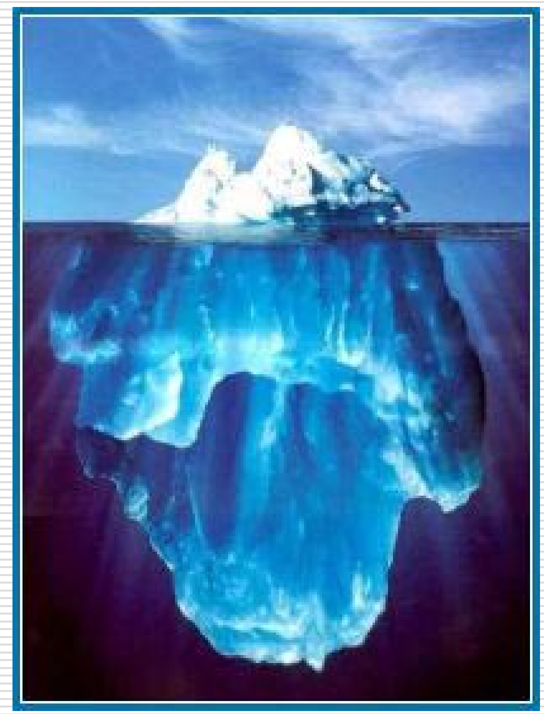
Computing Iceberg Queries Efficiently -

A summary of the paper by Fang, et al.

Noah Clemons
Data Mining
University of Chicago



Aggregate Function

Definition: a function that performs computation on a *set* of values rather than on a single value

- Examples: COUNT, SUM, AVG, etc.
- Suppose we would like to eliminate aggregate values below some threshold. Sounds simple enough?
- Table 1 – consider the relation
- ```
  SELECT      target1, target2, count(rest)
  FROM        R
  GROUP BY    target1, target2
  HAVING      count(rest) >= T
  ```
- If T = 3, we get the tuple <a,e,3>.
- **ICEBERG QUERY** – The relation R and the number of unique target values are typically huge (the iceberg), and the answer is very small
- Picture 1 - demonstration
  ```

Why are Iceberg Queries a concern? The amount of targets can get very large and must be computed efficiently!

"The *time* to execute iceberg queries dominates the cost of producing interesting association rules" – Park et al. ACM SIGMOD, May '95

- What are some good ideas for efficiency?
- Use compact, in memory data structures. **A nice start. But how?**
- Maintain array of counters in main memory, one counter for each unique target set -> answer query in single pass over data. **Insufficient – R usually X times larger than available memory.**
- Sort R on disk, then do a pass over it aggregating and selecting targets above the threshold. **Fine if you are willing to wait weeks (Gigabytes) or months (Terabytes).**
- Unfair assumption – R is materialized. With the newest Wal-Mart data, R could easily be too large to be explicitly materialized.

  Example – finding word pairs in 100,000 web documents, avg word length 118 words. Original storage, 500 MB.

  R over which the iceberg query is to be executed has all pairs of words that occur in the same document. New storage, 40 GB. **Answer size: 1MB!**

Key: Avoid sorting or hashing realized or unrealized R by keeping compact, in-memory structures that allow use to identify **threshold targets.**

Solution: Extend sampling and multiple hash function algorithms to improve performance and reduce memory requirements.

- Why work with these new algorithms? Sounds like a lot of work
- Examples why executing Iceberg Queries with efficient care is important.
- Example 1 – try the most current sorting technique. Large response time to query. Need to make algorithm that performs different amount of work *depending on size of query's output*
- What if we change criteria for selecting item to be $10?
- Key point to keep in mind throughouth the paper: traditional techniques lead to unacceptable turnaround times and disk space. Their new algorithms make large improvements.

Terminology for the Algorithms - Assumptions

Beginning with Relation R with <target,rest> pairs.
Executing simple iceberg query with groups on the single target in R

- **V** – ordered list of targets in R such that V[r] is r^{th} most frequent target in R (r^{th} highest rank), **n** = |V|
- **Freq(r)** – the frequency of V[r] in R
- **Area(r)** – total number of tuples in R with r most frequent targets
- Typical Iceberg Query: select target values with frequency higher than threshold **T**
- $\mathbf{r_t}$ = max{r|Freq(r)>=T} gives: *as an answer to the query*
  H = {V[1], V[2], … , V[r_t]} ----- HEAVY TARGETS
  L = the remaining *light* values

a priori flaws in their algorithms? You decide.

- All their algorithms compute set of *potentially* heavy targets **F** that contains as many members of **H** as possible.

- F – H != { } -> false positive (light values reported as heavy).

  Solution: use procedure *Count(F)…*

  Scan and explictly count frequency of targets in F. Only targets that occur T or more times are output in the final answer.

- H – F ! = { } -> false negatives (heavy targets are missed). Very dangerous – post processing to “regain” false negative inefficient.

  Regain false negatives efficiently in *high skew* case – example: very small fraction of targets account for 80% of tuples in R, while other targets together count for the other 20%.

Simple Algorithms to compute **F** – building blocks for the more sophisticated algorithms

SCALED-SAMPLING

- Take random sample of size *s* from **R**.
- If count of each target in the sample, scaled by *N/S*, exceeds the specified threshold, target is part of the candidate set **F**.
- PROS: Simple and efficient to run. ☺
- CONS: We obtain *both* false-positives and false-negatives. Removing the shear amount of them is difficult. ☹

Simple Algorithms to compute **F** – building blocks for the more sophisticated algorithms

COARSE-COUNT (Probablistic Counting)

- Intuition: Use an array A[1..*m*] of *m* counters and a hash function h_1 which maps target values from $\log_2 n$ bits to $\log_2 m$ bits, m << n
- Intialize all *m* entries of A to zero. Perform a linear scan of **R**.
- For each tuple in **R** with target *v*, $A[h_1(v)]$++ : THE HASHING SCAN (HS)
- Compute bitmap array $BITMAP_1[1..m]$ by scanning through array A
  - If bucket *i* is heavy (i.e. $A[i] \geq T$) then set $BITMAP_1[i]$

Reclaim memory allocated to A.
Compute F by performing *candidate-selection* scan of **R**:

- Scan **R** and for each targer *v* whose $BITMAP_1[h_1(v)] = 1$, add *v* to **F**. Remove false-positives by executing *Count*(**F**).

Pros: NO FALSE NEGATIVES

HYBRID techniques – combine sampling and counting approaches for a better algorithm.

- Intuition – sample data to identify candidates for heavy targets, then use coarse-counting to remove false-negatives and false-positives.
- PROS: reduce number of targets that fall into heavy buckets – i.e. fewere light targets becoming false positives.
- Cons: more difficult to implement. You must make prudent choices on which algorithm to spend your time on depending on your **R.**
- In other words, you need to know a lot about the composition of your data *a priori.*

DEFER-COUNT

Intuition: Find a way to get fewer heavy bucket and therefore get fewer false positives

- First compute a small sample (size $s << n$) of the data using the sampling techniques mentioned.
- Select the f , $f < s$, most frequent targets in the sample and add them to **F**.
- Remove false positives by executing *Count*(**F**)
- CONS: splits up main memory b/t samples set and buckets for counting.

It's also hard to choose good values for s and f that are useful for a variety of data sets. Overhead is also high.

MULTI-LEVEL

Intuition: Do not explicitly maintain list of potentially heavy targets in MM. Use the sampling phase to identify potentially heavy *buckets.*

- Peform sampling scan of data. For each target v, increment $A[h(v)]$ for the hash function h.
- After sampling s targets, consider each of the A buckets.
  - If $A[i] > T \times s/n$, mark the i^{th} bucket to be potentially heavy.
  - For each i^{th} bucket allocate m_2 auxiliary buckets in MM

NOW reset all counters in A array to zero. Perform hashing scan of data.

For each target v in the data, increment $A[h(v)]$ if bucket corresponding to $h(v)$ is **not** marked to be potentially heavy. If the bucket is marked, apply new hash function $h_2(v)$ and increment corresponding auxiliary bucket

MULTI-STAGE

Intuition: use available memory more efficiently.

- Do the same pre-sampling phase as MULTI-LEVEL. (Identify heavy buckets)
- But allocate a common pool of auxiliary buckets $B[1,2,\ldots,m_3]$.
- Perform hashing scan of the data:
  - For each target v in the data, increment $A[h(v)]$ if the bucket corresponding to $h(v)$ is **not** market as potentially heavy.
  - If bucket is marked, apply the second hash function h_2 and increment $B[h_2(v)]$.
- Disadvantage: determining how to split memory across primary buckets and auxiliary buckets can only be determined empirically ☹

MULTIBUCKET algorithms

Optimize the HYBRID algorithms and alleviate flaws

- Flaws in HYBRID algorithms:
  - Many false-positives if many light values fall into buckets with
    - Problem 1: One or more heavy targets
    - Problem 2: Many light values

Sampling helps problem 1, but heavy targets not identified by Sampling could lead to several light values falling into heavy buckets.

HYBRID cannot avoid problem 2.

Solution: use multiple sets of primary and auxiliary buckets

PROS: reduces # of false positives significantly

CONS: complicated to implement, complicated to describe

MULTIBUCKET ALGORITHMS

- Single Scan DEFER-COUNT with multiple hash functions (UNISCAN)
- Multiple Scan DEFER-COUNT with multiple hash functions (MULTISCAN)
- MULTISCAN with shared bitmaps (MULTISCAN-SHARED)

Interesting and useful extension to the algorithms – a SUM query

- PopularItem Query
- Example 2

Rules of Thumb

- MULTI-LEVEL did not perform well in their experiments
- DEFER-COUNT works well when one expects the data to be very skewed (i.e. *high skew* 80/20 case: very few targets are heavy, but constitute most of the relation). Be sure to use a small *f* set
- MULTI-STAGE works well if data is not skewed, since it does not incur the overhead of looking up the values in *f*.
- If data distribution is flat, don't use a sampling scan.
- If you want to get crazy and use MULTIBUCKET algorithms, take a look at their long technical report and take the plunge