Buffering of Intermediate Results in Dataflow Diagrams

Allison Woodruff and Michael Stonebraker
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley[1]
Berkeley, CA 94720
email: *tioga@postgres.berkeley.edu*

Abstract

Buffering of intermediate results in dataflow diagrams can significantly reduce latency when a user browses these results or re-executes a diagram with slightly different inputs. We define the optimal buffer allocation problem of determining the buffer contents which minimize the average response time to such user requests. We show that this problem has several characteristics which render traditional latency reduction techniques ineffective. Since optimal buffer allocation is NP-hard, we propose heuristic methods for buffer management of intermediate results.

We present a simulation of the behavior of these heuristics under a variety of conditions, varying graph structure and access pattern. We argue that history mechanisms which track user access patterns can be used to improve performance. We further show that graph structure and access pattern determine the factor of improvement which is possible. The performance enhancements we describe can be applied to minimize query response time in visual dataflow languages.

1: Introduction

Dataflow languages apply a sequence of operations to specified inputs. In many cases, the final output of a dataflow diagram is the only result examined by a user. However, when performing tasks such as debugging or tuning, a user may wish to view intermediate results. In a naive implementation, intermediate results are not saved when a dataflow diagram executed. As a consequence, if a user asks to view intermediate results, these results need to be recalculated. Since computation costs may be extremely high, the delay in response time can be substantial. A more sophisticated implementation can support buffering of intermediate results. Because blindly buffering all intermediate results may not be feasible, such a system must attempt to select for buffering those intermediate results which most significantly minimize latency.

We examine strategies for buffering of intermediate results in dataflow diagrams in the context of Tioga [1], a graphical application development tool which uses the boxes and arrows notation popularized by scientific visualization systems such as AVS [2], Data Explorer [3], and Khoros [4]. Tioga improves upon these systems by providing sophisticated data management using the POSTGRES database management system (DBMS) [5].

In the Tioga programming model, boxes represent user-defined database queries or browsers which display data, and edges between boxes represent flow of data. Although a limited number of boxes has currently been

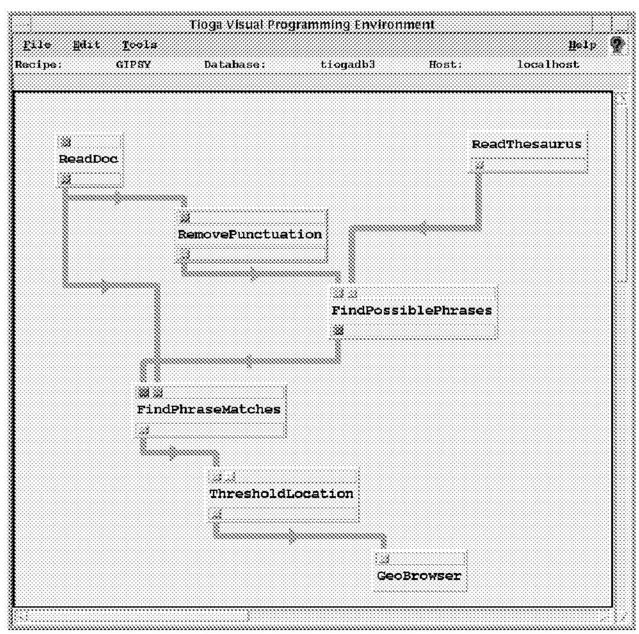

Figure 1
A Sample Tioga Recipe

[1] This research was sponsored by NSF under grants FD94-00773 and IRI-9411334.

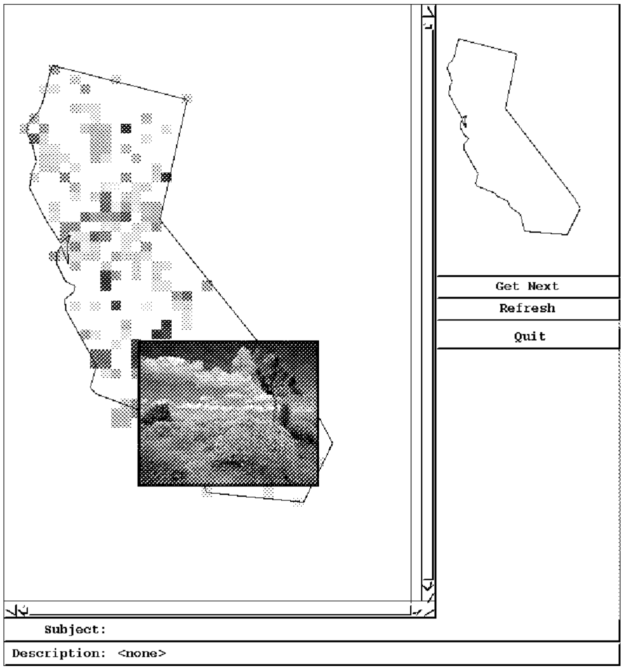

Figure 2
Data Displayed in a Tioga Browser

implemented, additional boxes may be programmed by users. Nonexperts build visual programs called *recipes* by interactively connecting boxes together using a graphical editor. Current applications include a photographic 35mm slide library and a geoindexing system.

Figure 1 shows the recipe of the geoindexing system which indexes text documents according to the geographic locations to which these documents refer. At the end of the recipe is a browser box which displays documents according to the resulting indexes. The default Tioga browsing paradigm allows users to visualize data results in a multidimensional space. Users navigate through their data using a flight-simulator interface. Additional browsers may be implemented by advanced users. Figure 2 shows a browser displaying the final results of the recipe in Figure 1. Objects are displayed in a latitude/longitude viewing space that contains California. One object, a digitized 35mm slide, has been selected and displayed.

Tioga was motivated by the needs of scientific users in the SEQUOIA 2000 project [6]. In a typical task, these users will construct a recipe, run it on a specified set of inputs, and view the final result. If this result contains an anomaly or some unintuitive or unwanted result, users might want to perform the following types of actions:

- **search query**. In this case, the user examines intermediate results of Tioga boxes to locate data of interest, e.g. the source of an anomaly. Intermediate results may be viewed by placing browsers at arbitrary points in the recipe.
- **modification query**. Users may want to tune a recipe. In this case, they will rerun it using different parameters as input to specific functions. Alternatively, they may wish to modify the code of a particular box and rerun the entire recipe with the new box. Finally, they may incrementally develop an application, as supported by systems such as Weaves [7].

Searching and modification may be performed individually or in combination. For example, debugging may entail a sequence of search and modification queries to locate and correct a faulty processing step or data.

Attempts to reduce the latency of these types of queries raise several interesting issues. Specifically, we observe that buffering of intermediate results can significantly improve performance. In this paper, we examine buffer management strategies to improve the performance of Tioga on search queries. The results presented are directly relevant to modification queries.

In Section 2, we define the problem of optimal buffer allocation. In Section 3, we present our assumptions. In Section 4, we describe our mechanism for generating graphs which are input to our model. In Section 5, we present our simulation model and our heuristics. In Section 6, we present our results. Finally, in Section 7, we discuss our conclusions.

2: Problem definition

Consider the recipe graph of Figure 3 in which box A is an input and box I represents the final output. In this recipe, if the buffer is empty and box F is being calculated, the results from boxes A, B, C, and E are calculated as part of the computation of box F, and the results of A, B, C, E, and F are therefore all candidates for buffering. However, if there is insufficient buffer space to contain all these intermediate results, we must choose which box results to buffer. The *worthiness* of box results (their potential to reduce latency) depends on a variety of complex considerations, beginning with compute cost and buffer space requirements.

The worthiness of a box result is also significantly

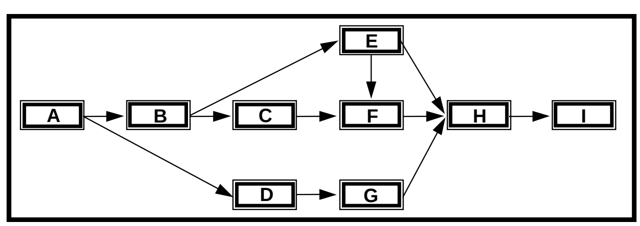

Figure 3
Structure of a Sample Dataflow Diagram

affected by the structure of the recipe graph. Box results in the buffer can save time during the computation of other boxes by obviating the need to compute all their ancestors. For example, if the results from boxes C and E are in the buffer and box F is being queried, only box F needs to be computed during this move (since boxes C and E have *guarded* boxes A and B, making their computation unnecessary). We additionally observe that the current residency of the buffer pool affects the worthiness of other box results.

Finally, the probability that a box result will be accessed impacts its worthiness. This probability is constructed according to a probabilistic move model described in detail in section 5. In this model, we define a *query* to be the user's request to view the results of a single box. A *query path* is a sequence of queries. The current query is the user's *position* in the query path. At each position, a possible *buffer allocation* may be made. A set of buffer allocations for the entire path is called a *solution*.

We define the optimal buffer allocation at a given position as the one which will minimize the average response time to an unknown future sequence of queries on intermediate results. This allocation will be made based on a *configuration* which includes the following information: the structure of the recipe graph, the user's current position in the recipe graph, the current contents of the buffer, and the probability distribution of the user's expected movements. We assume that box results which are in the buffer pool or the results of boxes which must be computed at a given position are *candidates* for inclusion in the buffer allocation at that position.

As a result of the complications listed above, calculating the optimal buffer allocation is in fact NP-hard. This can be shown by a polynomial reduction from the Knapsack problem [8] to our problem of optimal buffer allocation. In such a reduction, compute times and buffer space requirements correspond to the value and size of objects to be placed in the knapsack. Intuitively, the search space is extremely large since it must consider the impact of all possible allocations on the latency of all possible query paths which could be followed in the future (and in turn all the buffer allocations for each position in each path). Performing an exhaustive search of all buffering solutions for all future paths is not feasible. Naively calculating the optimal allocation for positions in short query paths through graphs with a small number of boxes (e.g. 10) took days on a DEC Alpha.

However, we believe that near-optimal buffer allocations can be made by heuristics. We further posit that the following characteristics of optimal buffer allocation render existing buffer management techniques ineffective: (1) box results can guard other boxes; (2) the sizes of box results vary; (3) the compute times of boxes vary; and (4) the future reference stream is impossible to predict. Because traditional caching and buffer management techniques [9, 10, 11] such as LRU do not consider guarding, items of variable size, or variable fetch costs, they make poor buffer allocation decisions for intermediate results. Register allocation is in many ways more similar to the optimal buffer allocation problem we are considering. However, heuristics to solve register allocation are predicated on an understanding of the future reference stream [12]. This understanding allows the register allocation heuristics to eliminate many results from consideration. Because the reference stream in search query paths in dataflow diagrams is unpredictable, it can not be constrained. Finally, although techniques exist which minimize buffer usage within a single execution of a dataflow diagram [13], these techniques do not consider retaining intermediate results for search or modification queries.

As a consequence, we have developed new heuristics which are more appropriate to the optimal buffer allocation problem. In this paper, we compare the behavior of a number of these heuristics on a variety of graph types and user access patterns.

3: Assumptions

We make several assumptions. We assume the existence of recipe graphs which have already executed and materialized final results. All inputs and final results are saved as tables in our underlying DBMS. However, we assume that intermediate results of recipes are not saved. We assume that the system has bookkeeping information which tells us the exact compute time for each box, as well as the exact size of its output.

For simplicity, we assume the common case in which each graph has one box which is a *terminal* box. The terminal box has no outputs and is saved in a DBMS table. We refer to boxes which do not output data except to the terminal box as *sinks*. These boxes are of interest because queries to them can require the computation of a large number of boxes, therefore providing a large set of results as candidates for the buffer pool. Boxes which do not take input from other boxes are called *sources*. In a query path, a user views the results of the terminal box and then performs a sequence of queries on the recipe graph, visiting a number of boxes before terminating the search. We assume that this sequence is not known in advance and that the terminal box is never revisited.

We assume the existence of a workspace in which computations are performed. For each recipe, we assume the existence of a separate buffer of limited size. Each time a user asks to view an intermediate result, our strategies determine which box results they would like to retain in the buffer after its computation. Desired box results in the workspace are copied into the buffer. We assume that box results may only be buffered in their entirety. We assume without loss of generality that this buffer space is on disk. In most cases the buffer space available will not be sufficient to store all intermediate results. Our goal is to choose which intermediate results to buffer to minimize the average latency of future search queries.

4: Graph generation

Because only a limited number of recipes generated by users are available to us, we randomly generate recipe graphs for our tests. We observe that boxes in dataflow diagrams, both in Tioga and in other systems, are typically composed of groups. While groups have a relatively high degree of interconnectivity, there tends to be a relatively low degree of connectivity between groups in a graph.

Therefore, we generate groups as follows. The group generator creates a certain number of boxes (a value randomly chosen from a specified range). The graph generator takes as input a range of orders of magnitude for buffer sizes and compute times. To assign a value from one of these ranges, we first randomly select an order of magnitude from the specified range. We then randomly select a number from within that order of magnitude. The resulting distribution of numbers generated resembles an exponential distribution. In our studies, we focus on graphs in which buffer sizes vary by up to three orders of magnitude (between 1 and 100) and compute times vary by up to six orders of magnitude (between 1 and 100,000). These values are based on observations of the scientific applications which we support. After the boxes have been assigned buffer sizes and compute times, we add edges which result in acyclic graphs with a controlled *branching factor* (the average number of edges per box). In this study, we generated graphs ranging from a low branching factor of approximately 1.2 to a high branching factor of approximately 1.8.

The graph generator makes calls to the group generator a specified number of times. It then adds edges between the groups as follows. The graph generator connects a source, sink, or intermediate box in the first group to a source, sink, or intermediate box in the second group according to a specified probability function. In the results presented in this paper, plausible values are chosen based on informal observations of existing dataflow diagrams. We designate the box in the first group a source, sink, or intermediate box with probability 10%, 80%, and 10%, respectively. The box in the second group is a source, sink, or intermediate box with probability 25%, 70%, and 5%, respectively. The probability distribution of these connections controls the relative numbers of sources and sinks in the final graph. Finally, after all groups have been connected, all sinks are connected to a terminal box.

5: Simulation model

We next implemented a simulator which would measure the performance of a variety of buffering strategies on various graphs. We defined a move model which specifies the sequence of intermediate results examined by a user. If a user is positioned at a given box, it is possible for them to:

- move **backward** in the graph to a parent (e.g. from box F to box C in Figure 3).
- move **forward** in the graph to a child (e.g. from box C to box F).
- move **sideways** to a spouse box which shares a child (e.g. from box F to box G).
- move to a **random** box in the graph (e.g. from box G to box B).
- **reset** (the query path ends; terminate and clear the buffer).

Each of these five possibilities is assigned a probabilistic value; the five values sum to one hundred percent. The probability of resetting indirectly controls the length of a single query path. We studied a variety of probability distributions including, for example, a largely backwards, short query path characterized by 50-10-15-10-15 and a relatively random, long query path characterized by 13-13-18-53-3 (backward-forward-sideways-random-reset).

The simulator generates a complete query path and passes it to procedures which mimic the behavior of buffering strategies on that path. At each position, the buffering strategy has a list of candidate box results which could be retained. This includes results which existed in the buffer previously as well as results which must be calculated at the current position. A *viable* candidate is one which will fit in unallocated space in the buffer. At each position, the strategies assume they have the entire buffer space to allocate and fill it according to their heuristic. The allocation ends when the strategies determine the buffer is full or when they determine that none of the unbuffered candidate box results is viable.

For each heuristic, the cost of the entire query path using its solution is recorded. We have examined a large number of strategies in this way. For the sake of brevity, we discuss only the following in this paper:

- **No Buffering**: No intermediate results are cached. This represents the worst case.
- **First-in-first-out (FIFO)**: At each position in the query path, FIFO buffers box results in reverse timestamp order. Timestamps represent the creation time of a box result within a query path. Within a position, we assume the necessary boxes are computed in topological order, since no box may be computed until its ancestors are computed. Timestamps are therefore assigned according to a post-order, depth-first traversal.
- **Random Average**: At each position in the query path, this strategy uniformly at random selects box results and attempts to buffer them. A buffer allocation is complete when the buffer is full or all available box results have been buffered.
- ***k*-Random**: Random Average as above is run *k* times on a fixed query path; the *k*-Random solution is the one with the best running time. For data in this paper, values for *k* range between 64 and 256; separate simulations have established that higher values of *k* yield only marginal improvements.

  Note that the performance of *k*-Random may not be achievable in practice. Because *k*-Random runs multiple times on the same query path, its solutions are based indirectly on knowledge of future moves in the specific path being tested. Consequently, it does not try to optimize for the average case, but for the specific path which is being tested. In other words, *k*-Random takes advantage of information about future moves not available to other heuristics.
- **Path Cut**: At a high-level, Path Cut's heuristic is to minimize the cost of hypothetical backward query paths. These hypothetical paths are the set of all paths which begin at any node and consist only of backward and reset moves. The cost of such backward paths can be decreased by the buffering of their midpoint. Results of boxes which have some combination of the following characteristics are therefore desirable: (1) midpoint of multiple backward query paths; (2) midpoint of at least one expensive backward query path; or (3) midpoint of at least one backward query path which is likely to occur. At each position in the fixed query path, Path Cut assigns the midpoint of each hypothetical backward path a path cut value (PCV). The worthiness of a box result is the sum of its PCVs for all paths. A greedy algorithm is used to attempt to buffer box results in the expected order of their worthiness.

  The worthiness function is computed as follows: we assume that we are making a buffering decision while computing a current box *c*. For every box in the recipe graph, we construct all paths to each of its ancestors. We calculate the sum S_p of the compute costs for all boxes in a path *p* from a box *n* to an ancestor *a*. The compute cost of a single box is the sum of the costs of all its ancestors (all the boxes on which it is dependent for input) assuming that no box results are buffered. We identify the computational midpoint *m* of path *p*. *m* is the box along *p* such that the sum of the compute costs between *a* and *m* is greater than or equal to 50% of *S*. We then calculate two probabilities. For this calculation, we assume that we have perfect information about the probability distribution of the moves described above. First, we calculate the probability P(*p)* of the path between *n* and *a* occurring given that the user reaches box *n* (this is the probability of a backwards move to the power of the length of *p*). We then calculate the probability $P(c \rightarrow n)$ that *n* will be the box visited immediately after *c*. The PCV of *m* is equal to $S_p * P(p) * P(c \rightarrow n)$. The worthiness is the sum of all PCVs for a given box.
- **Path Cut No Probabilities (NP)**: This heuristic is identical to that above, with the exception that the PCV assignment considers no probabilities, i.e. the PCV of *m* is equal to S_p.

We simulated the above strategies on a variety of graphs and with a variety of access patterns. We present the results of these simulations in the next section.

6: Results

This section quantitatively demonstrates the benefits of buffering of intermediate results. First we show that significant gains can be achieved by such buffering. We next show how these benefits vary with heuristic, graph structure, and access pattern. We then examine the behavior of heuristics for varying buffer sizes. Finally, we discuss the results of additional experiments we conducted.

We begin by characterizing the maximal reduction in latency which can be achieved by buffering of intermediate results. We assume the buffer is empty when computation begins. The best possible performance occurs when every box result which is computed is inserted in the buffer and not removed until queries on the graph are complete. (The computation cost in this case is not simply the cost of executing the recipe since a user may choose to examine only a subset of the intermediate results.) In this situation, the buffer space needed is at

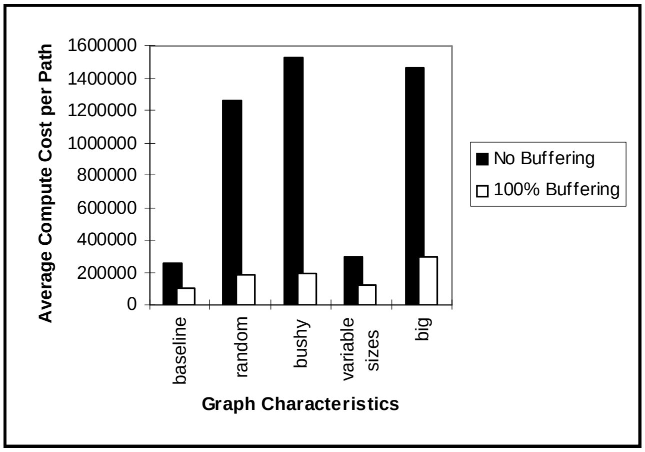

Figure 4
Compute Costs For Varying Graph Structures and Access Patterns

most the sum of the space requirements of all boxes in the graph, since each box result need be stored at most once.

Figure 4 presents the relative performance (according to average compute cost per path) of No Buffering and 100% buffering solutions to query paths for a variety of graph structures and access patterns which can be characterized as described in Table 1. The **Size** column notes the average number of boxes in the graph. Graphs with an average number of 18 boxes consist of three groups. The **BF** column characterizes branching factor (low of 1.2 and high of 1.8). The **C** and **B** columns indicate the number of orders of magnitude variation among the compute times and buffer space requirements of box results in a recipe graph. **P(back)** and **P(random)**, indicate the probability of backwards movement and random movement. **PL** represents the path length, indirectly controlled by the probability of resetting.[2]

	Size	BF	C	B	P(back)	P(random)	PL
baseline	18	low	6	1	high	low	short
random	18	low	6	1	low	high	long
bushy	18	high	6	1	high	low	long
variable	18	low	6	3	high	low	short
big	54	low	6	1	high	low	long

Table 1
Graph Structures and Access Patterns

For each type described above, we generated dozens of graphs and ran thousands of query paths within each graph. Figure 4 presents the average of the results. We observe that if no buffering is done, three independent conditions can make query paths expensive. First, longer query paths are more expensive. Second, query paths through larger graphs are expensive (because the computation of a single box may depend on the computation of a larger number of ancestors). Third, query paths in bushy graphs are more expensive (a higher degree of connectivity also implies that the computation of a single box may depend on the computation of a larger number of ancestors).

It is apparent that buffering 100% of the results as they are computed significantly reduces the average compute time per path. The greatest gain is achieved for the random graph; 100% buffering in this case is 13% of the cost of No Buffering. This data clearly demonstrates that buffering of intermediate results is desirable.

We next examine the improvements which would be possible with a smaller buffer. In Figures 5-7, we show the relative benefits of the various buffer management schemes under a variety of conditions. We note that there is a certain minimum computational cost which will be incurred by any solution. Therefore, we compare solutions according to the computational cost they incur above this minimum. We compare a heuristic solution with the No Buffering solution as follows. If a heuristic cost is h, the 100% buffering cost is B_{100}, and the No Buffering cost is NB, the Y axis contains $(h - B_{100}) / (NB - B_{100})$. Values close to 0% mean the heuristic closely approximates 100% buffering; higher values mean the heuristic performs in a manner similar to No Buffering.

Figure 5 presents the performance of various heuristics on the same graphs, access patterns, and query paths detailed in Figure 4, assuming a buffer 10% of the size of the 100% buffer. We see that graph structure and access pattern affect the behavior of the heuristics. Considering graph structure, we see that the heuristics are clustered together fairly tightly for the variable size graph set. This occurs because each heuristic attempts to fill the fixed size buffer completely. As the buffer fills, an increasingly small number of box results are viable candidates. As a

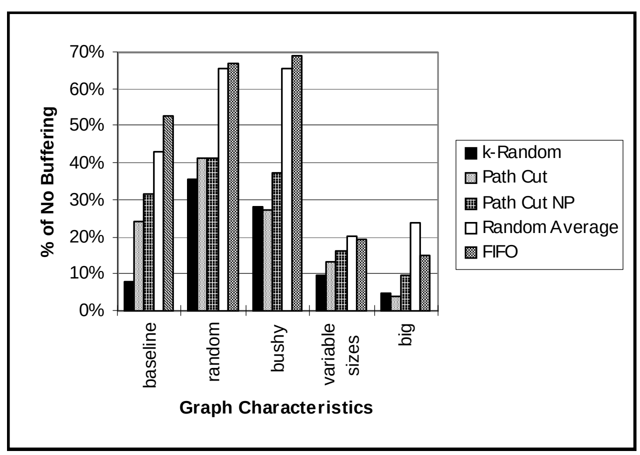

Figure 5
Performance of Heuristics with 10% Buffering

[2] The complete move probability distributions are 50-10-15-10-15 (baseline), 13-13-18-53-3 (random), 53-13-18-13-3 (bushy), 50-10-15-10-15 (variable), and 53-13-18-13-3 (big).

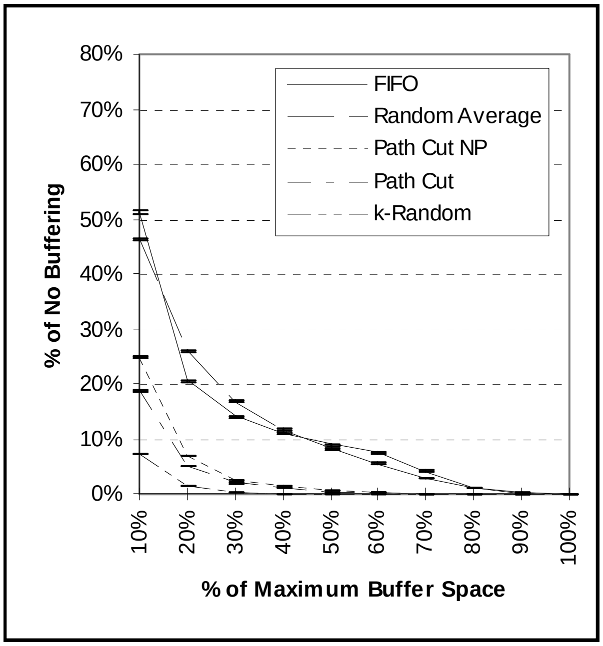

Figure 6
Heuristic Performance for Baseline Case

consequence, there is a certain set of box results with small space requirements which is chosen by most heuristics. This similarity in buffer allocations compresses the difference among the heuristics. Considering variations in access pattern, we see that in the random graph set (which has longer query paths), the heuristics only come within approximately 30% of the 100% buffering case. However, this is not because the heuristics are making poor buffering decisions, but rather is a result of the small buffer size. Even in the optimal solution for these query paths, items must be removed from the buffer and calculated again later.

We also see that in many cases, heuristics come close to the performance of 100% buffering. Specifically, *k*-Random and Path Cut tend to do extremely well. The relative performance of the heuristics for each graph type and access pattern is relatively consistent, with a few interesting exceptions. For example, in most cases Path Cut tends to outperform Path Cut NP. However, it loses its advantage on query paths with a high degree of randomness. This implies that the worthiness assignment being used by Path Cut may be most effective for access patterns in which the user has a high probability of moving backward through the graph. We also observe that in the bushy and big graphs, Path Cut actually outperforms *k*-Random. This is because *k*-Random has a much lower chance of finding a good solution for a long query path than for a short query path (since *k* is fixed and the set of solutions for a long query path is much larger than the set of solutions for a short query path). Also observe that FIFO does quite poorly in general, often worse than Random Average. Intuitively, since FIFO buffers the most recently generated box results, it always attempts to buffer the result of the box which is being visited. Since the user most often moves away from that box, often to ancestors which have no dependence on it, this is a poor strategy.

Figures 6 and 7 show the performance of heuristics as a function of the size of the buffer. Error bars represent 95% confidence intervals. Note that the performance of the heuristics quickly converges as the buffer size increases. Observe also that for certain strategies, a buffer which is approximately half the size of the maximum buffer can yield the same performance as the maximum buffer. This is largely because many query paths do not access all box results, and so much of the maximum buffer remains unused.

We tested many different types of heuristics. Due to space constraints, we will not enumerate them here. We simply note that, in general, the other heuristics we examined performed slightly worse than Path Cut and slightly better than Path Cut NP.

We also investigated the usefulness of approximate rather than perfect information about the probability distribution of the user's movements. The performance of heuristics using approximate information is quite close to that of the heuristics using perfect information. Since history mechanisms may be used to approximately predict the user's future movements, we conclude that keeping history about the user's access patterns is advisable.

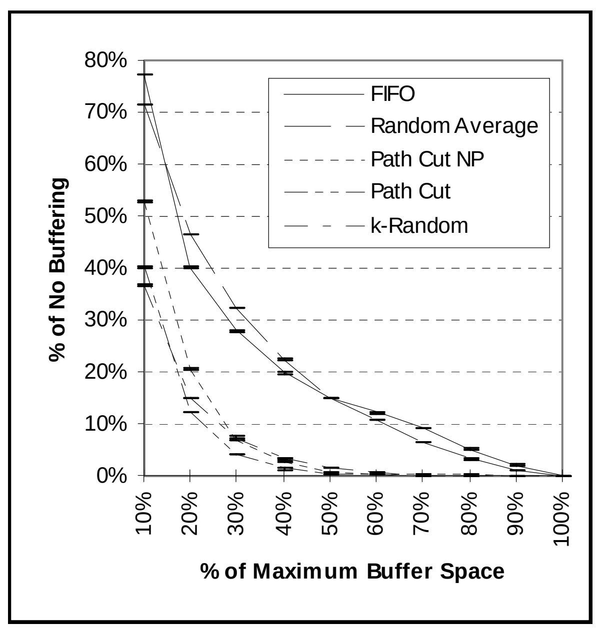

Figure 7
Heuristic Performance with High Branching Factor and Long Query Paths (C = 6, B= 1)

7: Conclusions

We have seen that buffering of intermediate results can significantly reduce the latency of search queries in dataflow diagrams. Use of a relatively small buffer can provide substantial improvements over no buffering. Further, traditional strategies such as FIFO are much less effective than the new heuristics which we propose. These heuristics approach the maximal improvement possible. The most effective heuristics make predictions about the user's access pattern, suggesting that a history mechanism is warranted.

There are many potential directions for further research. Certainly a variety of other heuristics could be examined. Additionally, extending our model and simulator to consider modification in addition to search queries would be straightforward. Further, we could consider optimal buffer allocation for a multiuser buffer pool, i.e. when the amount of buffer space for a recipe can vary over the course of the query path.

A more complex extension would consider buffering of partial results. First, modifications may affect only part of a box result. Second, since browsers can display a subset of a box result, it may not be necessary to calculate an entire box result. We believe the buffering of partial results is a fruitful direction and have begun to study these issues. In this context, slaved browsers [14] raise an additional complication. When two browsers are slaved together, examining a partial result in one browser spawns a process which generates a corresponding partial result in another browser. Our model could be extended to consider this type of dependency in the access pattern.

References

[1] Stonebraker, M., Chen, J., Nathan, N., Paxson, C., and Wu, J., "Tioga: Providing Data Management for Scientific Visualization Applications," Proceedings of the 1993 VLDB Conference, Dublin, Ireland, August 1993.

[2] Upson, C., Faulhaber Jr., T, Kamins, D., Laidlaw, D., Schlegel, D.,. Vroom, J., Gurwitz, R., and VanDam, A., "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, 9:4, July 1989, pp. 32-40.

[3] Lucas, B., Abram, G., Collins, N., Epstein, D., et al., "An Architecture for a Scientific Visualization System," Proceedings of the 1992 IEEE Visualization Conference, Boston, Massachusetts, October 1992.

[4] Rasure, J. and Young, M., "An Open Environment for Image Processing Software Development," Proceedings of the 1992 SPIE Symposium on Electronic Image Processing, San Jose, California, February 1992.

[5] Stonebraker, M. and Kemnitz, G., "The POSTGRES Next-Generation Database Management System," *Communications of the ACM*, 4:10, October 1991, pp. 78-92.

[6] Stonebraker, M. and Dozier, J., "SEQUOIA 2000: Large Capacity Object Servers to Support Global Change Research," SEQUOIA 2000 Technical Report 91/1, University of California, Berkeley, March 1992.

[7] Gorlick, M., and Razouk, R., "Using Weaves for Software Construction and Analysis," Proceedings of the 13th International Conference on Software Engineering, Austin, Texas, May 1991.

[8] Garey, M. and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, New York, 1979.

[9] Smith, A., "Cache Memories," *ACM Computing Surveys*, 14:3, September 1982, pp. 473-530.

[10] Denning, P., "Virtual Memory," *ACM Computing Surveys*, 2:3, September 1990, pp. 153-188.

[11] Chou, H.-T., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems," Proceedings of the 1985 VLDB Conference, Stockholm, Sweden, August 1985.

[12] Steenkiste, P., "Advanced Register Allocation," in *Advanced Language Implementation* (P. Lee, ed.), MIT Press, Cambridge, Massachusetts, 1991.

[13] Tsui, K., Fletcher, P., and Hutchins, M., "PISTON: A Scalable Software Platform for Implementing Parallel Visualization Algorithms," Proceedings of Computer Graphics International, Melbourne, Australia, June 1994.

[14] Woodruff, A., Wisnovsky, P., Taylor, C., Stonebraker, M., Paxson, C., Chen, J., and Aiken, A., "Zooming and Tunneling in Tioga: Supporting Navigation in Multidimensional Space," Proceedings of the IEEE Symposium on Visual Languages, St. Louis, Missouri, October 1994.