데이터 스트림 저장을 위한 순환버퍼 기법

신재진*, 유병섭*, 어상훈*, 이동욱*,배해영*
*인하대학교 컴퓨터정보공학과
e-mail:jaejyn81@hotmail.com

Circular Buffer Technique for Storing Data Stream

Jae-Jyn Shin*, Byeong-Seob You*, Sang-Hun Eo*,
Dong-Wook Lee*, Hea-Young Bae*
*Dept. of Computer Science and Information Engineering,
Inha University

요 약

본 논문은 데이터 스트림의 저장을 위한 순환버퍼 기법을 제안한다. 빠르고 많은 입력을 가지는 데이터 스트림의 처리를 위해 DSMS(Data Stream Management System)란 새로운 시스템에 대한 연구가 활발히 진행되고 있다. 현재 입력되고 있는 데이터 스트림과 과거에 발생했던 데이터 스트림을 동시에 검색하는 하이브리드 질의를 위해서는 데이터 스트림이 디스크에 저장되어야 한다. 그러나 데이터 스트림의 빠른 입력 속도와 메모리와 디스크 공간의 한계 때문에 저장된 데이터 스트림에 대한 질의보다는, 현재 입력되고 있는 데이터 스트림에 대한 질의에 대한 연구들이 주로 이루어졌다. 본 논문에서는 제안하는 순환버퍼는 데이터 스트림의 입력과 출력을 비동기적으로 빠르게 실행할 수 있다. 또한 입력되는 네트워크 패킷의 감소와 출력 시 디스크 I/O의 감소를 위하여 순환버퍼에서는 데이터 스트림의 묶음 단위로 입출력이 진행된다. 실험을 통하여 기술이 대량으로 입력되는 데이터 스트림을 빠르게 저장시킬 수 있다는 것을 보인다.

1. 서론

증권 정보 관리 프로그램, 센서 네트워킹, 웹 패킷 감시, 물류 관리 시스템 등의 응용들은 끊임없이 많은 양의 데이터를 생산하는 특징을 가진다[1,2]. 데이터 스트림이라고 불리는 이러한 응용들은 데이터의 갱신보다는 삽입이 주된 연산을 이룬다. 그러나 기존의 DBMS(Database Management System)은 데이터의 삽입보다는 갱신이나 검색에 초점을 두고 연구, 개발이 되어왔다. 따라서 기존의 DBMS에서 연구되었던 데이터 삽입, 갱신, 검색과 같은 연산들은 데이터 삽입이 주된 연산인 데이터 스트림에 맞추어 새롭게 정의되어져야 한다.

새롭게 정의 될 연산중에서 검색 연산의 경우 기존의 DBMS에서는 디스크에 있는 데이터만 질의에 이용하는 히스토리 질의에 대한 연구가 이루어져왔다[3]. 반면에 데이터 스트림 분야에서는 현재 입력되고 있는 데이터에 질의를 하는 연속질의에 대한 연구가 이루어져 왔다[4]. 그러나 데이터 스트림에서는 과거 데이터와 현재 데이터 모두에 대한 질의인 하이브리드 질의가 발생할 수 있다[5]. 과거 데이터를 검색하기 위해서는 데이터 스트림을 디스크에 저장하여야 한다. 그러나 데이터 스트림의 빠른 입력 속도와 메모리, 디스크의 저장 공간의 한계성 때문에 대부분의 데이터 스트림 연구들은 히스토리 질의나 하이브리드 질의보다는 연속 질의에 초점을 맞추고 있다. 다음 질의는 하이브리드 질의에 대한 예제이다.

- 최근 1시간의 고속도로 통행량이 어제 오후 1시부터 3까지 고속도로 통행량에 비하여 얼마나 증가하였는지 계산 하여라.

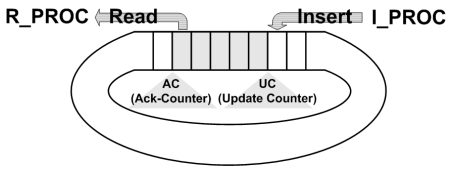
본 논문은 데이터 스트림의 빠른 저장을 위한 순

환버퍼 기법을 제안한다. 순환버퍼는 외부에서 데이터 스트림을 받아 디스크로 저장하는 역할을 한다. 또한 순환버퍼는 기존의 NBB 알고리즘을 확장한다. 이 때 입출력의 속도를 높이기 위해 입출력의 단위는 NBB처럼 개개의 레코드가 아니라 레코드 셋 단위로 이루어진다.

2장에서는 제안하는 기법의 관련연구를 설명한 후 3장에서 제안하는 기법인 BCB를 설명한다. 그리고 4절에서 제안된 시스템에 대한 성능평가를 한 후, 5장에서 결론과 향후연구에 관해 설명하겠다.

2. 관련연구: NBB(Non Blocking Buffer)

NBB는 이벤트 메시지의 전달을 위한 논블록킹 버퍼 알고리즘이다. NBB 알고리즘은 상태 메시지 전달을 위한 논블록킹 알고리즘인 NBW(Non Blocking Writer)에 기반을 두고 있다[6]. NBB를 이용하면 이벤트 송신자는 볼록킹 없이 메시지를 전달 할 수 있고, 메시지 수신자 또한 블록킹 없이 메시지를 수신할 수 있다. 또한 원형 버퍼를 이용하여 메시지 송신자는 대량의 메시지 전달을 가능하게 한다. [그림 1]는 NBB의 전체적인 구조를 나타낸다.



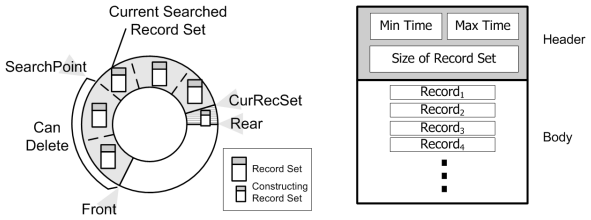
[그림 1] NBB의 구조

NBB는 한 개의 순환버퍼와 입력, 출력 지점을 가리키는 두 개의 변수로 이루어져 있다. *AC(Ack-Counter)*는 데이터가 읽혀지는 장소를 가리키는 변수이고 *UC(Update Counter)*는 데이터가 삽입되는 장소를 가리킨다. 이 두 변수로 인하여 NBB는 Insert 프로세스인 I_PROC와 Read 프로세스인 R_PROC, 두 개의 프로세스가 블록킹 없이 동시에 작동 할 수 있다.

3. 제안기법: BCB(Bulking Circular Buffer)

[그림 2]은 순환버퍼의 내부 데이터를 가리키는 포인터 변수들과 레코드 셋의 자료구조를 나타낸다. NBB와는 다르게 입력 시 네트워크 패킷의 수를 줄이고, 출력 시 디스크 I/O를 줄이기 위하여 입출력의 단위는 레코드 셋이 된다. 순환버퍼 안의 변수들은 순환버퍼에는 입력될 위치를 나타내는 *Rear*, 데이터 삭제와 검색 위치는 나타내는 *Front,* 현재 생성되고 있는 레코드 셋의 위치를 나타내는 *CurRecSet*, 그리도 현재 입력을 받아들여 레코드 셋을 생성하는 레코드 셋의 위치를 나타내는 *SearchPoint*가 있다.

레코드 셋의 헤더에는 레코드 셋이 가지고 있는 레코드들의 최소 생성시간과 최대 생성시간, 그리고 레코드 셋의 크기를 가지고 있다. 최대 생성시간과 최대 생성시간은 검색 질의를 빠르게 처리하기 위한 인덱스로 사용되고, 레코드 셋의 크기는 레코드가 입력되었을 때 완성된 레코드 셋으로 인정을 할 것인지 판단을 하기 위해 쓰인다.



(a) 순환버퍼의 포인터 변수들 (b) 레코드 셋

[그림 2] 순환버퍼와 레코드 셋의 구조

알고리즘 3은 순환버퍼에 레코드 셋이 입력되는 알고리즘, 알고리즘 6은 검색 알고리즘, 알고리즘 7는 플러시 알고리즘을 나타낸다.

레코드 셋의 삽입은 생성 중인 레코드 셋이라 불리는 *CurRecSet*이 가리키는 레코드 셋에서 처리 된다. 생성 중인 레코드 셋의 크기가 *min_recset_size* 이상이 되면 완성된 레코드 셋으로 인정이 되며 새로운 레코드 셋이 레코드의 입력을 받기 위하여 순환버퍼에 추가 된다. 또한 생성 중인 레코드 셋에 입력된 레코드 셋을 추가 할 경우 max_recset_size 보다 커도 새로운 레코드 셋이 생성된다.

검색은 *Front*가 가리키는 레코드 셋에서부터 *Rear*이 가리키는 레코드 셋까지 진행된다. 검색이 진행되지 않을 때는 *SearchPoint*는 항상 *CurRecSet*을 가리키고 있다. 검색이 시작되면 *SearchPoint*는 레코드 셋 검색의 시작을 알리는 *Front*로 지정되고, 하나의 레코드 셋에 대한 검색이 끝날 때마다 *SearchPoint*는 다음 레코드 셋으로 갱신된다.

플러시 도중엔 검색이 불가능 하지만 검색 도중에는 플러시가 가능하다. 플러시는 *Front*에서 시작되어 *CurRecSet*까지 진행된다. 하지만 현재 지우려는 레코드 셋을 *SearchPoint*가 가리키고 있으면

*SearchPoint*가 다음 레코드 셋을 가리킬 때까지 대기한다. 이런 방법으로 검색 도중에 플러시가 가능하게 하고 검색이 끝난 레코드 셋만 버퍼에서 삭제가 가능하게 할 수 있다.

레코드 입력 시에는 Front를 한 번 읽고 Rear 또는 Rear와 CurRecSet을 갱신한다. 또한 플러시 연산에서 호출되는 레코드 셋 삭제 시에는 CurRecSet를 한 번 읽고 Front를 갱신한다. 변수의 갱신이 갱신 도중 애매모호한 값으로 머물러 있지 않는 원자적 연산이라고 가정하면 레코드 입력과 레코드 셋 플러시는 서로 동시 수행이 가능하다. 이 점은 NBB에서 I_PROC와 R_PROC가 서로 AC와 UC, 두 변수를 공유하면서 갱신하지만 동시 수행이 가능한 것과 같은 원리이다.

```
Algorithm InsertToBlkBuffer ( record )
Input
record      :          블록 순환버퍼에 입력된 패턴 트리 레코드
Output
result      :          레코드가 성공적으로 입력되었는지 여부

Begin
01:       rec_size := get_record_size( record );
02:       cur_recblk_size := get_blk_size( CurRecBlk );
      //레코드를 입력했을 때 크기가 max_recblk_size를 넘는거나
     //또는 현재 생성 중인 레코드 블록의 크기가 min_recblk_size를 넘는다
     //새로운 레코드 블록을 생성
03:       if (cur_recblk_size + rec_size > max_recblk_size)
or (cur_recblk_size > min_recblk_size)
04:          create_block_header( blk_header );
05:          if CircularInsert( blk_header ) = FAIL
06:             result := FAIL;
07:             return result;
08:          end if
09:          if CircularInsert( record ) = FAIL
10:             result := FAIL;
11:             return result;
12:          end if
13:          CurRecBlk := Rear;
14:          adjust_cur_blk_header( record )
//기존의 레코드 블록에 데이터 스트림 추가
15:       else
16:          if CircularInsert( record ) = FAIL
17:             result := FAIL;
18:             return result;
19:          end if
20:          adjust_cur_blk_header( record );
21:       end if
22:       result := SUCCESS;
23:       return result;
End
```

```
Algorithm GetRecordFromBuffer ( time_range )
Input
time_range          :          검색할 레코드의 시간 범
Output
total_result_list   :          결과 레코드를 받아올 리스트
Variables
blk_header          :          탐색 중인 레코드 블록의 헤더
blk_size            :          탐색 중인 레코드 블록의 크기
result_list         :          중간 결과 레코드 리스트

Begin
01:       if is_flushing() = TRUE   //플러시 중이면
02:          wait_flush_finish()
03:       end if
04:       SearchPoint := Front;
05:       repeat   //SearchPoint가 Rear보다 작거나 같을 때까지 반복
06:          if CircularRead( SearchPoint, blk_header, BLK_HDR_SIZE ) =
FAIL
07:             result := FAIL;
08:             return result;
09:          end if
10:          blk_size := get_blk_size( blk_header );
11:          if CircularRead( SearchPoint, rec_blk, blk_size ) = FAIL
12:             result := FAIL;
13:             return result;
14:          end if
15:          result_list := get_record( rec_blk, time_range );
16:          process_history_query( rec_blk, hist_query, result_list );
17:          add_data( total_result_list, result_list );
18:          SearchPoint:= (SearchPoint+blk_size) % Buf_size;
19:       until
20:          SearchPoint <= CurRecBlk;
21:       SearchPoint := CurRecBlk;
22:       result := SUCCESS;
23:       return result;
End
```

```
Algorithm FlushBuffer ()
Input
없음
Output
result      :          플러시의 성공적인 수행 여부
                       디스크가 꽉 찼을 경우 FAIL을 반환

Begin
01:       repeat   //Front가 CurRecBlk보다 작을 때까지 반복
02:          if Front = SearchPoint;   //현재 검색되고 있는 레코드 블록이면
03:             wait_until( SearchPoint != Front );
04:          end if
05:          CircularRead( 0, blk_header, BLK_HDR_SIZE );
06:          blk_size := get_blk_size( blk_header );
07:          CircularRead( 0, rec_blk, blk_size );
08:          if DiskWrite( rec_blk ) = FAIL
09:             result := FAIL;
10:             return result;
11:          end if
12:          Front := Front + blk_size;
13:       until
14:          Front < CurRecBlk;
15:       result := SUCCESS;
16:       return result;
End
```

삽입은 언제나 검색, 플러시와 동시에 수행될 수 있어서 빠르게 수행된다. 그러나 검색과 플러시는 조건에 따라서 반-동시 수행이 가능하다.

4. 평가

본 논문에서 제안한 BCB는 C++로 개발이 되었으며, 시스템은 Intel Pentium 4 3.00GHz CPU, 3.24GB의 램을 가지고 있고 Microsoft Windows XP가 OS로 설치되어 있다.

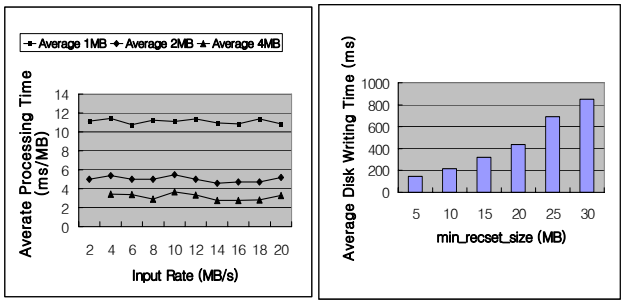
다양한 입력속도에 따라 일정한 크기의 데이터의 입력이 얼마나 빨리 처리되는가는 측정한다. 다음은 계산되는 총 입력시간, $avgProcessingTime$의 공식이다.

- $recsetSize_n$: n번째 레코드 셋의 크기
- $inputTime_n$: n번째 레코드 셋이 입력되는데 걸린 시간
- $avgProcessingTime = \sum_{n=1}^{30} recsetSize_n / \sum_{n=1}^{30} inputTime_n$

[그림 3]의 (a) 다양한 입력 속도에 따른 레코드 입력 처리 시간의 변화를 나타낸 그래프이다. 한 번에 입력되는 평균 레코드 셋의 크기를 1MB, 2MB, 4MB로 다양하게 하여 실험을 하였다. 순환버퍼 안의 레코드 셋의 최소 크기인 min_recset_size는 25M로 하였다 .

[그림 3]의 (a) 통하여 시스템이 입력되는 속도가 같으면 한 번에 입력되는 레코드 셋의 평균 크기가 클수록 입력 시간이 길게 나오는 것을 알 수 있다. 1초에 10M를 입력한다고 할 때 입력되는 레코드 셋의 평균 크기가 작으면, 더 많은 수 레코드 셋을 입력해야 한다. 따라서 입력에 대한 연산이 많아지기 때문이다. 또한 입력 처리는 입력 속도가 아니라 얼마나 많은 개수의 레코드 셋이 입력되는가에 따라 달라진다는 것을 알 수 있다.

[그림 3]의 (b)를 보면 min_recset_size 즉, 플러시 되는 단위가 30MB가 되더라도 디스크 입력시간은 1초를 넘기지 않는다. 따라서 입력되는 쪽에서 레코드를 셋 단위로 모아 초당 30MB 정도 입력시키면 제안하는 버퍼는 충분히 저장이 가능하다.



(a) MB 당 입력속도 (b) 디스크 입력 속도

[그림 3] 실험 결과

4. 평가 및 향후연구

본 논문은 빠르게 레코드를 저장할 수 있는 순환버퍼를 제안한다. 순환버퍼는 레코드 셋의 저장과 입력과 플러시의 동시성을 위하여 NBB를 확장한 순환버퍼로 구현된다. 이 순환버퍼는 벌크 연산으로 구현되어 검색이나 플러시보다 삽입의 성능을 최대화 할 수 있다. 제안 기법은 센서, RFID, 물류, 네트웍 이벤트 분석 등 빠르고 많은 양의 데이터를 저장할 필요가 있는 모든 분야에 적용이 가능하다.

제안되는 기법은 레코드 삽입의 성능의 최대화 할 수 있다. 그러나 입력받는 순환버퍼가 한 개면 다수의 스키마가 한 순환버퍼에 뒤섞일 수 있다. 그러나 순환버퍼는 시간 별 인덱스 밖에 지원을 해주지 않아서 원하는 스키마를 찾으려면 조건 절의 시간에 해당하는 모든 레코드를 검색해야 한다. 따라서 향후 여러 스키마를 고려하는 순환버퍼의 설계가 이루어 져야 한다.

참고문헌

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and Issues in Data Stream Systems” *PODS*, pp. 1-16, 2002.

[2] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World", In Proceedings of CIDR, 2003

[3] S. Chaudhuri and U. Dayal, "An Overview of Data Warehousing and OLAP Technology", In SIGMOD Record, 1997, Vol.26, No.1, pp.65-74

[4] S. Babu, and J. Widom, "Continuous Queries over Data Streams", ACM SIGMOD Record 2001, pp.109-120

[5] S. Chandrasekaran, and M. J. Franklin, "Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams", VLDB, 2004, pp.348-359

[6] K. H. Kim, "A Non-Blocking Buffer Mechanism for Real-Time Event Message Communication", Real-Time Systems - The International Journal of Time-Critical Computing Systems, 2006, Vol. 32, No. 3, pp. 197-211