

A database application with 66 configurations
Noises in the cloud

- Noises are common in the cloud
  - Shared environment means inherent noise from other tenants
  - Even more noisy under failures, stragglers
  - Strawman solution: run multiple times, but high overhead.

- Bayesian optimization is good at handling additive noise
  - Merge the noise in the confidence interval

\[ f(\bar{X}) + \epsilon \]

(learned by monitoring or historical data)
Challenge: Multiplicative noise

A lot of the noise in cloud is multiplicative

- Example: if VMs IO slows down due to overloading
  - Writing 1G to disk (5 sec normally) now takes 6 secs (by 20%)
  - Writing 10G to disk (50 sec normally) now takes 60 secs (by 20%)

\[
\hat{C}(\vec{X}) = C(\vec{X}) \times (1 + \epsilon)
\]

The max. relative variance in a given cloud.

Use the log function to change to additive noise

\[
\log \hat{C}(\vec{X}) = \log C(\vec{X}) + \log(1 + \epsilon)
\]
Further customizations

- **Discretize features:**
  - Deal with infeasible configs and large searching space
  - Discretize the feature space

- **Stopping condition:**
  - Trade-off between accuracy and searching cost
  - Use the acquisition function knob

- **Starting condition:**
  - Should fully cover the whole space
  - Use quasi-random search
Evaluation settings

5 big-data benchmarks
- Database:
  - TPC-DS
  - TPC-H
- MapReduce:
  - TeraSort
- Machine Learning:
  - SparkML Regression
  - SparkML Kmeans

66 cloud configurations
- 30 GB–854 GB RAM
- 12–112 cores
- 5 machine types
Evaluation Settings

**Metrics**
- Accuracy: running cost compared to the optimal configuration
- Overhead: searching cost of suggesting a configuration

**Comparing with**
- Searching: random search, coordinate descent
- Modeling: Ernest [NSDI’16]
CherryPick has high accuracy with low overhead

CherryPick

Coordinate search

Coord. search has 20% more searching cost in median and 78% higher running cost in the tail for TPC-DS

Not accurate

Not adaptable

3.7 times searching cost

ERNEST

Random search

Coordinate search

350
200
250
300
400

TPC-DS

Running cost normalized by opt. Config. (%)

Searching cost normalized by CherryPick’s median (%)
CherryPick corrects Amazon guidelines

- **Machine type**
  - AWS suggests R3, I2, M4 as good options for running TPC-DS
  - CherryPick found that at our scale C4 is the best.

- **Cluster size**
  - AWS has no suggestion
  - Choosing a bad cluster size can have 3 times higher cost than choosing a right cluster config.

- Combining the two becomes even harder.
Conclusions

Adaptivity

Black-box modeling:
Requires running the cloud configuration.

Low overhead

Restricted amount of information:
Only a few runs available.

High accuracy

“... do not solve a more general problem ... try to get the answer that you really need”
—Vladimir Vapnik
Thanks!

Please try our tool at:

https://github.com/yale-cns/cherrypick