Adaptive graph processing

Interim Project Report

Rob Moore (rrm115)
Holger Pirk (pirk)

Version 9

1. Introduction

Connected data is increasingly being leveraged to produce real world value, as we can see from the increasing dominance of companies whose business model is centered around querying connected data - Google/Facebook et. al. As these datasets grow larger, faster methods for mining graph data will be required to continue to increase the value that can be delivered to users.

We are looking to propose a novel method of processing connected data which performs online restructuring and compression to improve query performance. We are currently pursuing the application of existing adaptive processing techniques to graph processing problems, as well as exploring the possibility of adaptively building existing graph indices.

2. Background and Related Work

Workload-aware frameworks

I will here briefly mention two of the most relevant workload-aware frameworks that have been researched.

Aluc et. al. proposed the group-by-query (G-by-Q) representation for RDF data, which uses an adaptively built cluster-index on edge-disjoint partitions of the connected data being represented. This index is adaptively maintained and the stored data can be restructured arbitrarily to optimise for the current workload, hence the name of the implementation: chameleon-db.

G-by-Q proved to be fast and robust in experiments against other popular frameworks for querying RDF data: MonetDB, 4store and both column and row store versions of Virtuoso Open Source. However, the framework is complicated - the authors' implementation was reported to be over 35,000 lines of C++. In this work we are aiming to produce simpler contributions.

One of the inspirations for G-by-Q and this work too, was database cracking, a technique for relational databases proposed by Idreos et. al., which disjointly partitions relevant pieces of a copy of a column by scanning them, maintaining information about the pieces to identify the relevant tuples for a given query. In this work, we apply cracking to perform adaptive clustering on adjacency lists as a baseline against which to test our final contributions, and to see if through optimisations such as adaptive compression, this can indeed be perhaps among our contributions. An illustration of the method is seen in figure 2.1.

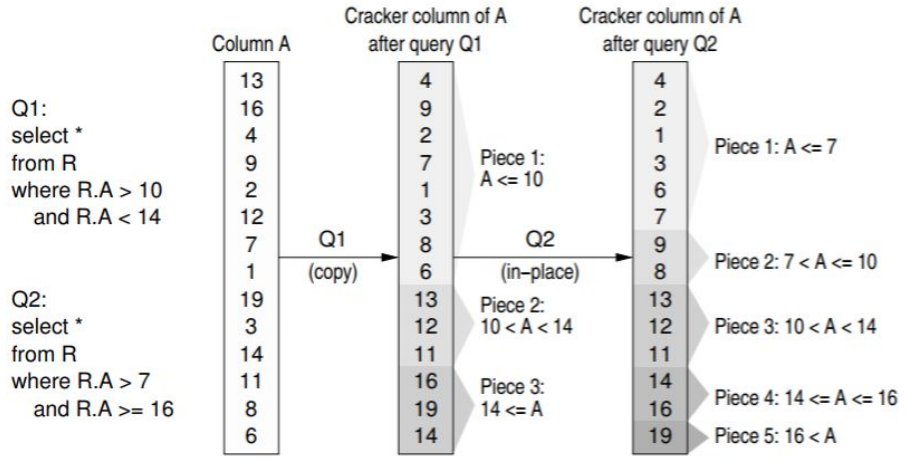

Figure 2.1. Illustration of cracking, taken from Idreos' paper

Graph database indexing

There has also been much study in the building of graph database indices to improve performance. We intend in this work to select an effective index and develop online algorithms to build it, with a view to gain the benefits of an index, without incurring the overhead cost of building it upfront.

GRIPP is a graph database index by Leser et. al. based on utilising pre- and post-order numbering of nodes in a depth-first search tree in order to answer distance and reachability queries. Building this index is efficient in space and time relative to other options (such as computing the transitive closure). In reachability experiments, GRIPP between just 2 and 4 times slower than the transitive closure, which is obviously very fast, however, for experiments on distance queries, GRIPP is only faster than (non-clustering) breadth-first search for graphs under 20,000 nodes. In this work we are going to be ambitious and aim to produce a framework capable to handling traversal queries such as breadth-first search, on large graphs, which may mean we don't use GRIPP or that the scope of this work changes - this decision is not yet final.

SPath is a general purpose index for graph databases, where queries are modelled as graph patterns, based on the fact that a given graph query will touch a vertex and some pattern of neighbours and edges, that is, the query is representable as a (probably small) graph. Zhao et. al. propose an index which takes a single parameter, the neighbourhood scope, and compute for every vertex the number of distinct shortest paths out of it with length less than or equal to the neighbourhood scope hyperparameter. This neighbourhood signature, as it is called, is used to characterise all a graph's subgraphs. This index is used to prune the number of candidate graphs that must be matched against by the vertices of the query graph.

This index is faster to build than GRIPP and is more general purpose, however its implementation is more complex. It also lends itself to adaptive building, because any k-depth breadth-first-searches you compute can be used to partially build the index, which can be then used to prune future queries. We have not yet looked into this.

Index-free adjacency

This storage technique, rather than storing nodes and edges in adjacency tables, stores all relationships as pointers contained within records. This avoids the need to do any expensive recursive self joining when querying the data structure, but it is also more difficult to exploit locality when querying connected data stored in this way. We have no plans to explore it in this project so far, but it is not excluded from investigation.

3. Plan

The bulk of the time available to work on this project will inevitably lie after March exams i.e. during Easter and during Summer term. In anticipation of exam focus, not much work is planned for the remaining time before Easter.

During/after stints of work on a problem, challenges faced and overcome, such as unexpected edge cases and significant implementation details have been logged in a journal in order to use as material for the final report. The entries will help to provide many details in what will become the design, implementation and optimisation sections of the final report.

Compressive Cracking

- Eager: Run length encode the cracker column during scans of it to build an index. Possibly combine this with the cracker index for further gains.
- Lazy: No compacting of the columns data. Store offsets within the cracker column back into the original column representing compressed pieces. Potentially combine with SIMD operations to achieve high-speed recursive self-joins for out-neighbour processing.

Graph Index

- Choose index: Survey various potential indices to determine if they can be built adaptively.
- Implementation: We are using Rust because it is easy to program (unlike, we believe, C++) while still being performant on the same level as C++ and C. The code must be very efficient in order for the results to be significant.

Evaluation

- More than just BFS: PageRank, Bellman-Ford, maybe more.
- Datasets: Varying topologies, sizes and densities. We also intend to look into real-world data sets such as yahoo, twitter and some chemical molecule networks.

Report

- The final contributions to be reported will be decided on prior to the start of writing.

Roadmap

- Before exams: Implement more compressive cracking methods, including but not limited to eager and lazy compression.
- Early Easter: Choose graph database index or optimal structure and determine how it can be built on-the-fly.
- Mid-late Easter: Implement chosen technique in Rust
- Early summer term: Gather benchmarking results, create appropriate graphs.
- Mid-late summer term: Report write-up and presentation preparation.
- Run up to deadline: Package and document code, proof read/polish report.

4. Evaluation

Dataset

We have been using randomly generated trees of up to 30,000 nodes so far, testing on the breadth-first search algorithm. The results so far will be subjected to further optimisations. They have been put into the section on the current state of the project.

The intention is to use graphs of varying topologies, sizes and densities for the final benchmarking tests, or as many varieties as our final contributions are relevant to. We will also want to look at real world datasets, such as the twitter graph and yahoo web graph.

Current methods

The graph is represented as an adjacency list with two columns labelled `src and `dst, representing directed edges. Nodes in the graph are identified by integers starting from 1.

The breadth-first search benchmark creates a random spanning tree with bidirectional edges and picks a random start node. Candidate algorithms are required to traverse the entire tree from there, returning a list containing the nodes they have visited, in order (so that it is easy to verify that the algorithm is correct by sorting and ensuring every node is visited).

This breadth-first traversal uses the same basic approach in each implemented algorithm - a frontier of nodes is maintained, initially containing only the start node. At every iteration, a new frontier is populated containing the neighbours of nodes within the current frontier, excluding those which have been visited before. To check if a node has been visited before, a bitvector is used, a 1 at the nth index of the bitvector indicates that the nth node has been visited. The only essential difference in each algorithm is how the neighbours of a given node are found, which in the case of preclustering, depends on preprocessing of the graph. I will detail the way in which the adjacency list is used to find the neighbours of a given node in the frontier for each of the algorithms we have currently implemented and benchmarked.

Naive nested loops

In this method, the entire adjacency list is scanned to find all the neighbours of a given node. It is fast on very tiny graphs because there is no preprocessing overhead, however it does not perform on par with other methods beyond graphs of around 100 nodes.

Preclustering

The adjacency list is sorted on the src column before the algorithm starts. To get the neighbours of a node, it doesn’t scan, but does a binary search of the adjacency list. Because the values are not clustered, the binary search can land anywhere within the contiguous range of instances of the sought value, therefore two scans are done from the point landed at, one forwards and one backwards, in order to visit every instance in the adjacency list of the sought value. The preclustering ensures that for every node sought, the cost of finding its neighbours is the cost of a binary search plus the cost of scanning all of its instances - much less than the naive nested loops, which must scan the entire adjacency list every time.

Preclustering with RLE

Essentially the same as preclustering, but adjacency list is instead converted into an array of arrays, in which the nth index of the array contains the neighbours of the nth node in the graph. This means that the cost of finding the neighbours of a node is simply the cost of indexing into the adjacency list.

Adaptive clustering

Queries the adjacency list while cracking with every query, which clusters the visited nodes in the src column. In this way it is similar to both the naive nested loop method and the preclustering method. It is still scanning, but only pieces of the column known to contain a given vertex are scanned, giving a significant speedup, while having no preprocessing overhead.

Adaptive clustering with compaction

In terms of the breadth-first search this works the same as normal adaptive clustering, however for every visited node its instances in the adjacency list are physically compressed, reducing its size. The details of this method are given in the next section. It is currently giving results slower than normal cracking (but still faster than naive BFS), however, unlike the other methods we have not yet profiled the runtime performance of this method to search for improvements. We are intending to use this method also as a stepping stone to explore other methods of online compression using cracking.

5. Current State

Implementation

So far, we have implemented a method of adaptive clustering on adjacency lists and have performed some initial benchmarks of this system, outlined in the section on evaluation. We also have an intermediate system which uses adaptive clustering, but also compresses the data after clustering. At the moment, data is actually compacted. We intend to explore different methods of implementing online compression on top of cracking, as detailed later in this section.

Compressive Cracking

Cracking

For our implementation, we use an array as the cracker index rather than an AVL tree as in the original implementation. This is done so that it is trivial to access the key-value pairs for adjacent keys in the index, which is required when checking if it is appropriate to compress, and also during tuple reconstruction to confirm if the value being read is compressed or not.

Compression

Consider the table in figure 5.1, containing an integer cracker column, crk and another column, x, representing the payload. The cracker index is denoted crk-idx. Starting from the left, we see what happens during the compaction of a value within the cracker column. The ofs column stands for offset.

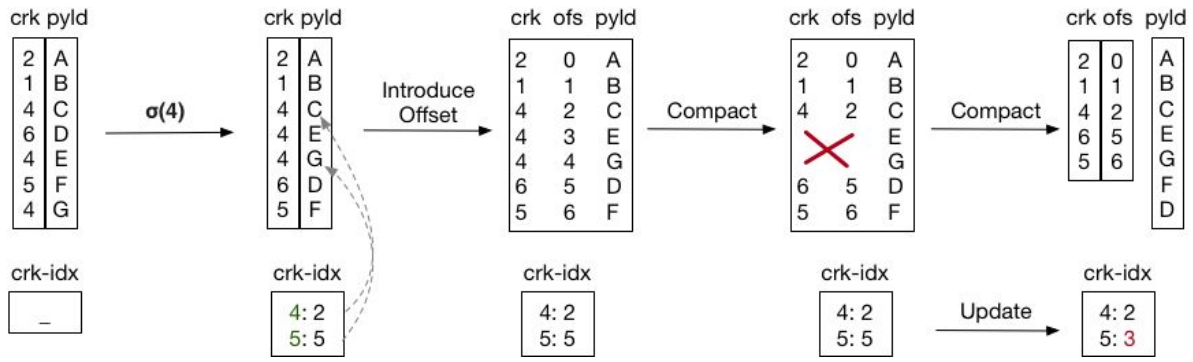

Figure 5.1. Steps in online compression with cracking.

1. A query for 4 comes in. The system performs cracking as usual. The earliest position of any value in the column greater or equal to 4 is known - index 2. The earliest position of any value in the column greater or equal to 5 is also known - index 5. Having two consecutive integers stored in the cracker index, it is known that every value with an index in [2, 5) is equals 4, and that there are no other 4s anywhere in the column. Therefore it is safe to compress.
2. Firstly, offsets into the uncracked column are introduced - simply the consecutive integers from 0 upwards. This represents the current correspondence from the cracker column into the base columns.
3. The contiguous instances of the value being compressed are deduplicated in the array representing the cracker column, and the later elements in the array are moved towards the beginning to fill the gaps.
4. Some values in the cracker index may now be incorrect, because the elements whose value is greater than the value that was compressed, that is, those elements of the column that have been shuffled towards the start of the array, may have stored indices in the cracker index that are now inaccurate.
5. The shift was by a number of places one less than the original number of instances of the compressed value. In this case, the first 4 occurs at index 2 and the last at 4. This is the same information as we used to decide to compress. We know that during deduplication values were shifted 5 - 2 - 1 = 2 places towards the array's start.
6. To account for this, we subtract 2 from all values in the cracker index whose key is greater than the compressed value.

Tuple reconstruction

After compressing the cracker column, we must change the way we reconstruct tuples, because the cracker column no longer corresponds directly to the base columns. Rather than using the cracker column as a direct index into the base columns, we use the offset column to index into the base column, using the cracker index to check how many values we will be reading.

Design space of online compression schemes built on cracking

This is the only compression method we have implemented at the moment, but there are others possible. For example, it is not necessary to compact columns after identifying a compression opportunity, a lazier approach, however it is also possible to run length encode values during the cracking scan, a more eager approach. We intend to investigate other compression methods which can be built on top of cracking and evaluate their performance to see if any runtime performance can be gained from applying this derived knowledge about contiguous single values in the column.

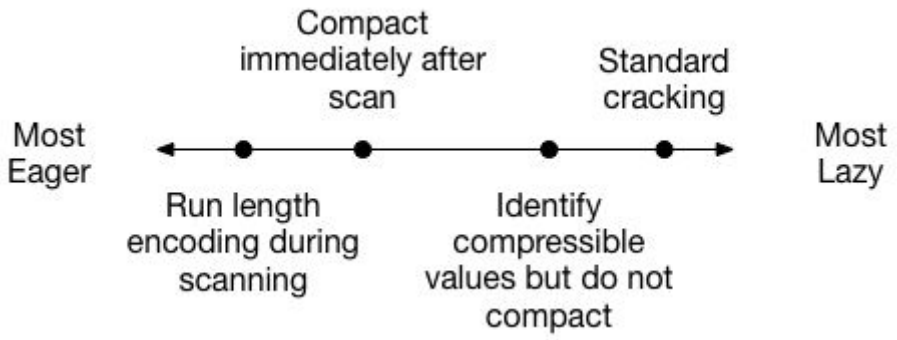

Figure 5.2. The design space of compressive cracking.

Evaluation

Adaptive clustering (standard cracking)

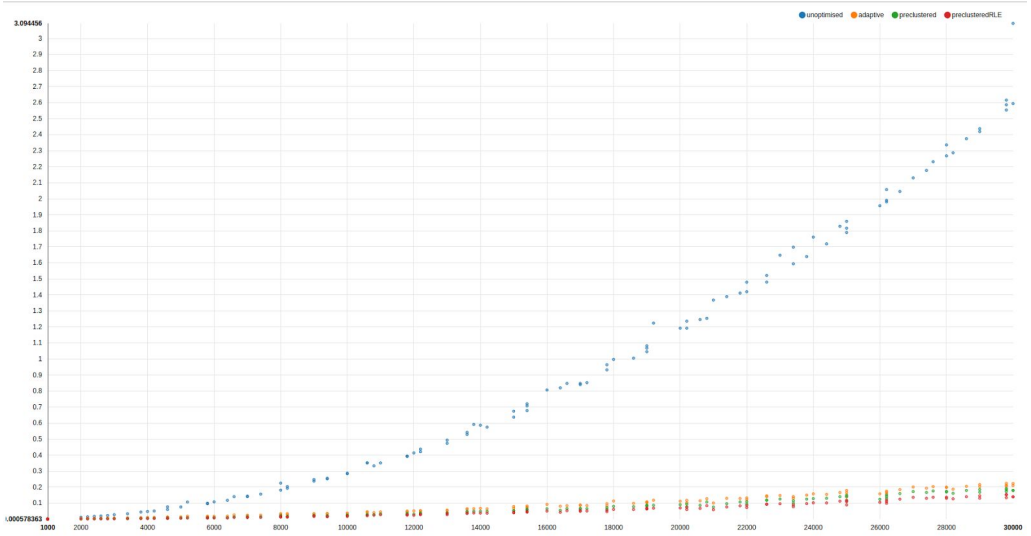

Figure 1. Adaptive clustering with standard cracking performs comparably to preclustered data on graphs up to 30,000 nodes in size.

Adaptive clustering with compaction (cracking + compacting)

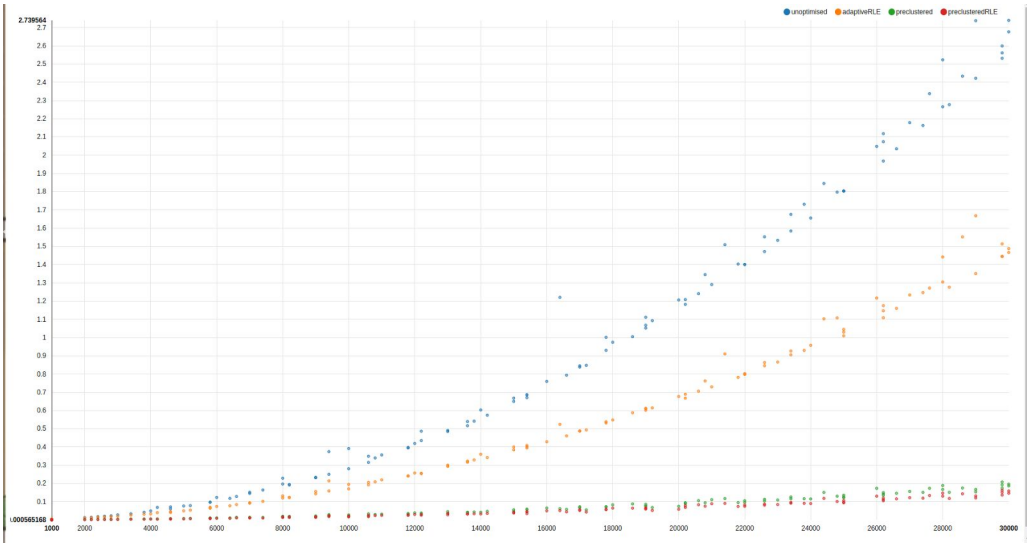

Figure 2. Introducing compaction makes adaptive clustering much slower using the same comparison.