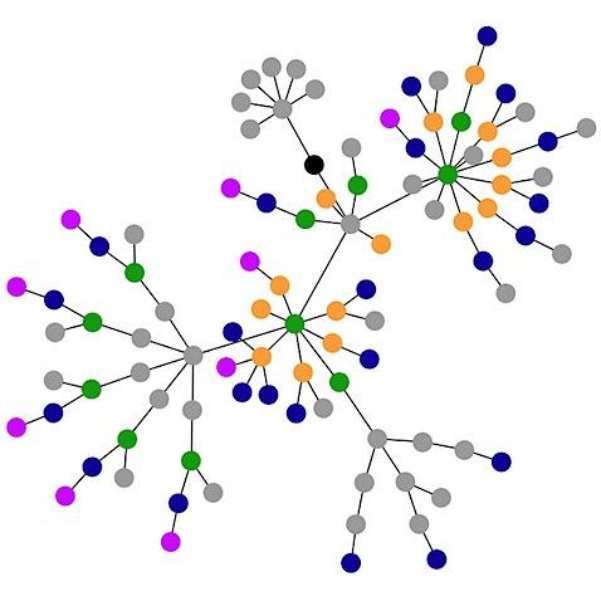
Adaptive Compression for Graph Processing

Rob Moore

Graphs

- Vertices and edges
  - People and relationships
  - Sites and links
- Queries
  - Trends
  - Relative importance



How can we model a graph on a computer?

- Definition: G = (V, E)
  - A set of vertices with their associated data, and a set of edges with their associated data.
- Edges
  - src -> dst
- Vertex data
  - (V -> a,b,c)

Improving on an edge array

- Cluster edge array by vertex on one side

Example:

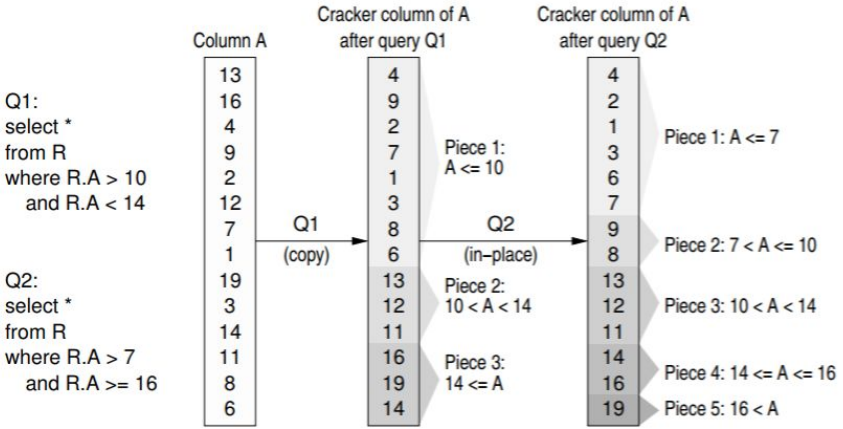
- [1 -> 2, 3, 4; 2 -> 1; 3 -> 1, 2] (clustering by src node)
  - 1 has outward edges to 2, 3, 4
  - 2 has an outward edge to 1
  - 3 has outward edges to 1, 2
  - 4 has no outward edges

Going faster

- Unpredictable workload: New trends and news stories are popping up all the time, triggering different queries.
- Option 1: Build indices beforehand, you'll just have to guess which ones you'll need.
    - Offline indexing
- Option 2: Build indices when you've decided you need them: But you'll still have to wait while they're getting built.
    - Online indexing

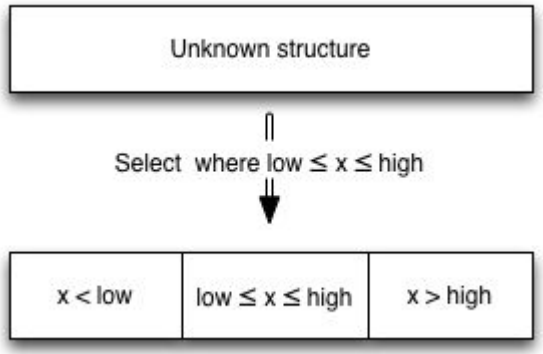
Adaptive Indexing and Database Cracking

- Cracking was invented for relational databases.
- Workload aware auto-tuning
- Basic idea: Queries partition the column around the given pivots and the boundaries of the resulting column fragments are stored in the cracker index.



Cracking Algorithm

- Base column copied into cracker column
- Scanning region defined
- Edge pointers fully tightened, cracker column restructured in-place
- Column fragments created
- Fragment boundaries saved in cracker index



$$(\texttt{value},\ \texttt{index})\ \epsilon\ \texttt{CrkIdx} \implies \forall i < \texttt{index} : crk[i] < \texttt{value}$$

Cracking an edge array

- Two arrays, *src* and *dst* representing the edge array
- Queries are for single node identifiers
- The edge array is adaptively clustered by cracking

Cracking an edge array

- Two arrays, *src* and *dst* representing the edge array
- Queries are for single node identifiers
- The edge array is adaptively clustered by cracking

Edge array -> Adjacency list

= Compression!

Challenge: Cracking + Compression

Opportunities: When can we compress?

- When we know a contiguous section of column to be uniform
- At varying granularity
  - Coarse: Per-fragment compression
  - Fine: Run-length encoding

Exploitation

- Basic idea: Do not scan sections of column known to be uniform.
- Per-fragment: Cost of repeated query = Cost of lookup in cracker index
- Run length encoding: Scans over any previously touched and encoded elements are faster.

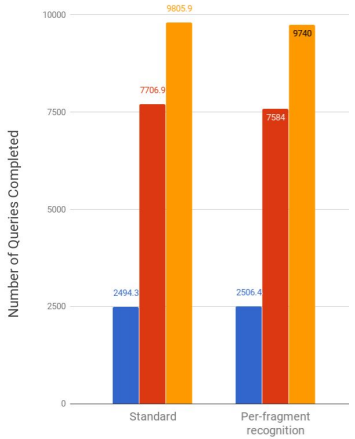
Per-fragment Compression

Recognising uniform fragments

- Cracker index updates
- Minimally different values
  - 3 and 4 are minimally different integers
  - 5.5 and 5.6 are minimally different decimals to 1dp
  - 7 and 9 are minimally different odd numbers
- Fragment is known to be uniform iff two minimally different values are in the cracker index.
- Repeated queries involve do no scanning at all, and some other queries too

Recognising uniform fragments: Performance

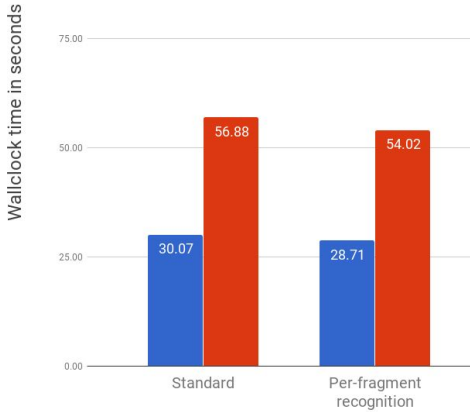
- Only difference: Check for uniformity of the fragment immediately after retrieving the values of the edge pointers from the cracker index.
- Cost: One branch per query
- Benchmark: Break even point versus sorting



All: No repeated queries
Blue: 100K nodes
Red: 500K nodes
Yellow: 1M nodes

Recognising uniform fragments: Performance

- Only difference: Check for uniformity of the fragment immediately after retrieving the values of the edge pointers from the cracker index.
- Benefit: Less scanning
- Benchmark: Pagerank



Both: ~1.9M edges
Blue: 50 iterations
Red: 100 iterations

Run-length encoding

- Basic idea: 3 3 3 3 3 (five threes) represented as 5 3 (five threes).
- Less scanning - hopping over duplicate values when scanning a previously touched section.
- Harder scanning - building runs

Challenge: Build & maintain RLE while cracking

- Information stored in *run_length* array, which contains number of elements that can be skipped by a scanning pointer arriving at that element.
- Runs marked on both sides
- Fragments are built as a big run, so we get the benefits of per-fragment compression.
- Integrity of run length markings must be preserved when swapping.
- Two implementations which perform very similarly

RLE Cracking: Performance

- Initial situation:
  - No runs are marked
  - Many will require marking
  - Consequence: Early scans will be slower
- Later scans:
  - Many runs are marked
  - Will not have much marking to do
  - Consequence: Later scans will be faster

Cracking is a lightweight operation

- Immediate adaptivity to the workload
- Prompt query responses, low overhead

RLE cracking is too heavyweight!

- We found the the early costs for RLE to be prohibitive.
- Less than 40 answered queries in the time taken to answer 10,000 with standard cracking

What did we buy with all that work?

- Time to complete a number of pagerank iterations no slower or faster than standard cracking
- Once a sufficient amount of work has been done building runs, it catches up!

What did we buy with all that work?

- Time to complete a number of pagerank iterations no slower or faster than standard cracking
- Once a sufficient amount of work has been done building runs, it catches up!

Worth it?

What did we buy with all that work?

- Time to complete a number of pagerank iterations no slower or faster than standard cracking
- Once a sufficient amount of work has been done building runs, it catches up!

Worth it?

- No overall speed boost
- Not aligned with the goals of adaptive indexing

How could we have made RLE cracking better?

- Complex branching => Predication
  - Used effectively for standard cracking
- Use cutoff parameter for building runs
- Simpler algorithms with less branching, perhaps with less maintenance during swapping
- The underlying complexity is the same as standard cracking - it’s still a scan

Summary

Per-fragment vs RLE

- Per-fragment compression maintains the lightweight property of standard cracking, RLE doesn't.
- RLE improves scans over previously scans sections, not just repeat queries.

Per-fragment vs RLE

- Per-fragment compression maintains the lightweight property of standard cracking, RLE doesn't.
- RLE improves scans over previously scans sections, not just repeat queries.

Overall:

- Per-fragment performs much better

Standard Cracking vs Per-fragment

- Standard cracking is more lightweight, but not by much
- Per-fragment is faster over many queries, but not by much

Standard Cracking vs Per-fragment

- Standard cracking is more lightweight, but not by much
- Per-fragment is faster over many queries, but not by much

Overall:

- Small, but non-trivial trade-off
- Best case would be to combine them.

Summary of Achievements

- Explored the space of adaptive compression using database cracking
- Created per-fragment compressive cracking and showed how it outperformed standard cracking on overall speed.
- Demonstrated two implementations of RLE cracking and compared its properties to standard cracking.

Future work

- Workload-robust RLE cracking
- CPU efficient and parallel implementations

In the report but not detailed here

- Compaction
- RLE: Underswapping vs Overswapping

Ask me about it