Adaptive graph processing

Interim Project Report

Rob Moore (rrm115)
Holger Pirk (pirk)

Version 3

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.

Current Progress

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, which takes significant time and hence is very slow, however, this is very closely related to a system in which the data is not compressed, but offsets into the indexed column are still stored and utilised to reduce the number of touched array values. Details of the next steps to be taken are in the section on planning.

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 partially sorts relevant pieces of a copy of a column by scanning them and maintains 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.

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.

Index-free adjacency

A term often associated with some popular graph databases, such as Neo4J, this is a storage method which, 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.

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.

Benchmarking

- 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.

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, which it is to be stressed are not final and will be subjected to further optimisations, have been put into the section on current results.

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.

The breadth-first search benchmark creates a random fully (bidirectionally) connected tree with a random start node and candidate algorithms visit every node 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).

Naive nested loops

In this method, the src column is scanned repeatedly to get dst nodes, which are added to the frontier (if not seen before) and the process is repeated until the frontier is empty.

Preclustering

The adjacency list is sorted on the src column before the algorithm starts. During each iteration of the algorithm it doesn’t scan, but does a binary search for each node in the frontier, to get the next frontier.

Preclustering with RLE

Essentially the same as preclustering, but the src column is compressed into an array of arrays which store the adjacent nodes. Because the nodes are numbered like indices, this means that no binary search is necessary, at every iteration of the loop the next set of potential frontier additions is queried by simply indexing the adjacency list with a member of the current frontier.

Adaptive clustering

Queries the adjacency list while cracking with every query, effectively clustering 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.

Adaptive clustering with compaction

This is an intermediate technique towards adaptive clustering (without compaction), it is not expected to be performant, because the nature of compaction is that it requires deleting arbitrary elements from within an array, which then requires shuffling all later elements in the array to the left. This linear time element removal is expensive and slow, making compaction not a good option for performance. The argument for saving memory is not especially strong because if you really needed to save memory you wouldn't use cracking.

Current results

Adaptive clustering (standard cracking)

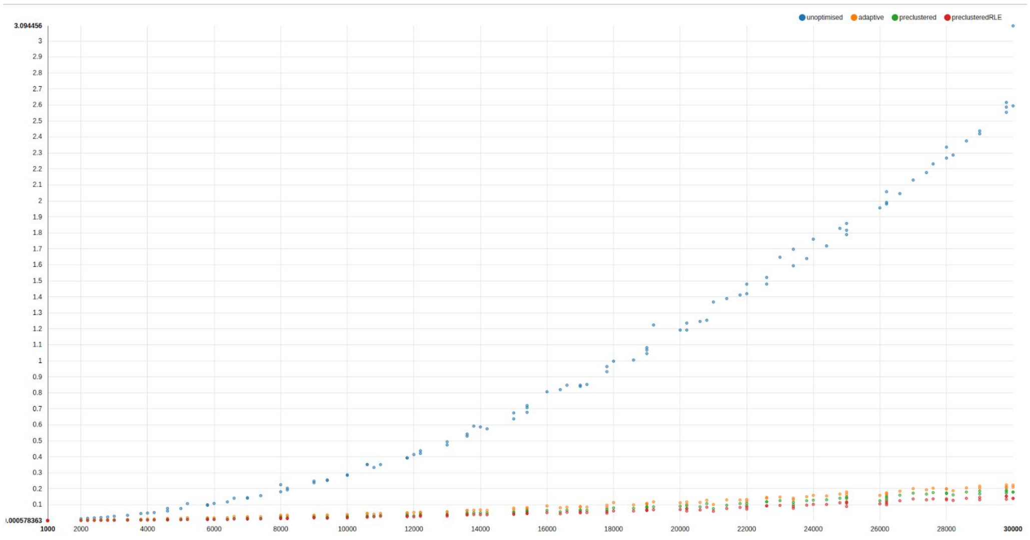

Figure 1. Adaptive clustering with standard cracking performs comparably to preclustered data on graphs up to 30,000 nodes in size.

Adaptive clustering with compaction (cracking + compacting)

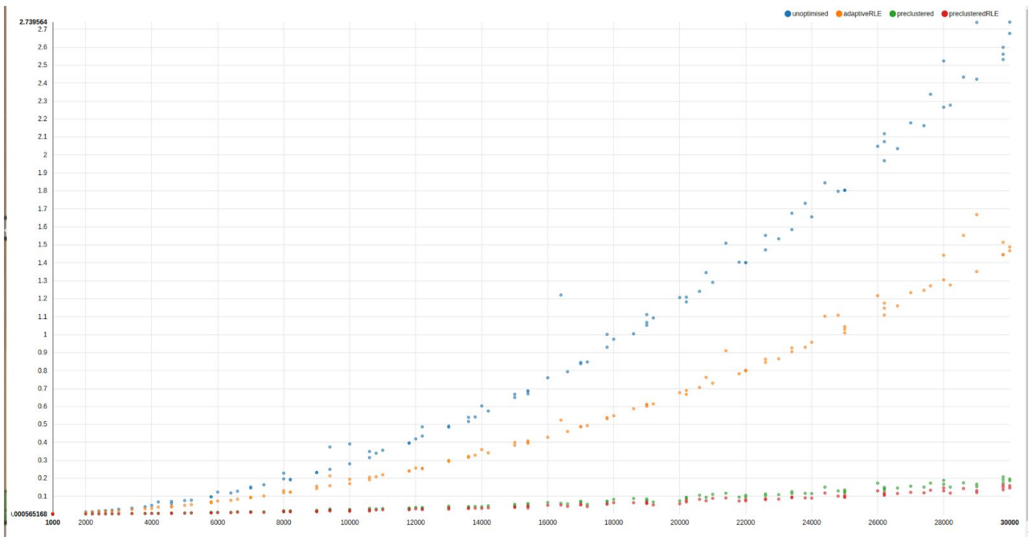

Figure 2. Introducing compaction makes adaptive clustering much slower using the same comparison.