Fast Approximations for Analyzing Ten Trillion Cells

Filip Buruiana (filipb@google.com)

Reimar Hofmann (reimar.hofmann@hs-karlsruhe.de)
Outline of the Talk

- Interactive analysis at AdSpam @ Google
- Trade off accuracy for speed
- Sampling - challenges, accuracy, results
- Data sketch inspired approach for reducing memory usage
AdSpam Team @ Google

● Click police: filter “invalid clicks” & charge only for organic traffic

● Typical data sets: billions of records

● Typical use-cases:
  ○ Ops - manually review suspicious clicks
  ○ Eng - research new filter ideas
PowerDrill: Workflows

- one UI interaction $\Rightarrow$ multiple SQL queries

- data discovery, finding patterns $\Rightarrow$ many interactions

* Displayed data is from PowerDrill usage logs
Interactive: Motivation

weekly latency, overall (in seconds)

Waiting for results:
• costs money
• users get bored

red: 95%-quantile
blue: average
Trade off accuracy for speed

- Some workflows: exact (e.g. reporting)

- Others: approx is fine
  - Analyze longer-time trends
  - “Good enough” intermediate results ⇒ quick decisions
  - Hints for the user: what to explore next
UI features enabled by faster queries

- Histogram in **field info box**

* Schema is from PowerDrill usage logs
UI features enabled by faster queries

- **Cross chart highlighting**

  Hovering over a value in a chart marks positively correlated values in other charts

* Displayed data is from PowerDrill usage logs
UI features enabled by faster queries

- Cross chart highlighting

* Displayed data is from PowerDrill usage logs
Sampling

Memory reduction with maximum query flexibility:

- Particularities of our system
- Predicting accuracy
- Performance results
Hierarchical relational data

PowerDrill contains hierarchical data

Sampling happens on the top level
→ Consider when estimating sampling accuracy of lower level fields
→ Consider when designing data model for new data
Sampling: Dealing with Inaccuracy

“Accurate” → $P(\text{relative error} < 10%) > 90$

“Inaccurate” → greyed out

Accuracy depends on
● the number of records supporting the estimate, and
● the variance of the data column.

* Displayed data is from PowerDrill usage logs
Sampling: Predicting Accuracy

Evaluation on real queries (10% sample):
- avoid computing sample variance $\rightarrow$ heuristic

<table>
<thead>
<tr>
<th></th>
<th>Accuracy Prediction based on</th>
<th>True accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sample Variance</td>
<td></td>
</tr>
<tr>
<td>classified as ...</td>
<td>correctly</td>
<td></td>
</tr>
<tr>
<td>... “accurate”</td>
<td>67%</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42%</td>
</tr>
<tr>
<td>... “inaccurate”</td>
<td>33%</td>
<td>61%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>58%</td>
</tr>
<tr>
<td></td>
<td>classified as ...</td>
<td>correctly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>75%</td>
</tr>
</tbody>
</table>

based on 30 K cells of queries from the query log
Sampling: Success Metrics

**weekly latency, overall** (in seconds)

- **red**: 95%-quantile
- **blue**: average

**weekly latency, sampled queries** (17% of overall)

- **red**: 95%-quantile
- **blue**: average
Bottleneck: Disk loads

- Low latency variance for sampled: better cache hits (# items on disk: 27% for sampled vs. 34% non-sampled)

- Data >> available RAM ⇒ memory + disk

- Disk loads = high latency
Expensive fields

- 2% of overall queries: 70% of all memory ⇒ affects all queries

- Top-K queries with expensive fields

```
SELECT SearchQuery as Query, Count() as Count
FROM Table
GROUP BY Query
ORDER BY Count
LIMIT 10
```

many unique long string values ⇒ 450 MB memory / server

- 10% of memory caches for only one field!
Execution of Top-K Queries

Example: \( K = 1, C = 3 \)

<table>
<thead>
<tr>
<th>Search Query</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats</td>
<td>100</td>
</tr>
<tr>
<td>dogs</td>
<td>90</td>
</tr>
<tr>
<td>opossum</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search Query</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>dogs</td>
<td>320</td>
</tr>
<tr>
<td>cats</td>
<td>300</td>
</tr>
<tr>
<td>opossum</td>
<td>5</td>
</tr>
<tr>
<td>trout</td>
<td>1</td>
</tr>
</tbody>
</table>

\( C^*K \) string lookups
String values ⇒ IDs

Example: K = 1, C = 3

Top C*K SearchQueryID

<table>
<thead>
<tr>
<th>Search QueryID</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>100</td>
</tr>
<tr>
<td>609</td>
<td>90</td>
</tr>
<tr>
<td>321</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search QueryID</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>609</td>
<td>140</td>
</tr>
<tr>
<td>1023</td>
<td>100</td>
</tr>
<tr>
<td>789</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search QueryID</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>100</td>
</tr>
<tr>
<td>609</td>
<td>90</td>
</tr>
<tr>
<td>321</td>
<td>3</td>
</tr>
</tbody>
</table>

0 string lookups

K string lookups
Reverse Lookup Service

Globally consistent IDs: hard
ID space needs to be used efficiently

Search Query ID | Search Query
--- | ---
321 | opossum
609 | dogs
789 | trout
1023 | cats

Billions of distinct values
Most frequent items: memory
The rest: distributed file system

Reverse lookup dictionary
Global IDs = Hashes

Use predefined hash function

- globally consistent
- controllable size of IDs
- collisions can lead to ambiguous, “lifted” results

<table>
<thead>
<tr>
<th>SearchQuery</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘cats’ or ‘opossums’</td>
<td>900</td>
</tr>
<tr>
<td>‘dogs’</td>
<td>899</td>
</tr>
</tbody>
</table>
Collision Probability

Example:

some fields → D ≈ 3 bn distinct values
hash length 32 bits → N ≈ 4 bn distinct IDs
(more: implementation penalty)

⇒ Can’t afford to keep collisions rare

\[ 1 - \left(1 - \frac{1}{N}\right)^D \approx 50\% \text{ collision probability} \]
Background: The Count-Min-Sketch

Use multiple redundant hash tables:

- Results accurate unless item collides in both hash tables
- Well known error bounds
- Known tradeoff between number of hashes and hash size.

```
<table>
<thead>
<tr>
<th>count</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>900</td>
<td>cats</td>
<td>min</td>
</tr>
<tr>
<td>899</td>
<td>opossum</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>trout</td>
<td>919</td>
</tr>
</tbody>
</table>
```

\[ N = 2^{32} \]
## CM-Sketch-inspired approach

**Idea:**
Two data columns with two different hashes of `SearchQuery`

<table>
<thead>
<tr>
<th>Count min sketch</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash function as index into (dense) tables.</td>
<td>Hash function as encoding for a data column</td>
</tr>
<tr>
<td>WHERE-conditions can not be applied once table is filled</td>
<td>WHERE-conditions from other fields can be applied</td>
</tr>
</tbody>
</table>

- Error bounds of CM-sketches apply
- Collisions can be resolved
- Optimal hash sizes are different

But: Redundancy wastes memory.
CM-Sketch-inspired approach

- Half of the data is encoded with hash_1, other half with hash_2.
- No memory is wasted.
- Most collisions can be resolved approximately:

<table>
<thead>
<tr>
<th>hash_1 count</th>
<th>450</th>
<th>449</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opossum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dogs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trout</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hash_2 count</th>
</tr>
</thead>
<tbody>
<tr>
<td>435</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>455</td>
</tr>
</tbody>
</table>

‘Cat’ ↔ ‘opossum’:
\[ \text{count(‘cat’)} \approx 450 + 435 - 10 \]
\[ \text{count(‘opossum’)} \approx 2 \cdot 10 \]
Accuracy

- At least as good as CM-sketch with one hash table.
  Example: (31 bit hash for field SearchQuery)

\[ \text{Lift} := \text{Count}_{\text{est}} - \text{Count}_{\text{true}} \]

\[ E(\text{Lift}) = 7, \ P(\text{Lift} > 70) < 10\%, \ P(\text{Lift} > 700) < 1\% \]

\[ \text{Count} > 70000 \text{ for all TOP 100 SearchQueries} \]

\[ \Rightarrow P(\text{relative error} > 1\%) < 1\% \]

- Similarly low error rates for other fields and queries.

- Benefit of splitting data:
  Resolve ambiguities in reverse lookup.
Memory Benefits

**SearchQuery:** 325 GB → 90 GB (3.6x reduction)

**other field:** 95 GB → 60 GB (1.6x reduction)

⇒ Store ~ 3x more items in memory
⇒ Speed up all queries
Conclusion

• Approximations help to get faster / simpler
• Enable a new level of usability concepts
• Accuracy can be predicted