Scheduling in Real-Time Database Systems

S.B. Thorat
Director, Institute of Technology & Management, VIP Road, Nanded - 431602 (India), E-mail: suryakant_thorat@yahoo.com.

S.Y. Amdani
Asso. Prof., Dept of CSE & IT, B.N. College of Engg., Pusad-445215, (India), E-mail: salimamdani@yahoo.com, salim_amdani@rediffmail.com.

ABSTRACT: In many application areas data base management system has to operate under real-time constraints. We have taken integrated approach to explain algorithms for disk I/O scheduling. In all cases the algorithm directly address real-time constraints.

1. INTRODUCTION

A real-time database is a processing system designed to handle workloads whose state is constantly changing. This differs from traditional databases containing persistent data, mostly unaffected by time. For example, a stock market changes very rapidly and is dynamic. The graphs of the different markets appear to be very unstable and yet a database has to keep track of current values for all of the markets of the Stock Exchange. Real-time processing means that a transaction is processed fast enough for the result to come back and be acted on right away. Real-time databases are useful for accounting, banking, law, medical records, multi-media, process control, reservation systems, and scientific data analysis.

Database management systems provide tools for such organization, so in recent years there has been interest in merging database and real-time technology. The resulting integrated system, which provides database operations with real-time constraints is generally called a real-time database system (RTDBS).
Like a conventional database system, a RTDBS functions as a repository of data, provides efficient storage, and performs retrieval and manipulation of information. However, as a part of a real-time system, whose tasks are associated with time constraints, a RTDBS, has the added burden of ensuring some degree of confidence in meeting the system’s timing requirements.

In a real-time system, each computational entity e.g. a task, a process, a thread, or a truncation has a deadline when submitted to the system. These entities must be scheduled and processed in such a way that they are completed in their corresponding deadline. Real-time systems can be identified by two categories. In a hard real-time, missing a deadline may lead to a catastrophe, while in a soft real-time system, missing a deadline may only reduce the ‘value’ of the entity to the system.

Since performance criteria for real-time systems are quite different from that of conventional systems, there are many new and challenging issues raised in designing such a system. One of the challenging issues in real-time I/O scheduling. Because I/O devices are orders of magnitude slower than CPU speeds, the improvement of I/O efficiency is extremely important to the performance of a real-time system. This motivates our interest in examining the real-time disk scheduling problem.

2. TRANSACTION SCHEDULING
A major part of real-time system research concerns scheduling of jobs (of which transactions are one kind) in a multiprogramming environment.

Much of the work done on real-time job scheduling focuses mainly on CPU scheduling. Transaction scheduling, however, involves not only the CPU. In fact, due to the extensive data processing requirements of a database system, resources such as data, I/O, and memory are also subject to severe competition among concurrently running transactions. Careful scheduling the use of these resources is very important to the performance of RTDBSs.
In this section, we discuss some general issues of transaction scheduling.

Since most of the real-time scheduling protocols revolve around the use of priority, we will discuss how priority is assigned to transactions.

A transaction has many attributes that may affect its priority. Below is a list of those attributes that are most relevant to a RTDBS. The parenthesized variables next to each attribute represent the individual quantitative measure of each concept.

1. **Criticalness ($C$):** the more critical a transaction is, the higher is its priority. Sometimes, the criticalness of a transaction can be expressed as a value function.

2. **Deadline ($d$):** the earlier its deadline, the higher is the transaction’s priority.

3. **Amount of unfinished work ($l$):** a transaction with less amount of unfinished work may be given a higher priority than a transaction with large amount of unfinished work. In the extreme case when a transaction has begun its commit phase, its priority could be raised to a higher value. This enables a committing transaction, who requires minimal computation, to finish fast. Resources held by the committing transaction can thus be released sooner to reduce blocking of other transactions.

4. **Amount of computation already invested ($c$):** a transaction that already has a large amount of computation done may be given a higher priority. Preempting a transaction in a database system requires not only the release of resource but also careful rollback of the transaction. It is sometime easier and less wasteful of system resources to rollback a transaction that has only run for a short time.

5. **Age ($a$):** a transaction that arrived early should be given a higher priority than those that arrived late. This scheme reduces turnaround time and helps keep data externally consistent.
6. **Slackness (s):** slackness measures how long a transaction’s execution can be delayed while still making it possible to meet the transaction’s deadline.

If we denote the arrival time of a transaction by \( t_a \) then slackness can be expressed as:

\[
s = d - t_a - c - l
\]

The tighter the slackness of a transaction is, the higher should be its priority.

It is generally hard to capture the idea of urgency by only one of the items discussed above. Consequently, it is suggested that a combination be used to compute a priority value function \( pr() \). In particular, the following formula is suggested as an example:

\[
pr(T) = (w_1 a - w_2 d + w_3 c - w_4 l)
\]

where the \( w \)'s are weights reflecting the relative importance of the various factors.

We note that when priority computation is based on the amount of unfinished work and slackness, a good prediction of transaction execution time is needed.

As an attempt, we can generally decompose the execution time \( t_{exec} \) into three components as follows:

\[
t_{exec} = t_{fault} + t_{db} + t_{nondb}
\]

where \( t_{fault}, t_{db}, \) and \( t_{nondb} \) denote the times spent in page fault, data-processing operations, and non-data-processing operations respectively. We look at these terms one by one.

The term \( t_{fault} \) represents the amount of time spent in paging data from disk to memory. For periodic transactions, if data prefetching is possible, a memory resident database can be assumed. This removes any uncertainty on \( t_{fault} \) by essentially setting it to zero. Otherwise, \( t_{fault} \) includes all the time for I/O operations. Due to the wide gap between memory access time and disk access time, in a disk-based database, the use of a deterministic worst-case bound on \( t_{fault} \) is too pessimistic. A probabilistic model on estimating \( t_{fault} \) may be more effective in this case. Scheduling algorithms which
are based on execution time prediction, therefore, have to take into account the fact that the estimates are not precise.

The variable $t_{nondb}$ measures the execution time of non-database related operations while $t_{db}$ measures database related ones. It is generally harder to estimate $t_{db}$ than $t_{nondb}$. The reason being that the amount of data processing usually depends on the state of the database itself. Execution time on data processing is then estimated dynamically with the help of these metadata.

Due to the unpredictable job arrival pattern, conventional scheduling algorithms are usually on-line. That is, the order of transaction execution is not pre-computed. However, in RTDBS, if information about the transactions’ data access patterns, periodicities, deadlines etc. is available, transaction preanalysis should be carried out off-line. Transaction execution order is thus scheduled before transactions arrive. Since off-line schedulers are given more information, and sooner, they are more flexible and usually produce better schedules.

When there are concurrently running tasks in a system, there are potential conflicts on resource access. These resources include data, I/O, memory and others. When given a job, a conflict-avoidance scheduler detects and resolves conflicts among jobs over resources before the job is released for execution.

For conflict-avoidance be applicable, all resource requirements must be known in advance. A conflict-resolving scheduler, on the other hand, handles conflicts when they actually occur. A conflict-resolution protocol, for example, may decide that a resource requester aborts a resource holder, if it is determined that the requester has a higher priority over the resource. The penalty of using a runtime conflict resolution strategy is the uncertainty it introduces in transaction execution time.

3. NATURE OF THE DISK SCHEDULING PROBLEM

To service a disk request, several operations take place. First, the disk head must be moved to the appropriate cylinder (seek time). Then, the portion of the disk on which the disk page is stored must
be rotated until it is immediately under the disk head (*latency time*). Then, the disk page must be made to spin by the disk head (*transmission time*). The above components needed to service a disk request are illustrated in figure 1-2.

![Figure 1: Components of a Disk Access.](image)

Queues build up for each disk because the inter-arrival time of the disk requests can be smaller than the time required by the disk to service a disk request. Disk scheduling involves a careful examination of the pending disk requests to determine the most efficient way to service the disk requests.

The disk scheduling problem involves reordering the disk requests in the disk queue so that the disk requests will be serviced with the minimum mechanical motion by employing seek optimization and latency optimization. The disk scheduling problem can be reduced to the travelling salesman problem which is a classical graph theory problem known to be NP-complete. Thus, the disk scheduling problem is an NP-complete problem.

The desirable characteristics of the disk scheduling algorithms are maximizing throughput, being fair, minimizing the mean response time, and minimizing the variance of the response times (predictability). Traditionally, disk scheduling algorithms have
mainly been concerned with increasing the bandwidth utilization of the disks, by ordering disk requests so that the seek time is minimized.

Heuristic algorithms such as Shortest Seek Time First (SSTF), SCAN and Circular SCAN address this problem.

When real-time constraints are imposed on disk requests, minimizing seek time alone is not sufficient. Real-time algorithms should address both minimizing the seek time and satisfying the timing constraints. Existing real-time disk scheduling algorithms are discussed in the following section.

4. TAXONOMY OF EXISTING REAL-TIME DISK SCHEDULING ALGORITHMS

Close study of existing real-time disk scheduling algorithms depicted in figure. The classification is based on whether the scheduling algorithm handles the disk request mainly based on its position on the disk (the position is determined by the cylinder number and the sector number) or based on its deadline.

Each algorithm maintains one disk queue where incoming disk requests are stored.

Disk requests are serviced from the front of the queue (queue head). A newly-arrived disk request is inserted somewhere in the
disk queue. Normally, the disk queue is implemented as a double linked list. Each node of the double linked list represents one disk request. The double linked list is detailed in next section. These algorithms are detailed below. For each algorithm, the input, output, and its functionality are described in pseudo code.

4.1. Algorithm Earliest Deadline First

The Earliest Deadline First Algorithm, EDF, handles only Rdl disk requests. An Rdl disk request is a read disk request which has a soft deadline and can be lost. This algorithm attempts to service disk requests according to their deadlines. Thus, the disk request that has the earliest deadline in the disk queue is serviced first. This means that the disk requests are sorted according to their deadlines and stored in the disk queue. As an extension of the algorithm, when there is a tie (this means two or more disk requests have the same deadline), the algorithm sorts the disk requests in scan order according to their cylinder numbers. The algorithm is detailed below.

Algorithm EDF

Input
disk_request: the disk request to be scheduled.

Output
returns true if insertion is successful and false otherwise.

Function
1. Loop starting from the queue head and moving backward towards the queue tail.
2. If the deadline of disk_request is before the deadline of the current disk request, and disk_request can be inserted such that its deadline is not violated and the insertion does not result in the violation of the deadline of any other pending disk requests, then insert the disk_request, return true, and stop.
3. If it is not possible to insert the disk request, reject disk_request and return false.
4. Stop.
Complexity

Time Complexity: $O(n)$ node accesses are needed.

Space Complexity: $O(n)$ nodes are needed.

$n$ is the number of disk requests in the disk queue.

4.2. Algorithm SCAN – Real-Time 1

The SCAN – Real-Time 1 algorithm, SCAN-RT1, is a modification of the SCAN algorithm. This algorithm handles only Rd1 disk requests. An Rd1 disk request is a read disk request which has a soft deadline and can be lost. SCAN attempts to optimize the disk operation by servicing the disk requests in scan order. SCAN-RT1 attempts to honor the real-time constraints without excessively affecting the disk bandwidth utilization.

SCAN-RT1 is similar to the SCAN algorithm in optimizing the seek time.

However, it schedules disk requests in a manner that maximizes the number of disk requests that meet their deadline. This is accomplished as follows. When a new disk request arrives, it is inserted in the disk queue in the scan order, only if the insertion does not potentially violate the deadlines of the other pending disk requests and its own deadline. Otherwise, the disk request is appended to the end of the queue if it is expected that the disk request will be serviced before its deadline. A disk request that can not be inserted in scan order nor appended at the end of the queue is rejected and considered lost.

Algorithm SCAN-RT1

Input

disk_request: the disk request to be scheduled.

Output

returns true if insertion is successful and false otherwise.
Function
1. Loop starting from the queue tail in the disk queue and moving forward towards the queue head.
2. If the disk_request can be inserted in scan order such that its deadline is not violated and the insertion does not result in the violation of the deadline of any other pending disk requests, then insert the disk_request, return true, and stop.
3. If it is not possible to insert the disk request, reject disk_request and return false.
4. Stop.

Complexity
Time Complexity: \(O(n)\) node accesses are needed.
Space Complexity: \(O(n)\) nodes are needed.

\(n\) is number of disk requests in the disk queue.

4.3. Algorithm SCAN – Real-Time 2
The SCAN – Real-Time 2 algorithm, SCAN-RT2, is an extension of the algorithm SCAN-RT1 in order to handle both read and write disk requests. This algorithm handles only Rdl and Wnn disk requests. An Rdl disk request is a read disk request which has a soft deadline and can be lost. A Wnn disk request is a write disk request which has no deadline and can not be lost.

In this algorithm Rdl disk requests are handled exactly as Rdl disk requests in the algorithm SCAN-RT1. However, the write disk requests that are given to the data storage server have to be buffered in the main memory buffers of the data storage server. Thus, to avoid buffer overflow, a deadline for each write disk request is computed such that the buffer never overflows.

4.4. Computing the Deadline of Read and Write Disk Requests
The read disk requests have deadlines imposed by the data streams which are viewed by the users.
The Wnn disk request type has the characteristics listed below.

1. Any Wnn disk request to be written to the disk is already pre-stored in a main memory buffer pool.
2. A Wnn disk request has no real-time deadline. Therefore, it can afford longer delays than Rdl disk requests.
3. Although it does not have a deadline, a Wnn disk request can not be kept indefinitely in main memory due to the risk of loss in case of system crash or power failure. It has to be fulfilled some time in the future, to avoid buffer overflow. The longest possible duration that a write disk request can be put in hold is a system parameter.
4. A Wnn disk request can not be lost due to system load. Regardless of the system load, a Wnn disk request has to be fulfilled by the system at some point in time. Therefore, the deadline associated with the Wnn disk requests would have to be artificially computed by the data storage server based on the number of available buffers and the expected arrival rate of write disk requests.

The following formula is used to compute the deadline of a newly-arrived Wnn disk request.

\[
\text{deadline} = \text{present time} + \frac{\text{free buffer space}}{\text{arrival rate of Wnn disk requests}}
\]

4.5. Algorithm SCAN-RT2

**Input**

*disk_request*: the disk request to be scheduled.

**Output**

returns *true* if insertion is successful and *false* otherwise.

**Function**

1. Compute the deadline of *disk_request*.
2. If *disk_request* is a read disk request then.
3. Loop starting from the queue tail in the disk queue and moving forward towards the queue head.
4. If the disk_request can be inserted in scan order such that its deadline is not violated and the insertion does not result in the violation of the deadline of any other pending disk requests, then insert the disk_request, return true, and stop.
5. If it is not possible to insert the disk request, then reject disk_request, and return false.
6. Stop.
7. Else (disk_request is a write disk request).
8. Loop starting from the queue tail in the disk queue and moving forward towards the queue head.
9. If the disk_request can be inserted in scan order such that its deadline is not violated and the insertion does not result in the violation of the deadline of any other pending disk requests, then insert the disk_request, return true, and stop.
10. Loop starting from the queue tail in the disk queue and moving forward towards the queue head.
11. If disk_request can be inserted in scan order such that its deadline is not violated, then insert the disk_request.
12. If disk_request is not inserted then.
13. Append disk_request at the disk queue tail.
14. Loop starting from the queue head in the disk queue and moving backward towards the queue tail.
15. If the deadline of the current disk is violated.
16. Switch type of current disk request.
17. Case type Rdl:
18. Delete Rdl disk request from the disk queue and reject the Rdl disk request.
19. Case type Wnn:
20. Re-insert the Wnn disk request at the queue head as well as all other Wnn disk requests whose deadlines are violated.


22. Stop.

Complexity

Time Complexity: \(O(n^2)\) node accesses are needed.

Space Complexity: \(O(n)\) nodes are needed.

\(n\) is the number of disk requests in the disk queue.

5. CONCLUSION

Existing real-time disk scheduling algorithms are studied and taxonomized. It is shown that the existing algorithms handle only the Rdl and the Wnn disk request types. Other disk requests, namely Wdn, Rdn, Wdl, and Rnn, which are essential for data storage servers, can not be handled.

REFERENCES


