Scheduling Queries With Large Out-of-Core Data Sets

Aman Sinha and Craig Chase

TR ECE-PDS-1995-018 October 1995

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Scheduling Queries With Large Out-of-Core Data Sets*

Aman Sinha† Craig Chase‡

October, 1995

Abstract

We consider the problem of dynamic, preemptive scheduling of queries that have a shared data set. The model consists of (a) a processing node with limited physical memory, (b) a disk resident data set, (c) Poisson arrival of queries, (d) known probability distributions of processing times and (e) fixed, non-zero cost of data fetches from the disk. The objective is to minimize the flow time of the queries. In this paper we assume that the queries may be scheduled in any order. Data is fetched from the disk in units of "chunks", which are assumed to be fixed size. Each query may process the data chunks in any order. Since the data is shared, a single data chunk may be required by multiple queries. Hence, an efficient scheduling algorithm must ensure that the chunk is read from disk a minimum number of times. It is well known that the shortest expected remaining processing time (SERPT) rule can minimize the flow time for memory resident tasks. However, it is unclear how successfully SERPT can be adapted to exploit data locality.

In this paper we show that SERPT can be easily augmented to include data prefetching. Through simulation, we show that prefetched-SERPT (PSERPT) achieves very good locality of reference and is very effective at minimizing flow time even with very fine-grained computation, and a high-degree of inter-query data locality.

1 Introduction

In recent years, increasing processor speeds and distributed computing are making feasible query processing over very large databases. During the same period, the relative price of RAM memory and disk memory has diverged. Effectively, the price of a byte of RAM memory has more than quadrupled since 1992. These effects mandate out-of-core query processing in practical, large database systems. Equally disconcerting is the trend in disk performance. Although disks have become much less expensive, and much more reliable, disk access times have improved only marginally over the past several years. Relative to the microprocessor, which has experienced dramatic performance improvements in the same period, the disk has effectively gotten much slower. Hence, a database system engineer is faced with the following paradox. To design a cost-effective system, she must accept that most of the database will be on disk and not resident in memory. Yet, to design a high-performance system, she must avoid paying the high disk access time if at all possible.

Fortunately, these two design constraints are not necessarily mutually exclusive. If there is sufficient locality in the accesses to the database, we can have the majority of the data on disk, but still access that disk infrequently. In this paper we consider how to schedule queries to exploit this locality. We assume that locality comes through multiple, independent queries. That is, our database system may be a multi-user environment, responding to several requests. Each of requests is a query to the database, and will require accessing indices

*This work supported by NSF Grant CCR-9409736

†sinha@ece.utexas.edu

‡chase@ece.utexas.edu

and records that are likely to be on disk. If the intersection of the data accessed by these multiple queries is non-empty, then there is locality that can be exploited through proper scheduling and management.

To simplify the analysis, we consider only non-mutative, *i.e* read-only database accesses. That is, the database is unchanged by the query processing, hence the queries can truly be scheduled independently. Second, we assume that each query is itself a parallel task. If the database consists of M data items, then we allow for the query to consist of up to M separate tasks. We assume the flexibility to schedule these tasks in any order. We make these two assumptions because that allow us the greatest possible assumption in scheduling the computation so that data accesses will be minimized. We note that, although certainly not all queries meet these restrictions, many types of realistic data processing do meet them. In particular, we are targeting the Nile application, a NSF National Challenge problem in computational science for High Energy Physics research [9, 12]. This project involves distributed computation using read-only distributed data. The data set exceeds 10 Terabytes of experimental results from collisions of sub-atomic particles. A typical query operation involves first scanning the database for relevant records and then engaging in numerically intensive computation with those records. The output of the query is a set of statistics, for example, the number of particle collisions that resulted in the production of a new particle with a specific mass. Hence, the records can be processed in any order.

In the past, considerable work has been done on the problems of preemptive scheduling of jobs, both for the deterministic and stochastic case, see [8, 3, 14]. One of the earliest works in minimizing response time is reported in [4], where the case of minimizing the weighted flow time for a static list of tasks is considered. The problem of minimizing response times in a deterministic case is solvable by using the shortest processing time (SPT) rule when all release times are zero. If the release times are non-zero, then a preemptive schedule based on the shortest *remaining* processing time (SRPT) will be optimal. This policy has been used widely by operating systems for CPU scheduling in batch environments, see [13]. The corresponding stochastic case, where jobs have exponential processing times, is polynomially solvable using the shortest expected processing time (SEPT) rule (see [11, 10]). An excellent survey of complexity results related to various scheduling problems appears in [10]. It must be noted that for problems where jobs have different priorities and their processing times are not exponentially distributed, there is no proof that the weighted SEPT rule will give an optimal result. In this paper, to concentrate on the problem of locality, we make the simplifying assumption that all queries have equal priorities and all of them have exponential processing time distributions.

In [1], Abbot and Garcia-Molina consider the problem of scheduling real time transactions, which involves minimizing the makespan. Their scheduling model deals with the problems of concurrency control and I/O scheduling while scheduling the transactions. More recently, equivalent problems relevant to multimedia systems have been analyzed and reported in [5] and [6]. Specifically, the problem of when to fetch a 'hot' video and which video on disk to replace when there is no free space, is conceptually similar to trying to exploit inter-query locality. In [7], a heat-based policy that replaces segments of relatively unpopular videos with segments of currently referenced videos is studied. However, we are not aware of past efforts to exploit inter-query locality while minimizing the flow time of the queries.

We conducted several experiments to measure the ability of a task scheduler to exploit data locality between multiple queries. We choose as an objective to minimize the flow time (*i.e.,* response time) of the queries. We modify the base SERPT algorithm to do efficient prefetching and call it the PSERPT (Prefetch-SERPT) algorithm. That is, a datum will be fetched from the disk whenever it is needed by the task with the shortest remaining processing time, irregardless of what other (if any) tasks also need to process this data. We contrast the performance of PSERPT to several other heuristics. The most important of these are MaxPop, a heuristic that emphasizes data locality in the schedule, and MaxEX, a heuristic that attempts to minimize disk bandwidth. For all of the heuristics, the CPU uses SERPT scheduling among those tasks for which data is present in memory. The heuristics differ only in the decision about which data items will be in memory at any time. Our results show that, surprisingly, PSERPT performs extremely well with respect to the other heuristics. We conduct several experiment over a wide range of task granularity, inter-query data locality and system load. Although PSERPT is not the winner in all cases, it is within a few percent of the best performer. The message from these results is that prefetching can be extremely effective in database scheduling applications.

The remainder of this paper is organized as follows. Section 2 describes our model of the database system. Section 3 describes the methodology used in simulation. Section 4 describes the results of the experiments and Section 5 concludes the paper.

2 Scheduling Model

The data set is denoted as $D = \{d_1, d_2, \ldots, d_M\}$. A data unit, d_i can be any convenient quanta of information, a database record, a physical disk block, a memory page, *etc.* To characterize the physical memory constraint of a processing node, we define a *slot* as follows: s_j = a storage unit j for storing exactly one unit of data. Let the set $S = s_1, s_2, \ldots, s_m$ denote the contents of the physical memory. Each s_i is an element of D. Note that m (the size of the physical memory) is assumed to be much smaller than M, the size of the data set.

We assume that queries arrive dynamically at the system. That is, given queries $\{\ Q_1, Q_2, \ldots, Q_n\ \}$, the corresponding release times are $\{r_1, r_2, \ldots, r_n\}$. Processing of query Q_i can not commence until after time r_i. Each query, Q_i consists of M tasks, one task to process each data item in the system. Recall that we assume that tasks can be independently scheduled. We denote as $x(\tau_{ij})$ the *expected processing time* of the task from Q_i that processes datum d_j, and $X(Q_i)$ denotes the expected execution time of a query. We indicate the situation when query Q_i does *not* need to process datum d_j by setting $x(\tau_{ij})$ to 0.

Under this model, if all data items are resident in memory, then a preemptive scheduling policy based on SERPT will be optimal. However, if the data set is so large that only a fraction of the data can be in memory at any one time, then SERPT will be non-optimal as the following example clearly shows. Consider three queries, all with release times of zero, Q_1 which takes 10 units of time to process, Q_2 which takes 20 units and Q_3 which takes 30 units. SERPT suggests running the queries 1,2,3. If, however, queries 1 and 3 use the same data, and query 2 uses a mutually exclusive set of data, then the schedule 1,3,2 will result in a lower average response time whenever the disk access time is more than 5 units long.

2.1 Scheduling Policies

To simplify the discussion of various scheduling policies, we organize the policies around three independent decisions. These decisions are:

1. At each instant in time, what task should be allowed to run? — We call this decision the **Execution Policy**
2. If there are more than one tasks that need data from the disk, which datum should be fetched next? — We call this decision the **Fetch Policy**
3. When a new datum arrives from the disk, which of the currently resident data items should be removed from memory in order to make room for the new data? — We call this decision the **Eviction Policy**. For symmetry, we allow the eviction policy to select the data item that has just been fetched from the disk. That is, the system may read data from the disk and immediately discard that data without ever using it. In most instances, such a decision would be a needless waste of disk bandwidth. However, it is possible that during the time that the disk access was in progress new queries arrived. In some pathological cases, the optimal response to these new queries may truly be to discard the just-fetched data and continue to process tasks using the old data.

2.1.1 Execution Policy

Only those tasks whose data is resident in memory are runnable. Other tasks must wait until the data is fetched from disk. Subject to this constraint, our execution policy is SERPT. Our estimate of processing time is based on probability distribution functions that we assume are known *a priori*. This distribution function is a "best-case" estimate which is assumed to be an accurate assessment of the processing required if the necessary data is already resident in memory. To simplify the analysis, we assume that the processing time for tasks are independent, *identically distributed* (IID) random variables. That is, all tasks are (stochastically speaking) equally expensive to perform. We assume that the scheduler knows the expected value for the processing time of a task, but has no other information about the execution time.

The execution policy is simply to run the runnable task that has the least remaining expected processing time among all runnable tasks. There are two points worthy of mention. First, a task will be preempted when a new query arrives with new, runnable tasks with shorter expected processing times. This policy is no great surprise, since SERPT requires preemption in order to be optimal in the core-resident scheduling case. Second, a task will also be preempted when it ceases to be runnable because its data has been evicted.

2.1.2 Data Fetch Policy

The data fetch policy determines which item will next be read from the disk. We assume that data are read from the disk one at a time, and we assume that the data access time is fixed and uniform for all data. These simplifying assumptions are not completely unrealistic. Parallel or pipelined access to the disk will be *less* effective than simply having a single, sequential disk that is n-times faster. Hence, by simulating the fetch policy on a faster disk we can approximate the effects of parallel I/O. Uniform access times to the data is somewhat unrealistic when data sizes are small. However, if we assume that the unit of a data fetch (a "data item") is large, then transfer time will be dominated by the I/O bandwidth. Hence, in this case, the fetch latency will be insignificant, and the fetch time will be essentially constant for all data items (which are assumed to be of fixed size).

Our fetch policy is aggressive, possibly wasting bandwidth in the aim of ensuring that data is available when it is needed. A new data item is fetched whenever the disk is free. To see how this can waste bandwidth, assume that our fetch and eviction policies are optimal, that is, memory contains the best possible set of data. Consider a state in which the processor is running task τ_{ij} (datum d_j is in memory) and the disk is free. At this point, we optimistically fetch some other datum d_k. If τ_{ij} completes before the fetch completes, (and assuming our fetch was optimally chosen), then we have made the best possible use of the resources. If, however, τ_{ij} is still running when the fetch completes, then the memory already contains the best possible data for this configuration of the system. The optimal policy is to discard the newly fetched datum, d_k.

Since the execution time can not known with assurance, our decision to prefetch can not be perfect. We must either risk wasting bandwidth to the disk or risk starving the processor by not prefetching data. We choose the former evil for this analysis.

We consider six distinct fetch policies. They are as follows:

PSERPT Fetch the datum that is required by the task with the shortest remaining processing time of those tasks whose data are not already in memory. — This policy amounts to prefetching the datum for the next task that would be run with an in-core implementation of SERPT.

FIFO Fetch the datum needed by the task that has been waiting the longest.

MaxPop Fetch the datum that is required by more tasks than any other datum not currently in memory. — This policy emphasizes locality of reference, which will increase processor throughput, but possibly come at the expense of increased response time for jobs that require "unpopular" data.

MinPop Inverse of MaxPop, fetch the datum that is required be fewer tasks than any other datum not currently in memory.

MaxEX Fetch the datum for which the total processing time of all tasks that require the datum is maximized. — This policy seeks to fetch those data items which will remain in the memory slot the longest. MaxEX minimizes the consumption of disk bandwidth.

2.1.3 Eviction Policy

The eviction policy determines which datum will be replaced when a new datum, d_k, is fetched from disk. We permit the eviction policy to select d_k itself. This allows us to model speculative data prefetching. Note that Least Recently Used (LRU) is unlikely to perform well as an eviction policy in this case. We have made no assumption about temporal locality in the data set, and furthermore, we have direct knowledge (and control) of which data items will next be used in this application. Hence we should expect to do much better than LRU.

Intuitively, the eviction policy must complement the fetch policy. For example, if we are fetching the most popular data items, we should be evicting the least popular. We thus define five eviction policies by inverting the corresponding fetch policies. The missing eviction policy (there are six fetch policies) is FIFO. We experimented with FIFO fetches conjoined with four of the eviction policies, MaxPop, MinPop, MaxEX and MinEX. Of these four possibilities MaxPop resulted in the best performance in almost all cases.

2.1.4 Scheduling Heuristics

Based on the SERPT execution policy, the six fetch policies and using an eviction policy matched to the fetch, we developed and extensively simulated five distinct scheduling algorithms. We denote them with the names of their respective fetch policies, PSERPT, MaxPop, MinPop, MaxEX and FIFO. Thus, PSERPT fetches data for the shortest jobs and evicts data for the longer jobs. MaxPop fetches the most popular data item and evicts the least popular, while MinPop does exactly the reverse. MaxEX follows the eviction policy of evicting the datum for which the total processing time of all tasks that require it is minimized. FIFO fetch is used in conjunction with MinPop eviction policy.

3 Simulation Methodology

We use simulation to evaluate the scheduling policies. The simulator is event driven and is illustrated in Figure 1. Queries arrive dynamically at the system. We simulate Poisson arrival rate. Once a query arrives, the simulator computes the set of runnable and non-runnable (idle) tasks for that query. Recall that we assume one task for each datum in the data set. Based on the simulation parameter selected for locality modeling (described below), some of the tasks may be determined to require zero processing time. These tasks are immediately placed into the completed task set. That is, those tasks are completed without having to access any of the data items.

We measure the flow time (*i.e.,* response time) for query processing. The average response time is the average time between when a query arrives at the system, and the time when the last task from that query has been processed. The processing time for each task is a random variable with exponential probability distribution. In this paper, all tasks from all queries share the same probability distribution. Thus, when a query arrives, the simulator computes the processing time (using a pseudo-random number generator) for each of the tasks of that query. Now that the processing times are known, the simulator updates the two heaps in the system, and (if necessary) preempts the currently running job. The runnable task heap implements a priority queue of the tasks whose data is currently in memory. The data fetch heap implements a priority

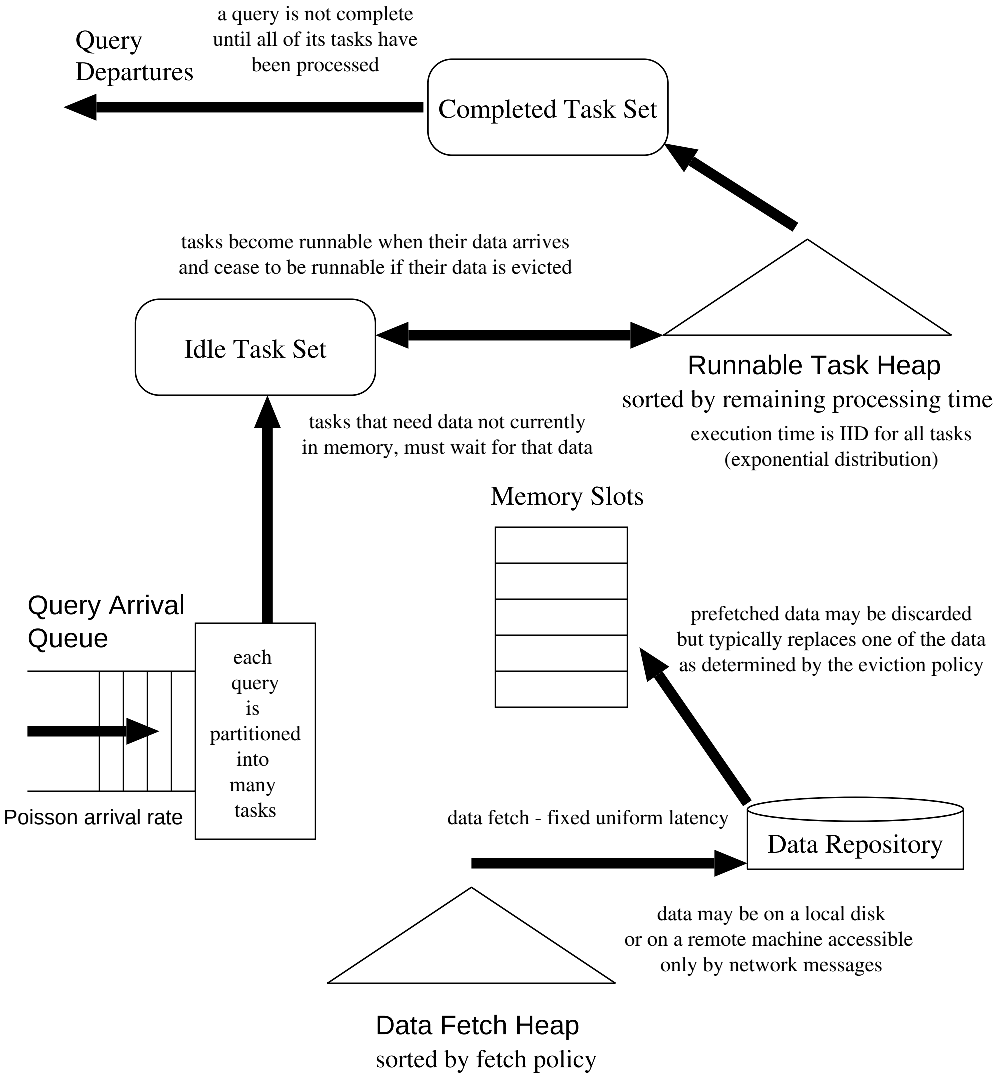

Figure 1: The structure of the simulation.

queue of the data items that are needed by tasks in the idle task set. The priority used to sort these heaps is given by the execute and fetch policies respectively.

We identify three important tunable parameters for the simulation. They are:

load The average number of queries being processed by the system at any time.

task granularity The ratio of the average time to process a runnable task to the time required to fetch a datum from disk.

locality The percentage of the total data set accessed by a single query.

3.1 System Load

We are interested in the steady state behavior of the system. Thus, we must choose the query interarrival times such that it is not so high that there is only one query present in the system at any given time. Further, it should not be so low that all queries arrive almost instantly very close to time 0 and then have to wait for a long time for the first data to arrive. We use the well known Little's result [2] from queuing theory : $N = \lambda * T$, where N is the average number of queries that can be handled by the system, signifying the average load, λ is the rate of arrival of queries and T is the average delay of a query in the system. For example, if the grain size is significantly small, the average delay of the query would be determined by the fetch speeds. We always run the simulation for a fixed number of queries (since we are interested in the response times, we have to run all the queries to completion). We fix N to be a fraction of the total number of queries and then find λ, which gives us an approximate estimate of the required interarrival time.

3.2 Grain Size

The granularity is an estimate of ratio of computation to communication performed during query processing. If queries are very coarse grained, then the data fetch time is insignificant. Hence, locality will be of little concern in selecting an optimal schedule. We vary the granularity over four orders of magnitude, ranging from data access latency 100 times that of the processing time to data access latency of 1/100th of the processing time. In our simulation, we choose to hold the processing time constant, and vary the data fetch latency, when measuring the effect of granularity.

3.3 Locality

Locality is the probability that a data item will be used by more than one query. We do not model locality directly. However, by varying the amount of data accessed by each query, we can control the locality indirectly. In each simulation we select a data access volume for each query. This value is the percentage of the entire database that the query will process. The data items that are accessed are uniformly distributed in the database. If the data access volume is very large, then the locality of data references will naturally be high, since most queries will be processing most of the database. By contrast, if the data access volume is very small, then the probability that any two queries will access the same datum is very small.

Scheduling characteristics are controlled by parameters listed in Table 1. Queries arrive at poisson rate. A memory size of 5 % the size of total data set is not unrealistic. Task processing times are exponentially distributed.

Parameter	Simulation Range	Comments
number of queries	fixed at 20	
data set size	fixed at 100	
size of memory	fixed at 5	
mean task processing time	fixed at 0.1	exponential
disk fetch time	0.001 to 10	
locality	10 to 100	
query interarrival time	chosen dynamically	Poisson distribution

Table 1: Simulator Parameters

4 Experimental Results

4.1 Response Time versus Grain Size

It is of interest to know how the various heuristics perform when the grain size is varied keeping the locality fixed. We measured the response times for processing times varying from 2 orders of magnitude above to 2 orders below the fetch time of a data item. This was done keeping the locality fixed at 50 %. At lower granularity, since the effective processing time of a query is determined by the slow fetch time, it is expected that any prefetch policy that tries to fetch data to satisfy the shortest task in the system will do reasonably well. This can be seen from Figure 2, which shows that at lower grain size, the relative performance of PSERPT over other heuristics is much higher. At the other end, when the fetch time is really fast compared to processing times, it is expected that since data is very likely to be available in memory soon after it is needed, a policy to satisfy the shorter queries first would again be the winner. This is validated by Figure 2.

4.2 Comparing PSERPT and MaxPop

MaxPop tries to fetch the data that will remain the longest in memory. However, by doing that it may be effectively blocking the shorter queries, since when their data arrives, they may not find an appropriate memory slot to overwrite. By doing away with this restriction, PSERPT aggressively tries to satisfy the shorter tasks and thus performs better than MaxPop over a larger range of grain size. For fine grain size, its performance is comparable to MaxPop, although at the cost of higher average number of evictions, (about 15 % more) compared to MaxPop and higher average number of fetches (about 5 % more).

4.3 Response Time versus Locality

In this experiment, we study the effect of locality on the response time. We keep the grain size fixed, with fetches being 10 times slower than processing times. By varying the locality, we are indirectly varying the probability that a required data will be found in memory, thus avoiding a disk access. Both MaxPop and PSERPT perform much better compared to the other heuristics. Figure 3 shows the performance of various heuristics. The following subsection provides a generalized explanation for the behavior of all heuristics.

4.4 Comparing MaxPop and other Heuristics

In almost all the experiments we ran, it was found that MaxPop performed better than the remaining Max/Min and FIFO heuristics. Further, its performance was quite comparable to PSERPT. To understand why this is so, consider a case of 100 % locality. By definition of MaxPop, the data item chosen by MaxPop would,

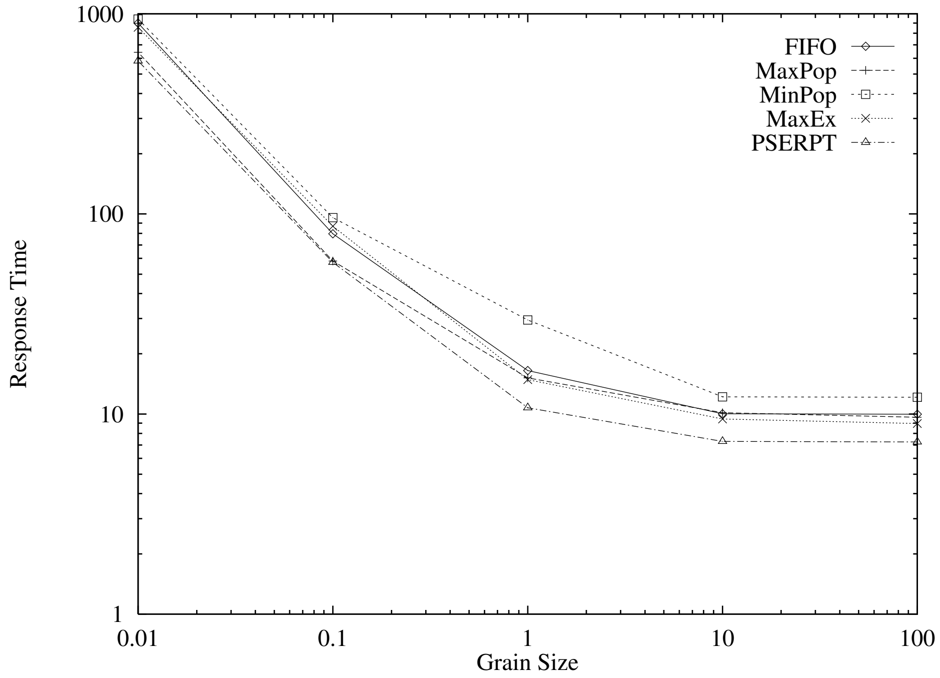

Figure 2: Response Time vs. Grain Size; Locality = 50

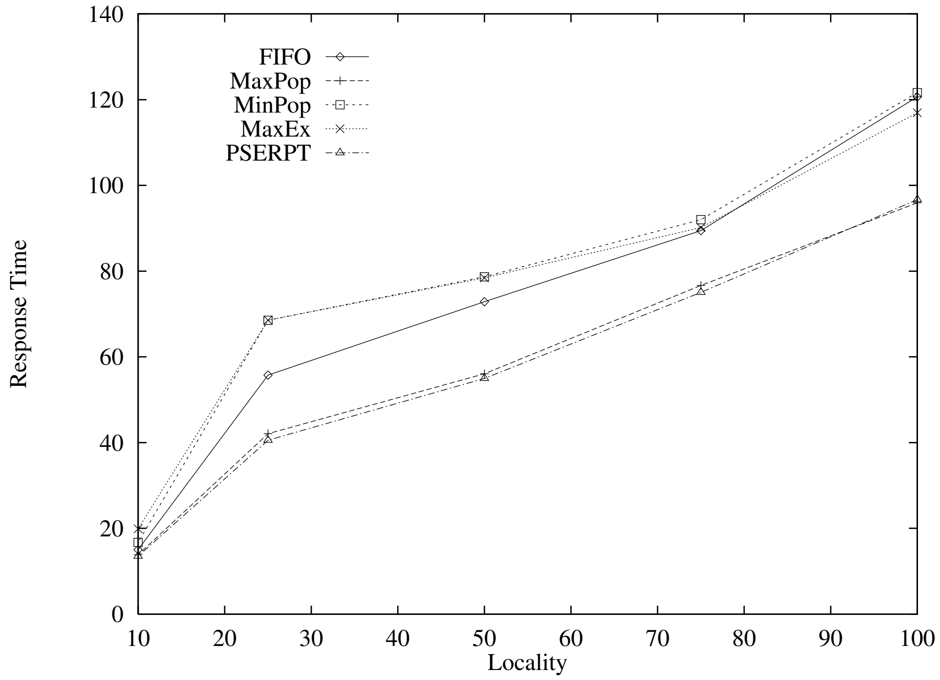

Figure 3: Response Time vs. Locality; Granularity = 0.1

in most cases, contain the shortest query in its list of waiting queries (the exception being when the query has already processed that data item). Thus, MaxPop is able to satisfy the shorter queries about as well as the PSERPT algorithm, whereas the other heuristics are inherently weak in this property. At small values of locality, one would expect this difference in performance to be insignificant. However, as can be seen from Figure 3, this is not necessarily true. The reason is that the other Max/Min and FIFO heuristics do not take full advantage of this reduction in locality.

4.5 Response Time versus Interarrival Time

Here, we study the effect of query interarrival time on the response time, while keeping the grain size fixed at 100 and locality fixed at 100 %. For high interarrival times, a newly arriving query is expected to be able to start execution almost immediately because the previous query is likely to have finished. Because of the fast fetches, a bad fetch or evict decision is amortized. It is therefore, not surprising to find from Figure 4 that almost all heuristics have comparable response times. However, at smaller interarrival times, since there can be more than one query in the system, it is clear from the preceding discussion why PSERPT and MaxPop perform better than others.

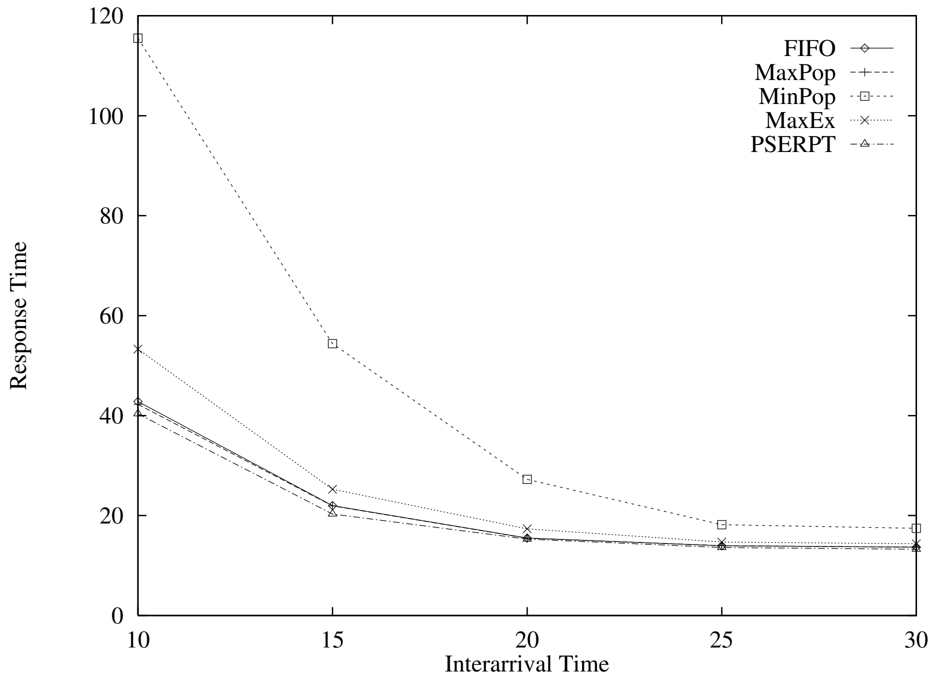

Figure 4: Response Time vs. Interarrival Time; Granularity = 100, Locality = 100

4.6 Effective Utilization of Bandwidth

In the case where fetch speed is comparable to the processing speed, we are interested in knowing how often does a new data item have to be discarded vis-a-vis the number of fetches. Although this metric may not have a direct impact on the response times, it nevertheless gives a good indication of the dynamics of the heuristics themselves. Figure 5 shows the results of this experiment, performed with a grain size of 1.0. An important observation is that heuristic MaxEx, which has been found to produce poor response times compared to MaxPop, does quite well in this evaluation. This is because MaxEx forces data items with greater total processing times, to stay in memory much longer, and thus minimizes the disk bandwidth.

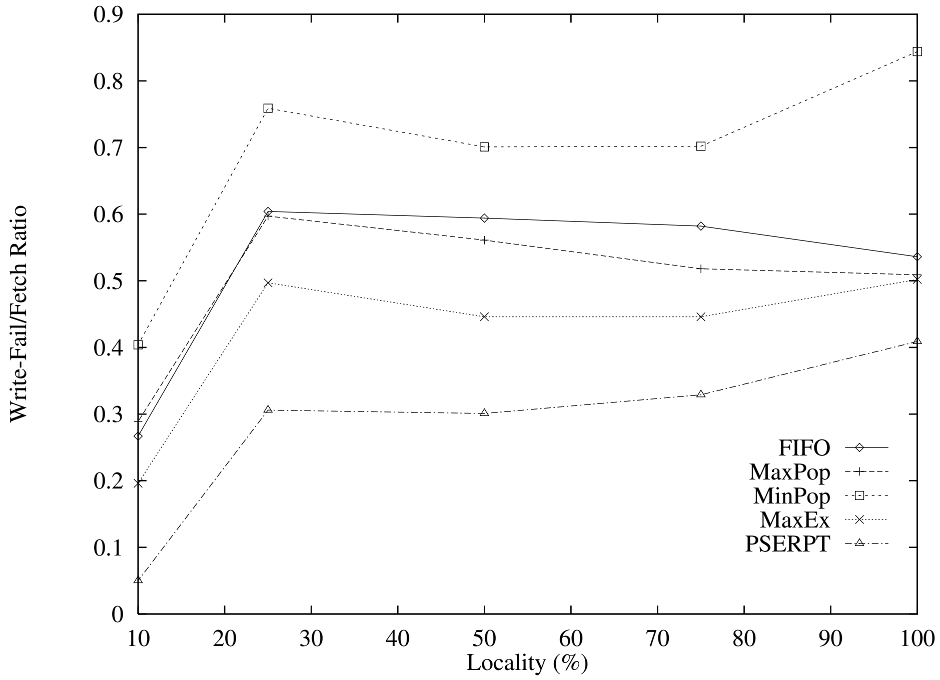

Figure 5: Write-Failures/Fetches ratio vs. Locality; Granularity = 1.0

5 Conclusion

In this paper, we have studied alternative heuristics for preemptive scheduling of queries accessing a read only data base. Two additional requirements for the system are that the data set accessed by various queries have a non zero intersection, and that the processing times are all exponentially distributed. Our goal is to minimize the average response time of all the queries, while exploiting data locality. Although there exists an algorithm, the shortest expected remaining processing time (SERPT), which is proven to produce an optimal schedule, it is not clear how well it would perform for different values of grain size and locality. We modified the traditional SERPT algorithm to do prefetching (PSERPT), and compared its performance with four other heuristics, which were designed based on different combinations of data fetch, eviction and task execution policies. Extensive simulation experiments show that the PSERPT consistently outperforms most of the other heuristics. The heuristic specifically designed to maximize data locality (MaxPop) does not result in significantly lower CPU idle time than the prefetched-SERPT policy. Similarly, the heuristic designed to minimize accesses to the disk (MaxEx) does not result in significantly fewer disk accesses than PSERPT. These results hold over a wide range of computation granularity (four orders of magnitude), inter-query data locality, and system load.

References

[1] Robert Abbot and Hector Garcia-Molina. Scheduling real-time transactions with disk resident data.

In *Proceeding of the Fifteenth International Conference on Very Large Data Bases*, pages 385–395, Amsterdam, 1989.

[2] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1987.

[3] Jacek Blazewicz. Selected topics in scheduling theory. *Annals of Discrete Mathematics*, 31:1–59, 1987.

[4] J. Bruno, E. G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17(7), July 1974.

[5] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *ACM, Multimedia'94*, October 1994.

[6] Asit Dan and Dinkar Sitaram. An online video placement policy based on bandwidth to space ratio. In *ACM, SIGMOD'95*, 1995.

[7] S. Ghandeharizadeh and C. Shahabi. On multimedia repositories, personal computers, and hierarchical storage systems. In *ACM, Multimedia'94*, October 1994.

[8] E.L. Lawler, J.K. Lenstra, and A. H. G. Rinnooy Kan. Recent developments in deterministic and stochastic scheduling: A survey. In M. A. H. Dempster, J. K. Lenstra, and A. G. H. Rinnooy Kan, editors, *Deterministic and Stochastic Scheduling : Proceedings of the NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*, volume 84, pages 35–73. D. Reidel Publishing Company, July 6-17 1982.

[9] M. Ogg and A. Ricciardi. Architecture Decisions for Wide-Area Applications. In *14th PODC*, page 259. ACM, 1995.

[10] Michael Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[11] Michale Pinedo. On the computational complexity of stochastic scheduling problems. In M. A. H. Dempster, J. K. Lenstra, and A. G. H. Rinnooy Kan, editors, *Deterministic and Stochastic Scheduling : Proceedings of the NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*, volume 84, pages 355–365. D. Reidel Publishing Company, July 6-17 1982.

[12] A. Ricciardi, M. Ogg, and E. Rothfus. The Nile System Architecture: Fault-tolerant Wide-area Access to Computing and Data Resources. In *6th Computing in High Energy Physics*, 1995. (to appear).

[13] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison Wesley, June 1994.

[14] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, pages 949–960, August 1987.