Adaptive graph processing

Interim Project Report

Rob Moore (rrm115)
Holger Pirk (pirk)

Version 1

Introduction

Connected data is increasingly being leveraged to find valuable intelligence, 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 advance the value that can be delivered to users.

We are looking to propose a novel method of query execution on connected data to perform online restructuring and compression to improve query performance.

Current Progress

At this stage, our method involves using a relational database as an adjacency list and adaptively indexing it based on incoming queries using Stratos Idreos' database cracking technique. We have also added run length encoding to this adaptive clustering method in order to assess any improvement on performance. We have run experiments with breadth-first search, evaluating against a range of techniques (covered later).

This is not however intended to be our final method. We are also looking into graph database indices with a view to create techniques which can build them on-the-fly. At the moment we're looking into is GRIPP, a graph indexing technique based on pre- and post-order labelled of depth-first search trees. GRIPP is a graph processing framework that is used on top of a relational adjacency list model.

Background and Related Work

Database cracking [Idreos07]

This is an adaptive indexing technique for relational databases. The DBMS maintains a copy of the ‘cracked’ column (the cracker column) and information about how this copy is partially sorted (the cracker index). Every query scans the appropriate section of the cracker column, applying a partial sort. The information about the location of certain values in the column gained from this partial sort is then stored in the cracker index.

Idreos showed that the cracker index built using this technique has been shown to be able to optimise other database operations, such as insertions, updates and joins.

We are applying cracking by clustering nodes on the ‘from’/’src’ side of an adjacency list, with a view to decrease scan time during the recursive self-joins. We are also exploring adaptive compression by using the run length encoding compression scheme to further reducing the amount of scanning required. At the moment we have only implemented an intermediate method which actually compacts the column, but is slower, at least for breadth-first search.

GRIPP [Leser06]

Using depth-first search trees pre- and post-order numbering scheme for a graph G, it is possible to build a GRIPP index, IND(G), which can be used to answer distance and reachability queries. The index has the benefit that it is fast to query, but also not impractically expensive to build like a transitive closure would be. The authors showed that querying the GRIPP index can also be optimised using various pruning strategies.

Index-free adjacency

A term often associated with certain popular graph databases, such as Neo4J, this is a storage technique which, rather than storing nodes and edges in tables, stores all relationships as pointers which are contained within every record. 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 not explored it so far in this project.

Ligra [Shun13]

"Lightweight Graph Processing Framework for Shared Memory" is a collection of programming routines for processing in-memory graphs in parallel. It uses an interface with two main functions: EdgeMap and VertexMap. The author's implementation is on github.

EdgeMap maps a given function over all frontier edges where the destination vertex satisfies a given predicate. Users of this framework must ensure that any side-effects of the given function are safe to apply in a shared-memory system. VertexMap maps a given function over a given set of vertices.

Workload-aware RDF database structure [Aluc13]

Aluc implemented chameleon-db, a database with no fixed physical design, using group-by-query (G-by-Q) representation of RDF data which is purely workload driven. Inspired by database cracking, he also proposed adaptive index building based on the properties of the edge-disjoint RDF clusters used by chameleon-db to prune non-matching triples.

Plan

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 graph operations.

Graph Index

- Choose index: Survey multiple potential indices, not just GRIPP
- Implementation: Must be very efficient.

Testing

- More than just BFS: PageRank, Bellman-Ford, maybe more.
- Datasets: Varying topologies, sizes and densities.

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, 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 appendix A.

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.