GRAPH CLUSTERING AND CACHING *

Alberto O. Mendelzon   Carlos G. Mendioroz

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4
{mendel,tron}@db.toronto.edu

ABSTRACT

We present the design of a system that stores and manipulates graphs on secondary storage. The goal is to minimize the I/O needed to access arbitrary directed graphs, using a bounded amount of main memory. The two key ideas are: a heuristic method for clustering on the same disk page nodes that are likely to be accessed together, and a caching mechanism that adapts well to a variety of graph traversals, including depth-first and breadth-first searches.

1 INTRODUCTION

Graphs are pervasive data modeling tools. Storing and manipulating large graphs on secondary storage can be useful in graph-oriented data management systems, such as G+ [1], hypertext systems like HAM [2, 3], or at the object storage level of object-oriented database systems. Several authors have pointed out [4, 5, 6] that most of the recursive queries that appear in practice can be viewed as graph traversals.

When graphs are large, simply loading them completely into main memory (or virtual memory) and applying standard main memory algorithms may not be practical or desirable. For example, suppose a geographic information system stores a complete description of the road and street network for every major city in Europe, and we want

---

*Work supported by the Natural Sciences and Engineering Research Council of Canada, the Institute for Robotics and Intelligent Systems, and the Information Technology Research Centre.

to compute a best route from a street address in Barcelona to a street address in Prague. Reading the complete network into memory and then running a standard shortest path algorithm is likely to be infeasible, and, even if feasible, highly inefficient. Another motivation for supporting secondary storage traversals efficiently is the implementation of object-oriented database systems (OODBMS's). An OODBMS typically views a database as a graph of objects and their relationships and tries to ensure that related objects are stored physically close to each other on the disk so they can be accessed with few disk accesses[14].

We present the design and prototype implementation of a graph storage server, dubbed *CGM* for "Clustered Graph Machine," that implements efficient clustering and caching for supporting common graph traversal algorithms. Our system supports only the topology of the graph; node and arc attributes are assumed to be stored separately and managed with conventional techniques. However, we do provide for labels on nodes and arcs. Labels are uninterpreted by *CGM* and can be used by clients to store *signatures* to represent node and arc attributes concisely.

2 CLUSTERING AND CACHING GRAPHS

2.1 Graphs

We use standard graph theory terminology, defined in the full paper. We treat directed labeled multigraphs unless stated otherwise.

2.2 Clustering and caching

Our model of a secondary storage device will be an array of *pages*, the unit of input/output (I/O) transfer, associating a constant cost to any page access operation. We will assume that this cost is high enough as to disregard any other cost associated with the accessing strategy such as address calculations, main memory management and in-core general computations. Thus, our goal from the performance point of view is to minimize page I/O.

One of the ways to minimize I/O is to store related records together in the same page, so when one page is brought to memory to operate on a record, related records are transferred as well and will not require a new I/O operation when they are accessed. The techniques used to achieve this are called *clustering* and the sets of related records, *clusters*. Note that several clusters may fit in one page and, conversely, one cluster may span many pages.

Another way of reducing I/O is by using a *cache*. A cache is an area of main memory used to store pages that have been read and are likely to be accessed again in the future By using a cache, when a page is to be read/written, then we first check whether it is in the cache and if it is, the operation can be completed in memory and we say that there was a *cache-hit*. For a general description of caching techniques, see [7].

In the rest of this section, we propose specific methods for clustering and caching when storing arbitrary graphs. Section 2.3 describes a naive file structure for graph storage, which is improved upon in Section 2.4. Section 2.5 describes the incremental clustering mechanism and Section 2.6 presents our approach to caching.

2.3 File structure for graphs

The basic operations supported by any file structure for graphs should be:

- Inserting, removing and updating nodes and arcs.
- Retrieving nodes and arcs, either by key or in sequence.
- Finding the outgoing and incoming arcs from a given node.

A graph can be represented by its nodes and its arcs. In a naive file structure, nodes are stored in a relation containing node keys and labels, and arcs in another relation containing pairs of nodes and arc labels. The basic operations can be supported by storing nodes ordered by key and arcs ordered by starting node. If such an ordering is used, all arcs that have a given node as tail are together and can be easily found. To retrieve all incoming arcs, either a second index on ending node can be used (thus introducing data redundancy in the structure) or a full scan of the file is required. [1]

2.4 Node oriented clustering

An approach that differs from the naive one above is to consider the graph as a set of nodes, and storing with each node all its incoming and outgoing arcs. Note that this implies duplicating arc information, since each arc is stored with both its head node and its tail node.

Now, assuming that we store nodes in pages, we face the problem of determining which nodes are to share the same page; in other words, we have to determine a good clustering of the nodes to optimize page access. Some work has already been done in designing node oriented clustering mechanisms. Restricting graphs to DAG's (directed acyclic graphs), Banerjee *et al.* [8] propose a clustering scheme that efficiently supports traversal of the graph only accessing outgoing arcs for each node, namely, "downward" or "forward" traversal. They define a *clustering sequence* to be the order of the nodes induced by a traversal method, and compare the performance of different clustering sequences. They propose a node oriented clustering and they do not deal with the problems involved in representing the arcs[2]. The physical organization is in pages that split and merge much the same as a B-tree, indexed by a virtual node key generated so that the clustering sequence is achieved.

Larson *et al.* [9] revise this approach to support traversal recursion. They propose a two-file organization and a new node key assignment that overcomes some problems with updates that implied mass reorganizations in the earlier approach. Nevertheless, the insertion effectively needs a dense node-key universe [3] to avoid mass reorganizations. They suggest that a simple heuristic approach to determine the clustering sequence (topological order) may be appropriate, since finding the best one is NP-Complete for reasonable cost measures.

---

[1] Another variant is to store one flag along each arc to indicate its direction and replicate the arcs, thus with only one index we can retrieve both outgoing and incoming arcs.

[2] They store predecessors and descendents in a variable length field of the node record. Note that they actually duplicate the arc information.

[3] Dense key universe implementation is not discussed in [9], but techniques similar to those used in extensible hashing may be used [10, 11].

One important handicap common to both approaches is that they are only useful for DAG's. Tsangaris and Naughton [12] propose a node clustering that overcomes this problem but only in the static case, *i.e.*, they propose a way to determine a good clustering for a given graph, but they do not deal with graph updates. Also, they count on the availability of a "probabilistic description" of the expected access patterns.

Other authors have explored graph clustering in the context of object-oriented database systems. The scheme used in the *Cactis* system[13] is essentially static, with no support for incremental clustering. The O2 approach[14] is based on exploiting schema information to create *placement trees* which are then flattened into clusters.

In the next Section we present a clustering scheme that can deal with arbitrary graphs and updates, and that also can exploit probabilistic information about access patterns when such information is available.

2.5 Dealing with cycles: clustering by heuristic

We have reduced our problem to finding a clustering of the nodes, assuming each node's arcs will be stored along with the node. We need to determine in which page each node should be placed.

2.5.1 Incremental cluster determination

Our scheme is to construct the clustering in an incremental manner: we provide simple heuristics to place a new node in the graph and to split a page when it becomes full. In doing so we consider arcs as liaisons between nodes that we try not to break apart, *i.e.*, we try to keep both ends in the same page. We call an arc with this property a *local* arc. In contrast, arcs that cross page boundaries, that is, the head node is in a different page than the tail node, are called *external*.

Node insertion When a new node is to be inserted into a graph, its chosen page is the one that maximizes the number of arcs that will end up being local. In other words, we choose the page that contains the most "neighbours" of the new node. This is most effective when the arcs are inserted along with the node, that is, when a node is inserted, so are the arcs incident to it, although our method also supports insertion of isolated nodes, as described later.

A similar heuristic was independently considered by Chang and Katz[15], who were interested in supporting clustering of objects incrementally in an CAD database system. The main difference between our approach and theirs is our more sophisticated page split algorithm, described next.

Page splitting When an insertion of a node or an arc would render a page overfull, the page needs to be split. In the process of so doing, we consider the subgraph induced by the nodes in the page. If this subgraph has more than one connected component, we separate some of the components into a new page, *preserving all local arcs as local.* If there is only one connected component, then we find a min-cut, that is, a cut-set with minimum number of arcs in it, and we cut the component into two pages with a minimum number of local arcs becoming external. Details of the page split algorithm are in Section 2.5.2.

We have discovered that by using this approach we are bringing the problem closer to two other seemingly unrelated problems, namely *graph drawing* and *component placement in VLSI design.* In both areas objects have to be placed in a space and the goal is to bring related objects together. Arcs (in the case of graph drawing) and connections (in VLSI design) make the liaisons. Although there are some important differences (we are not restricted to a 2-dimensional space, and we have to cope with graph updates), the approach we suggest has already been used in those areas to some extent [16, 17].

2.5.2 Page split

Arc weights There are some considerations that were oversimplified in order to present the clustering scheme. One such consideration is that arcs may not be all of the same importance: the graph might have some structure that makes some arcs more likely to be traversed than others.

This is captured by assigning *weights* to arcs. An arc has a default weight of 1. If some arc is assigned a weight of N, $N > 1$, that would mean that traversing that arcs is N times more likely than the default. On the other hand, a weight of 0 would be assigned to an arc that is very unlikely to be traversed. When splitting a page, the min-cut is calculated using this weights, so arcs with higher weights are more likely to stay local.

Page zero Also, we supposed that nodes were to be added with incoming and outgoing arcs, but what happens if a node is added *per se* ? In our scheme, there is no way of telling which page is the best one to put this new node... In fact, there is no best page.

Our solution to that is to have a distinguished page called *page zero* where we put all connectionless nodes. As soon as an arc is added to a connectionless node, it is relocated to the page of the other end node, thus rendering a local arc. The latter policy is dubbed "node stealing" because the node is being "stolen" from its original page.

Page split algorithm The details of the algorithm used to split a page are given in the full paper. We sketch the algorithm here. We first check if the page to be split is *page zero*, and if this is so, then we move out to a new page all nodes that have connections (only connectionless nodes are left in *page zero*). If that would leave *page zero* more than half full, we also move some connectionless nodes.

If it is not *page zero*, we construct a graph in memory representing the subgraph contained in the page we are splitting. If there is more than one connected component, we can split the page without turning any local arc into external just by moving to a new page some entire connected components. The problem now is: what is the best possible split ? *Optimal component distribution* (so that the number of nodes in each page is balanced) is shown NP-complete in [18], so we use the heuristic of sorting the components by size, and then one by one (in descending size order) putting each component in the page with less nodes so far, to obtain a balanced distribution.

We then have to deal with the case of only one connected component, in two different ways: If there is only one node in the component then the problem is that there is no space for more arcs in this page (this is a well connected node). A continuation page, a page that holds overflow arcs for very connected nodes, is created to hold some of the arcs and a special link is used to reference it. All the operations so far can be performed in time linear in the number of nodes and local arcs present in the page that is being split.

If there is more than one node in the connected component to be split, we calculate a min-cut in the page subgraph. The process starts by picking the two most connected nodes (the ones with more local arcs in the graph). The most connected, *source*, is going to stay in the page and the other, *target*, is going to be moved to a new page.

This choice of *source* and *target* is arbitrary. It is done to avoid calculating min-cuts between every possible pair of nodes, as that would add a quadratic factor to the work in the number of nodes.

Calculating a max-flow between *source* and *target* (using standard max-flow algorithms like in [19]) a family of min-cuts is determined: Considering the weights as arc capacities, we repeatedly find a path of minimum length between *source* and *target* (called an augmenting path), calculate the path capacity (the minimum capacity of the path arcs), and reduce the capacity of the arcs of the paths by the path capacity; until the graph is disconnected. Note that arcs with zero capacity are "discarded".

When the graph is disconnected, a min-cut has to be determined. It might be the case that there is more than one min-cut in a graph. To choose from possible min-cuts, we create two sets of nodes: one that is to stay with *source* and the other that is to leave with *target*. We put in the *source* set all nodes that ended in the same component as *source* after the max-flow was determined, and in the *target* set all nodes in the *target* component.

Afterwards, we begin to add connected components to the set with fewer nodes until all nodes are in one of the sets, thus producing a *balancing min-cut*, *i.e.*, a min-cut that balances the number of nodes in the *source* and *target* sets.

One a min-cut has been computed we decide if we indeed use it to split the page or not, depending on the balance achieved: if the resulting partition is very unbalanced, we make what we call a *crown* around *source* by putting *source* in a page of its own. What we want to prevent is to end up cutting a star-shaped graph one point at a time. By creating a crown we achieve this by generating a two-page cluster having its center in one page.

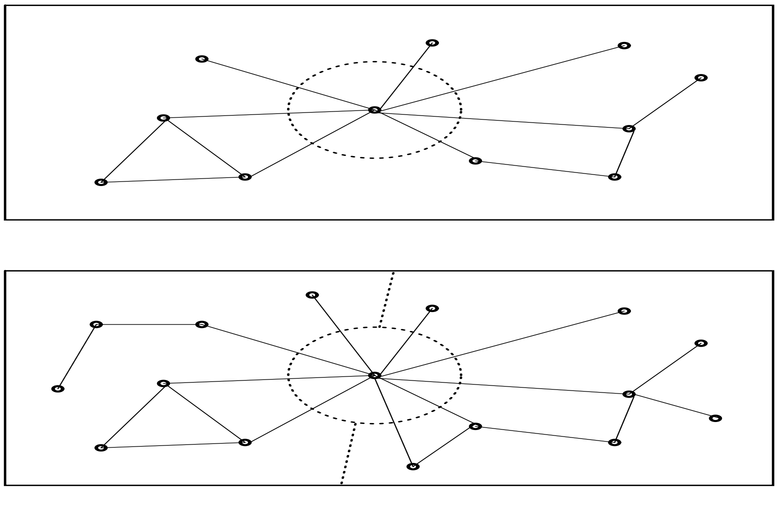

Figure 1: Multi-page cluster development

As more nodes are added, the crown may be further split thus creating a multi-page cluster as in figure 1.

2.5.3 Complexity of cluster determination

The only computationally non-trivial step in the scheme used to determine the clustering is the page splitting when the induced subgraph contains only one component.

There are no clear criteria to determine which cut-set is the best. Naturally, a min-cut is a candidate, though there may be exponentially many different min-cuts from which to choose. Also, if a min-cut leaves too many nodes in one of the pages, maybe by cutting off a leaf node, then it is very likely that the page will be split again in the near future. This process may repeat itself several times in cases of star-shaped components.

A balance between the output pages would be desirable, *i.e.*, that the number of nodes in each of the output pages be approximately half the number of nodes in the input page. Moreover, this is difficult to achieve. In fact, *optimal balancing graph split* is shown to be NP-complete in [18].

Therefore, we have adopted a heuristic as a solution.

2.6 Caching with hints

Clustering and caching should be considered together. Caching by itself is a performance aid, but clustering can boost the cache impact, by increasing page reference locality.

When using caches, the impact of the reclaim policy of the cache, *i.e.*, the algorithm used to select which cache page is to be freed when one is needed and all are used, is widely recognized. One common drawback of cache implementations is that, even though an application might have some knowledge of the future accesses that it will perform, no information channel exists for that knowledge to reach the cache subsystem.

In graph applications this is particularly relevant, because it is often the case that an application will access nodes in a particular order, and so it "knows" which pages [4] are going to be needed in the future. Furthermore, the application may at some point need to access a given set of nodes in no particular order and this degree of freedom might be exploited at the cache level.

We provide provide the necessary information channel in a clean way by creating a shared data structure to hold "nodes to be visited." The client accesses this data structure as a *dequeue*: nodes can be inserted at either end. The client inserts a node at the head of the dequeue when it expects that this node will be accessed in the future before any other node in the list, and it inserts it at the end if it thinks it will be accessed in the future, but after all other nodes in the list have been accessed. More precisely, the list holds *sets* of nodes as opposed to single nodes. Thus a client can insert a given set of nodes, say, at the head of the list, when it expects to access them in the future before any other nodes in the list, but it does not care about the relative order in which nodes within the set are accessed. The cache subsystem can access this data structure to decide which pages to reclaim and to reorder requests whenever possible, to take advantage of already-in-cache nodes.

[4] The application may not know which pages, but the combination of its knowledge and that of the graph clustering is enough to determine them.

It may be viewed as a burden on the client programmer to have to think about which nodes will be accessed next, but in many cases the programmer does not have to think about this at all; she can simply use standard graph traversal algorithms such as depth-first and breadth-first traversal, which will automatically place nodes in the dequeue in the right order.

3 PRELIMINARY RESULTS

We have implemented a prototype *CGM* server in C++ on top of Unix. The (ongoing) implementation is still incomplete: there are protocols only for adding nodes and arcs, and for retrieving them. No deletes or updates are allowed. However, we have performed some preliminary performance tests on the prototype with encouraging results.

We tested two basic client algorithms: BFS (breadth-first search) and DFS (depth-first search). We programmed variants of these algorithms to take advantage of node list management in two ways: using node-by-node insertion (*i.e.* unitary sets in the node list) or using set insertion. Also, a pseudo-BFS where successors are stored in sets according to their distance to the root of search was tested. The output of this last algorithm is not a BFS ordering but it possesses one of its properties, *i.e.*, being ordered by depth.

We compared their performance, measured as number of page read operations when searching random graphs of size ranging from 1,000 to 10,000 nodes and 3,000 to 30,000 arcs. The results showed that the savings due to clustering are not as important as those due to caching. Still, the I/O cost with *CGM*'s clustering was consistently smaller than that of a random clustering. Also, as expected, enabling hint passing to the cache subsystem makes the impact of changing the cache size much more important.

The most interesting result to date is the performance of the DFS algorithm when working with the full hint passing scheme, *i.e.* when using the node list to store the set of descendants of the current node and letting the server choose among them which to search next. In this case, the algorithm constructs a DFS ordering accessing the graph almost optimally, *i.e.* each page is read once.

Please see [18] for a complete description of the preliminary tests conducted and the results obtained.

4 FUTURE WORK

To complete the implementation is the obvious next step. Also, we would like to develop another system on top of *CGM* that would be capable of handling graphs with structured data associated to its nodes and arcs. *CGM* would act as a kernel to support graph primitives and hold indices and signatures to main data bases.

Ullman and Yannakakis [20] have an interesting result on the I/O complexity of computing transitive closure of a graph: they show that I/O equal to $O(n^3/\sqrt{s})$ is sufficient to compute transitive closure of an n-node graph using memory of size s. Also, they show this is a lower bound for a class of algorithms they call "standard". Interestingly, they claim that clustering is not an issue, as they state that a perfect clustering can be achieved, *i.e.*, the clustering that would achieve that each page read brings to memory exactly what is needed at that time. However, we believe that the complexity of calculating the perfect clustering for a graph may exceed that of calculating the transitive closure. We would like to test the performance of the algorithms proposed in [20] using *CGM* primitives, and to further investigate the impact of clustering on transitive closure I/O complexity.

Finally, the topic of a massive reorganization of the graph is also left open. The technique described in [12] seems to be a good candidate and we would like to merge that into *CGM*. We have also learned that simulated annealing is successfully used to perform similar tasks in graph display layout and VLSI component layout [21, 22]. It would be interesting to examine the applicability of this approach to graph clustering, and conversely, to see whether our techniques can be applied to the problem of laying out a graph on a two-dimensional screen.

REFERENCES

[1] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. G+: Recursive queries without recursion. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 355–368, 1988.

[2] Norman Delisle and Mayer Schwartz. Neptune: a hypertext system for CAD applications. In *Proceedings of ACM-SIGMOD 1986*, pages 132–139. ACM Press, 1986.

[3] Joe Goodman, Jim Bigelow, Brad Campbell, and Victor Riley. Hypertext abstract machine software functional requirements specification. Technical Report CASE-20-3-6, CASE Division, Tektronix, Inc., 1987.

[4] H. V. Jagadish, Rakesh Agrawal, and Linda Ness. A study of transitive closure as a recursion mechanism. In *Proceedings of ACM-SIGMOD 1987*, pages 331–344. ACM Press, 1987.

[5] Arnon Rosenthal, Sandra Heiler, Umeshwar Dayal, and Frank Manola. Traversal recursion: A practical approach to supporting recursive applications. In *Proceedings of ACM-SIGMOD 1986*, pages 166–176. ACM Press, 1986.

[6] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[7] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Inc., 1987.

[8] Jay Banerjee, Won Kim, Sung-Jo Kim, and Jorge F. Garza. Clustering a DAG for CAD databases. In *IEEE Transactions on Software Engineering*. Institute for Electric and Electronic Engineers, November 1988. vol. 14, No. 11.

[9] P.A. Larson and V. Deshpande. A file structure supporting traversal recursion. In *Proceedings of ACM-SIGMOD 1989*, pages 243–252. ACM Press, 1989.

[10] R. Fagin, J. Nievergelt, N. Pippenger, and R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM TODS*, 4:315–344, 1979.

[11] P. Larson. Dynamic hashing. *BIT*, 2:184–201, 1978.

[12] Manolis M. Tsangaris and Jeffrey F. Naughton. A stochastic approach for clustering in object bases. In *Proceedings of ACM-SIGMOD 1991*, pages 12–21. ACM Press, 1991.

[13] S. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database. *ACM TODS*, 14(9):291–321, 1989.

[14] V. Benzaken. An evaluation model for clustering strategies in the o2 object-oriented database system. In *Proc. Third Int'l. Conf. on Database Theory*, pages 126–140. Springer-Verlag LNCS 470, 1990.

[15] E. Chang and R. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented database. In *Proceedings of ACM-SIGMOD 1989*, pages 348–357. ACM Press, 1989.

[16] Thomas Fruchterman and Edward Reingold. Graph drawing by force-directed placement. Report UIUCDCS-R-90-1609, Dept. of Computer Science, University of Illinois at Urbana, 1990.

[17] M. A. Breaer. Min cut placement. *Journal of Design Automation and Fault Tolerant Computing*, 1(4):343–382, 1977.

[18] Carlos G. Mendioroz. Graph storage and manipulation in secondary memory. Master's thesis, Department of Computer Science, University of Toronto, 1991.

[19] Shimon Even. *Graph Algorithms*. Technion Institute Computer Science Press, 1979.

[20] Jeffrey D. Ullman and Mihalis Yannakakis. The input/output complexity of transitive closure. In *Proceedings of ACM-SIGMOD 1990*, pages 44–53. ACM Press, 1990.

[21] L. K. Grover. Standard cell placement using simulated sintering. In *Proceedings of the 24^{th} Automation Conference*, pages 56–59. Institute for Electric and Electronic Engineers, 1987.

[22] R. Otten and L. van Ginneken. *The Annealing Algorithm*. Kluwer Academic Publishers, Boston, 1989.