Journal of Korea Multimedia Society Vol. 10, No. 11, November 2007(pp. 1384-1397)

하이브리드 질의를 위한 데이터 스트림 저장 기술

신재진[†], 유병섭[††], 어상훈[†††], 이동욱[††††], 배해영[†††††]

요 약

본 논문은 데이터 스트림의 하이브리드 질의를 위한 빠른 저장 방법을 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 스트림를 동시에 검색하는 하이브리드 질의를 위해서는 데이터 스트림이 디스크에 저장되어져야 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 데이터 스트림의 입력을 받을 때 순환버퍼를 이용하여 메모리 이용률을 최대화하고 블록킹 없는 데이터 스트림의 입력을 가능하게 한다. 또한 최대한 많은 양의 데이터를 디스크에 저장하기 위하여 디스크에 있는 데이터를 압축한다. 실험을 통하여 제안되는 기술이 대량으로 입력되는 데이터 스트림을 빠르게 저장시킬 수 있다는 것을 보인다.

Data Stream Storing Techniques for Supporting Hybrid Query

Jae-Jyn Shin[†], Byeong-Seob You[††], Sang-Hun Eo[†††],
Dong-Wook Lee[††††], Hae-Young Bae[†††††]

ABSTRACT

This paper proposes fast storage techniques for hybrid query of data streams. DSMS(Data Stream Management System) have been researched for processing data streams that have busting income. To process hybrid query that retrieve both current incoming data streams and past data streams data streams have to be stored into disk. But due to fast input speed of data stream and memory and disk space limitation, the main research is not about querying to stored data streams but about querying to current incoming data streams. Proposed techniques of this paper use circular buffer for maximizing memory utility and for make non blocking insertion possible. Data in a disk is compressed to maximize the number of data in the disk. Through experiences, proposed technique show that bursting insertion is stored fast.

Key words: Data Streams(데이터 스트림), Fast Insertion(빠른 저장), Buffer Management(하이브리드 질의)

1. 서 론

증권 정보 관리 프로그램, 네트워크 상태 감시, 센서 네트워킹, 웹 패킷 감시, 물류 관리 시스템 등의 응용들은 끊임없이 많은 양의 데이터를 생산하는 특징을 가진다[1-3]. 데이터 스트림이라고 불리는 이러

※ 교신저자(Corresponding Author) : 배해영, 주소 : 인천시 남구 용현동 253 인하대학교(402-751), 전화 : 032)860-8712, FAX : 032)862-9845, E-mail : jaejyn81@hotmail.com
접수일 : 2007년 3월 9일, 완료일 : 2007년 9월 28일
[†] 준회원, (주)한국공간정보통신
[††] 인하대학교 컴퓨터정보공학부
(E-mail : bsyou@dblab.inha.ac.kr)
[†††] 인하대학교 컴퓨터정보공학부
(E-mail : eosanghun@dblab.inha.ac.kr)
[††††] 준회원, 인하대학교 컴퓨터정보공학부
(E-mail : dwlee@dblab.inha.ac.kr)
[†††††] 정회원, 인하대학교 대학원장
(E-mail : hybae@inha.ac.kr)

한 응용들은 데이터의 갱신보다는 데이터의 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[4]. 반면에 데이터 스트림 분야에서는 현재 입력되고 있는 데이터에 질의를 하는 연속질의에 대한 연구가 이루어져 왔다[5]. 그러나 데이터 스트림에서는 과거 데이터와 현재 데이터 모두에 대한 질의인 하이브리드 질의가 발생할 수 있다[6]. 과거 데이터를 검색하기 위해서는 데이터 스트림을 디스크에 저장한다. 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.

하이브리드 질의 처리를 위해서 데이터 스트림을 가장 빠르게 저장할 수 있는 방법은 입력되는 데이터들을 샘플링 하는 방법이다[7,8]. 데이터 스트림의 샘플링 방법은 연속적으로 입력되는 임의로 데이터를 선택을 하여 선택되지 않은 데이터는 연산에서 제외시키는 방법이다. 그러나 입력되는 데이터 개개의 데이터가 중요한 정보를 가지고 있는 경우엔 임의로 데이터를 샘플링해서는 안 된다. 예를 들어 많은 기계들이 작동하는 공장에서의 로그 정보는 어느 기계에서 결함이 있는지 찾아주는 중요한 정보가 된다. 따라서 이러한 로그들을 임의로 샘플링하면 어떤 기계에서 결함이 있었는지 알 수 없게 될 수 있다.

OSCAR란 방법은 하이브리드 질의를 처리하는 효과적인 방법이다[6]. 이 방법은 입력된 데이터의 축소된 복제본을 만들어 놓아 디스크에 대한 접근 시 축소된 복제본만을 질의에 사용함으로서 디스크 I/O를 줄이는 효과가 있다. 그러나 어떻게 모든 데이터 스트림을 메모리에 받아서 디스크에 저장할 것인지에 대한 논의가 이루지지 않았다. 또한 축소된 복사본은 만들어 디스크에 입력하는 것은 또 다른 디스크 I/O를 증가시키며 질의 시에는 축소된 복사본만을 사용하므로 모든 데이터에 대한 질의가 이루어지는 것이 아니다.

데이터의 입력과 출력을 비동기적으로 하기 위하여 NBB(Non-Blocking Buffer)란 방법도 제안되어졌다. 이 방법은 기존의 상태 메시지의 전달을 위한 NBW(Non-Blocking whiter)를 다른 모든 동류의 이벤트 메시지 전송을 할 수 있게 확장한 방법이다[9,10]. NBB는 한 개의 순환버퍼와 두 개의 변수만으로 이루어진 간단하고 빠른 알고리즘이다. 그러나 NBB는 입출력되는 데이터가 일정한 크기이다. 그러나 데이터 스트림들은 많고 다양한 스키마를 가질 수 있으며 가변길이 레코드 또한 발생 할 수 있다. NBB를 곧바로 데이터 스트림에 적용하기에는 무리가 있다.

또한 데이터를 디스크에 빠르게 입력, 검색을 하기위해서 배치 I/O와 인덱스를 결합한 방법도 있다[11,12]. 그러나 데이터 스트림은 많은 양의 데이터의 입력을 가지게 되므로 데이터 입력 시마다 인덱스를 갱신하는 것은 연산 양의 증가를 가져온다. 따라서 빠른 검색을 위해 인덱스를 생성하는 것 또한 불필요한 연산의 양을 증가시키게 된다.

본 논문은 하이브리드 질의 처리를 위한 데이터 스트림의 빠른 저장 기술을 제안한다. 데이터 스트림의 빠른 입력을 위해 다수의 데이터를 묶어 한 번에 저장하는 대량 삽입 방법을 사용하고, 데이터 스트림의 입력은 많기 때문에 효율적인 메모리 사용과 버퍼의 비동기적인 삽입, 삭제를 위해 순환버퍼를 사용한다. 또한 다수의 스키마를 받아들이기 위하여 다수의 순환버퍼를 사용한다. 가변길이 레코드의 입력을 위하여 일정한 단위의 페이지가 디스크 입력으로 쓰이는 것이 아니라 일정 크기 이상이 완성된 레코드 셋만이 디스크에 쓰여 질 수 있다.

2장에서는 데이터 스트림의 저장을 위하여 제안하는 순환버퍼 방법과 압축 방법을 설명한다. 그리고 3절에서 제안된 시스템에 대한 성능평가를 한 후, 4장에서 결론과 향후연구에 관해 설명하겠다.

2. 데이터 스트림의 빠른 저장 기술

본 논문이 제안하는 데이터 스트림의 입력을 받을 순환버퍼는 NBB 알고리즘을 기반으로 하고 있다. 또한 디스크 입력 쓰레드와 압축 쓰레드간의 데이터 교환 또한 NBB 알고리즘을 기반으로 구현되었다. 따라서 2.1절에서 NBB 알고리즘을 설명하고 2.2에서는 제안하는 방법의 전체적인 특징과 그 특징들의 동기, 그리고 2.3, 2.4에서 순환버퍼와 압축방법에 관해 설명하겠다.

2.1 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[9,10]. NBB를 이용하면 이벤트 송신자는 볼록킹없이 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 1은 NBB의 전체적인 구조를 나타내고 알고리즘 1과 알고리즘 2는 NBB의 송수신 알고리즘이다.

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.

Insert 함수를 호출하는 I_PROC는 마지막에 갱신된 UC를 기억하여 입력될 데이터 X와 함께 Insert 함수의 매개변수에 입력한다. 만약 UC-buffer_size*2의 값이 AC 값과 같으면 디스크는 꽉 차있다고 판단되어 삽입 연산을 하지 못하게 된다. 꽉 차있지 않다면 UC를 1증가시키고 X를 순환버퍼에 삽입 후, 다시 UC를 1 증가 시킨다. 이렇게 해서 UC가 홀수일 때는 데이터가 삽입되고 있다는 사실을 다른 프로세스에 알린다.

또한 Read 함수를 호출하는 R_PROC는 마지막에 갱신된 AC를 기억하고 데이터를 담아올 변수 X와 최대로 데이터 삽입 완료를 기다릴 수 있는 시간인

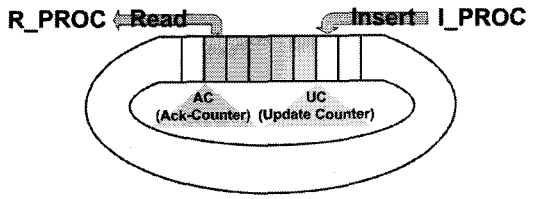


그림 1. NBB의 동작

Algorithm 1: Insert
Input: X: data to insert
lastUC: last updated Update Counter
Output:
procedure result
Procedure: Insert(X, lastUC)
1: tempAC := AC;
2: If lastUC - buffer_size*2 = tempAC
Buffer is full, return FAIL;
3: oldUC := UC;
4: UC := oldUC + 1;
5: insert X into the position pointed by (oldUC / 2 % buffer_size);
6: UC := oldUC + 2;
7: lastUC := oldUC + 2;
8: return SUCCESS;

Algorithm 2: Read
Input: X: memory space that will receive data
lastAC: last updated Ack-Counter
WT: limited processing time of procedure
Output:
procedure result
Procedure: Read(X, lastAC, WT)
1: tempUC := UC;
2: if tempUC = lastAC
Buffer is empty, return;
3: if tempUC - lastAC = 1
repeat tempUC := UC until tempUC = even number or time WT is passed ;
4: if time WT is passed
Insert fail, return;
5: read X from the position pointed by (lastAC / 2 % buffer_size);
6: oldAC := AC;
7: AC := oldAC + 2;
8: lastAC := oldAC +2;
9: return ;

WT를 매개변수로 하여 Read 함수를 호출한다. 만약 AC와 UP가 같으면 버퍼는 비어있다고 판단되어 Read는 실패 하게 된다. 그러나 비어있지 않을 경우 UC-AC가 1이면 데이터가 삽입 중인 것이라고 판단

하여 WT 시간만큼 데이터가 입력을 완료 할 때까지 기다린다. 기다리는 동안 AC를 읽어 AC가 짝수가 되면 데이터를 읽어 들이지만 WT 시간이 경과하면 데이터 삽입 도중 실패라고 판단하여 Read는 실패로 끝나게 된다. 옮바르게 데이터를 읽었으면 AC가 가리키는 장소에서 데이터를 X로 옮긴 후, AC를 2 증가시킨다.

2.2 FAST(FAst Storing Technique)의 특징

NBB는 수신 프로세스와 송신 프로세스가 이벤트 메시지를 주고받을 때 블록킹이 발생하지 않는다. 또한 순환버퍼를 사용하여 대량의 이벤트 메시지도 송수신이 가능하다. 이러한 특징은 대량의 데이터 스트림을 받는 버퍼 설계에 유용하게 쓰일 수 있다. 그림 2는 대량의 데이터 스트림을 저장 할 수 있는 FAST의 전체 구조를 나타낸다. 입력된 데이터 스트림은 스키마에 따라 다른 버퍼에 입력된다. 순환버퍼에 있던 데이터는 디스크에 있는 파일로 입력되고 일정한 크기 이상이 된 파일은 압축이 된다.

FAST는 다음과 같은 특징을 가진다.

가변 길이 레코드 입력을 위한 순환버퍼
다수의 스키마의 입력
레코드 셋 단위의 데이터 처리
시간 순서의 정렬
파일의 압축

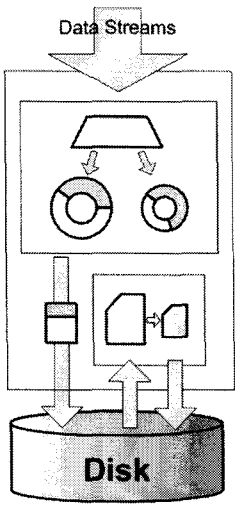
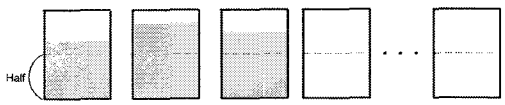


그림 2. FAST의 전체 구조

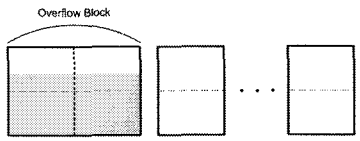
이 후 데이터는 레코드란 용어로 쓰이고, 레코드들의 집합은 레코드 셋이라고 표기한다.

FAST는 순환버퍼를 사용하여 쉽게 가변길이 레코드 입력을 받을 수 있다. 일정한 크기의 버퍼가 여러 개 존재하는 기존의 버퍼는 버퍼의 크기의 반보다 약간 큰 데이터가 여러 번 입력 될 경우 버퍼 공간의 낭비가 존재한다. 이러한 공간의 낭비는 많은 양의 레코드가 발생하는 데이터 스트림 환경에서는 큰 단점이 될 수 있다. 또한 버퍼 크기보다 큰 데이터가 입력될 경우 여러 개의 버퍼를 합쳐서 하나의 오버플로우 버퍼를 생성해야하는 연산의 부담이 있다. 그러나 순환버퍼는 데이터를 버퍼에 복사하고 간단이 Rear만 갱신하면 된다. 따라서 FAST에서는 가변길이 데이터의 입력을 쉽게 받을 수 있기 위하여 순환버퍼를 사용한다. 그림 3의 (a)와 (b)는 이러한 일정 크기 버퍼의 단점을 나타내고, (c)는 순환버퍼의 장점을 표현하고 있다.

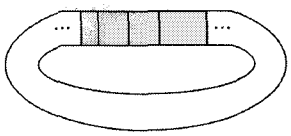
일반적인 메시지처리는 처리속도를 높이기 위해 고정길이 레코드를 이용한다. 그러나 데이터 스트림 환경은 무한히 많은 데이터가 존재하기 때문에 위와 같이 각각의 레코드에서 메모리가 조금씩 낭비됐을 경우 전체 메모리 낭비는 기하급수적으로 증가하게 된다. 따라서 데이터 스트림 환경에서와 같은 한정된



(a) buffer_size/2 보다 약간 큰 레코드들의 입력 시 고정크기버퍼의 상태



(b) buffer_size보다 큰 레코드의 입력 시 고정크기버퍼



(c) 다양한 크기의 레코드를 받아들이는 순환버퍼의 상태

그림 3. 버퍼의 가능한 구현 방법

메모리 버퍼를 가지는 환경에서 되도록 많은 양의 레코드를 저장하기 위해서는 위와 같은 버퍼의 낭비가 없는 가변길이 레코드 처리의 설계가 이루어져야 한다.

데이터 스트림의 대한 질의는 두 스키마에 대한 조인을 필요로 할 수 있다. 그러나 하나의 순환버퍼로만 데이터의 입력을 받으면 입력되는 모든 레코드는 한 순환버퍼에 섞이게 된다. 결국 질의 시에 한 순환버퍼 안에서 원하는 스키마의 레코드를 찾아야 하는 많은 연산의 부담이 있다. 따라서 FAST에서는 스키마의 개수에 따라 다수의 순환버퍼를 가진다. 그림 4은 다수의 스키마의 데이터 스트림을 받아들이는 다수의 순환버퍼를 나타낸 것이다. 만약 Switch에 원하는 스키마가 아니면 잘못 입력된 데이터 스트림으로 간주하여 단순히 삭제한다. 그러나 설정되어 있는 스키마의 데이터 스트림이 입력되면 Switch는 해당하는 순환버퍼로 레코드를 전해주게 된다.

FAST의 또 다른 특징인 레코드 셋 단위의 데이터 처리는 대량의 레코드 연산을 빠르게 해 준다. 데이터 스트림에 대한 질의는 크게 레코드 입력, 검색, 플러시로 나뉠 수 있다. 데이터 스트림은 많은 수의 입력을 가지기 때문에 레코드 별로 입력을 받으면 너무 많은 수의 네트워크 패킷이 발생한다. 따라서 데이터 스트림을 전송해주는 시스템에서 많은 수의 레코드를 묶어 레코드 셋으로 보내주면 입력 속도를 높일 수 있다. 또한 레코드를 셋 단위로 묶으면 검색 시 레코드 셋이 레코드들에 대한 인덱스를 제공 해주어서 검색을 빠르게 한다. 또한 플러시 시 레코드를 일일이 디스크에 쓰게 되면 막대한 디스크 I/O가 발생한다. 따라서 한 번 디스크 I/O가 입출력 크기 단위

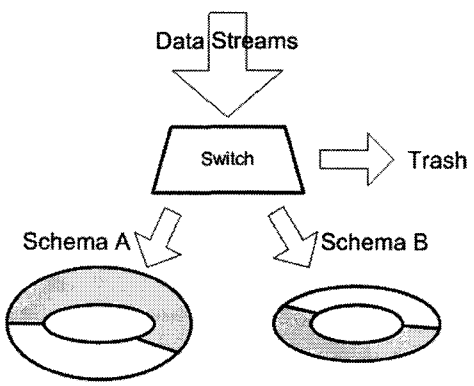


그림 4. FAST의 순환버퍼

인 블록 크기보다 같거나 큰 크기의 레코드 셋으로 레코드들을 묶어 디스크에 쓰면 디스크 입력 시간을 빠르게 할 수 있다.

그러한 레코드 셋을 생성하는 기준은 시간과 크기가 있다. 일정한 시간동안 모인 데이터를 한 레코드 셋으로 인정하면 각 레코드 셋들의 크기가 너무 크게 차이가 날 수 있다. 또한 레코드가 너무 느리게 입력되면 언제 레코드 셋으로 인정이 되어 연산이 되는지 알 수 없다. 그러나 레코드 셋의 단위를 크기로 한다면 고정크기버퍼와 같은 문제가 생길 수 있다. 따라서 FAST는 두 방법의 절충적인 방법으로 레코드 셋을 결정한다. 일정 크기 이상의 레코드가 모이면 레코드 셋이 완성된다. 이렇게 레코드 셋이 완성되기 위한 최소 크기를 min_recset_size라고 정의한다. min_recset_size의 크기만큼 레코드가 입력되지 않았더라도 일정 시간이 지난 레코드 셋은 한 레코드 셋으로 인정이 되어 생성을 완료 한다. 또한 레코드 셋의 지나친 크기 증가를 막기 위하여 최대 레코드 셋 크기인 max_recset_size를 정의한다.

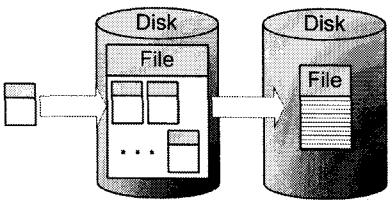
레코드 셋의 최대 크기 max_recset_size와 최소 크기 min_recset_size는 사용자에 의해서 정해진다. 레코드의 길이는 가변일 수 있기 때문에 min_recset_size는 max_recset_size가 너무 작으면 레코드 셋이 한 레코드를 담지 못할 수도 있다. 그러나 min_recset_size와 max_recset_size가 너무 크면 검색 인덱스로 쓰이는 레코드 셋의 개수가 줄어들게 된다. 따라서 min_recset_size와 max_recset_size는 DSSM가 적용되는 응용에 맞게 실험을 통하여 사용자가 알맞게 정의해 주어야 한다.

본 논문에서는 입력되는 레코드들은 각 레코드의 생성시간을 필드 정보로 가지고 있고, 이 생성시간의 순서대로 레코드가 입력된다고 가정한다. 따라서 입력되는 레코드들은 시간 순서대로 정렬이 되어 있기 때문에 레코드들을 묶는 레코드 셋들끼리도 또한 시간 순서대로 정렬이 되어있다. 결과적으로 그리고 다수의 레코드 셋을 담는 파일들끼리도 시간 순서대로 정렬이 되어 있다. 데이터 스트림은 끊임없이 입력기 때문에 데이터 스트림에 대해 질의를 하려면 반드시 시간에 대한 조건 연산이 질의에 포함되어야 한다 [1]. FAST의 레코드 셋과 파일들은 시간 순서대로 정렬이 되어 있기 때문에 검색질의에 빠르게 응답을 해 줄 수 있다. 한편 한 레코드에 생성시간이 표기되

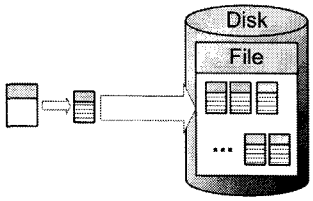
는 시점이 원격인가 로컬 시스템인가, 또는 시간에 의해 정렬되지 않은 레코드들의 입력에 대한 논의는 다른 논문에서 연구가 되어있다[13].

데이터 셋을 디스크에 저장하려면 많은 디스크 I/O가 발생한다. 또한 많은 데이터 스트림의 입력에 비해서 디스크 용량은 한정되어 있기 때문에 입력되는 모든 데이터 스트림을 디스크에 저장하지 못한다. 따라서 디스크 입력 I/O를 줄이고 되도록 많은 데이터를 디스크에 담기 위하여 데이터는 압축이 되어져야 한다.

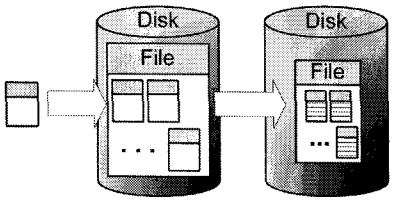
압축을 하는 기준은 세 가지가 있다. 첫 번째는 레코드 셋들을 파일에 입력을 하여 파일이 원하는 크기가 되면 통째로 압축을 하는 방법이다. 이 방법은 입력 시 압축에 의한 연산 부하가 없지만 레코드 셋의 크기가 클 경우 디스크 입력 I/O가 커지게 된다. 또한 파일을 검색 할 때 한 파일을 통째로 압축을 풀어 검색을 하여야 하기 때문에 검색이 지연되고 압축되지 않은 한 파일에 대한 디스크 공간의 낭비가 항상 발생하는 단점도 있다. 두 번째 방법은 완성된



(a) 파일 완성 후 통째로 압축



(b) 압축 후 파일에 입력



(c) 파일 완성 후 부분 별로 압축

그림 5. 가능한 압축 방법들

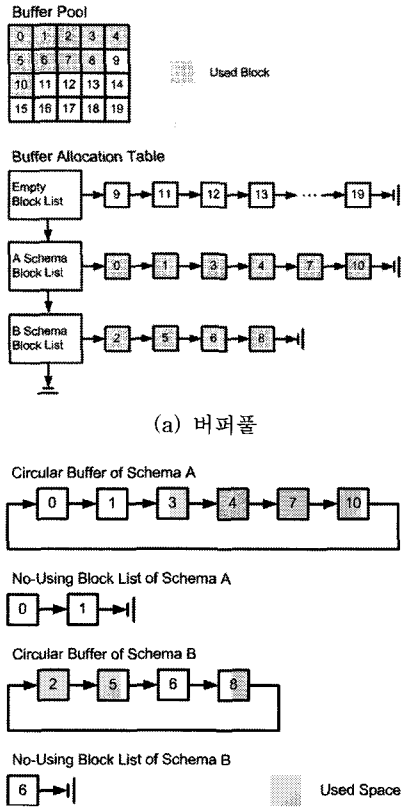
이 방법은 입력 시 압축 연산의 부하가 있지만 레코드 셋을 압축을 하여 파일에 입력하는 방법이다. 압축률이 좋을 경우 디스크 입력 I/O를 현저히 줄일 수 있다. 그러나 압축 시간이 많이 걸리면 입력 시간이 길어지는 단점이 있다. 세 번째 방법은 완성된 레코드 셋을 파일에 입력을 한 후, 파일이 원하는 크기가 되면 레코드 셋 별로 압축을 해서 새로운 파일을 생성하는 것이다. 입력 시 레코드 셋을 압축을 하지 않는다는 것은 첫 번째 방법과 유사하다. 그러나 한 파일을 나누어 압축을 하여 검색 시 원하는 레코드 셋만 압축을 풀면 되므로 검색의 속도를 높일 수 있다. FAST는 레코드 삽입 시 속도 저하가 없고 검색 속도 또한 느려지지 않는 세 번째 방법을 사용한다.

2.3 순환버퍼

순환버퍼들은 버퍼풀에서 필요한 버퍼를 받아온다. 버퍼풀은 시스템 구동 시 시스템에서 많은 양의 할당받아 필요시 개별 버퍼를 순환버퍼들에게 할당해주는 역할을 한다. 버퍼풀은 현재 버퍼가 어떤 스키마의 순환버퍼에 쓰이고 있는지를 버퍼 할당 테이블로 관리 한다. 사용자가 새로운 스키마의 입력을 설정해 주게 되면 버퍼풀에서 안 쓰는 블록을 받아와서 새로운 순환버퍼를 생성하게 된다.

만약 순환버퍼에 해당하는 스키마의 입력 속도가 빨라서 버퍼가 꽉 차게 될 것이라고 예상이 되면 버퍼풀에서 블록을 할당받아 순환버퍼를 재구성하게 된다. 또한 한 순환버퍼에는 현재 데이터가 저장되지 않는 블록의 정보를 리스트 형태로 가지고 있어서 일정 시간 이상으로 일정 개수의 블록이 사용되지 않으면 버퍼풀에 블록을 돌려주게 된다. 이렇게 순환버퍼는 버퍼풀에 블록을 할당, 반환함으로서 동적으로 크기가 관리된다. 그림 6은 버퍼풀과 버퍼풀에서 받아온 블록으로 순환버퍼를 구성한 모습을 나타낸다.

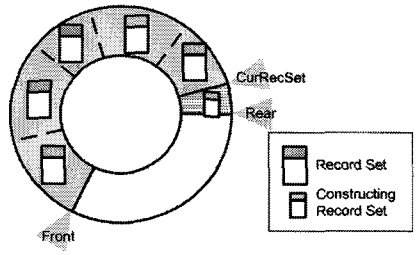
순환버퍼는 다수의 블록으로 구성이 되지만 내부적으로는 하나의 커다란 블록로 인식되어 동작한다. 그림 7은 순환버퍼의 내부 데이터를 가리키는 포인터 변수들과 레코드 셋의 자료구조를 나타낸다. 포인터 변수들은 순환버퍼에는 입력될 위치를 나타내는 Rear, 데이터 삭제와 검색 위치는 나타내는 Front, 현재 생성되고 있는 레코드 셋의 위치를 나타내는



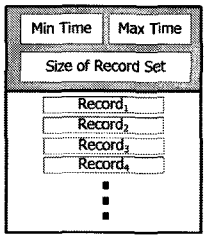
(a) 버퍼풀

(b) 블록들로 구성된 순환버퍼

그림 6. 버퍼풀과 순환버퍼의 구조



(a) 순환버퍼의 포인터 변수들



(b) DSSM에서의 레코드 셋

그림 7. 순환버퍼의 포인터 변수들과 레코드 셋의 구조

CurRecSet이 있다. CurRecSet이 가리키는 레코드 셋은 현재 레코드의 입력을 받아들이는 레코드 셋이다. 이 레코드 셋을 생성 중인 레코드 셋이라고 하며, 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. Front에서는 레코드 셋의 검색과 삭제가 이루어진다. 레코드 셋의 입력과 삭제의 위치를 다르게 둠으로써 입력과 삭제가 동시에 수행될 수 있는 장점을 가진다.

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.

알고리즘 3은 순환버퍼에 레코드가 입력되는 알고리즘을 나타내고, 알고리즘 6은 플러시 연산이 사용하는 레코드 셋 삭제 알고리즘을 나타낸다. 레코드 입력 시에는 Front를 한 번 읽고 Rear 또는 Rear와 CurRecSet을 갱신한다. 또한 레코드 셋의 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 유사하다.

레코드 셋에 대한 검색 연산은 레코드 입력과는 동시 수행 될 수 있지만 레코드 셋 삭제 연산과는 동시 수행 될 수 없다. 한 레코드 셋에 관해 검색을 처리하고 있는 동안 그 레코드 셋에 대한 삭제는 불가능하고, 그 반대로도 불가능하기 때문이다. 그러나 모든 레코드 셋을 삭제하는 플러시 연산과 검색 연산은 조건에 따라 동시에 수행 될 수도 있고 아닐 수도 있다. 플러시 연산중에는 검색 연산은 절대 발생될 수 없다. 그러나 검색 연산중에는 검색이 완료된 레코드 셋에 대한 플러시는 가능하다. 이러한 점을 적용한 순환버퍼의 자료 구조는 그림 8과 같이 다시 정의 될 수 있다.

Algorithm 3: Insert
Input: **record_set**: record set to insert
recset_size: size of the record set
Output:
procedure result
Procedure: Insert(record_set, recset_size)
1: **if** cur_recset_size + recset_size > max_recset_size
result := **InsertToNewRecSet (record_set, recset_size);**
return result;
2: **if** cur_recset_size > min_recset_size
result := **InsertToNewRecSet (record_set, recset_size);**
return result;
3: result := **InsertToCurRecSet (record_set, recset_size);**
4: return result;

Algorithm 4: InsertToNewRecSet
Input: **record_set**: record set to insert
recset_size: size of the record set
Output:
procedure result
Procedure: InsertToNewRecSet(record_set, recset_size)
1: new_rear := (Rear +recset_header_size + recset_size) % buffer_size;
2: **if** new_rear > Front
buffer is full, insert fail;
return FAIL;
3: Make Record Set Header;
4: Insert body of record_set;
5: CurRecSet := Rear;
6: Rear := new_rear ;
7: return SUCCESS;

Algorithm 5: InsertToCurRecSet
Input: record_set: record set to insert
record_size: size of the record set
Output:
procedure result
Procedure: InsertToCurRecSet(record_set, record_size)

1: new_rear := (Rear + record_size) % buffer_size;
2: **if** new_rear > Front
buffer is full, insert fail;
return FAIL;
3: Insert body of record_set;
4: Rear := new_rear ;
5: return SUCCESS;

Algorithm 6: **DeleteOneRecSet**
Input: **no**
Output:
procedure result
Procedure: **DeleteOneRecSet()**

1: recset_size := size of record set pointed by Front;
2: new_front := (Front + recset_size) % buffer_size;
3: **if** new_front > CurRecSet
No Record Set to Delete. Delete Fail;
4: Front := new_front
5: return SUCCESS;

검색은 Front가 가리키는 레코드 셋에서부터 CurRecSet이 가리키는 레코드 셋까지 진행된다. SearchPoint는 현재 검색을 진행하고 있는 레코드 셋을 가리키는 주소이다. 검색이 진행되지 않을 때는 SearchPoint는 항상 CurRecSet을 가리키고 있다. 검색이 시작되면 SearchPoint는 Front로 지정되고, 하나의 레코드 셋에 대한 검색이 끝날 때마다 SearchPoint는 다음 레코드 셋으로 갱신된다.

플러시 도중엔 검색이 불가능 하지만 검색 도중에는 플러시가 가능하다. 플러시는 Front에서 시작되어 CurRecSet까지 진행된다. 하지만 현재 지우려는 레코드 셋을 SearchPoint가 가리키고 있으면 SearchPoint가 다음 레코드 셋을 가리킬 때까지 대기한다. 이런 방법으로 검색 도중에 플러시가 가능하게 하고 검색이 끝난 레코드 셋만 버퍼에서 삭제가 가능하게 할 수 있다.

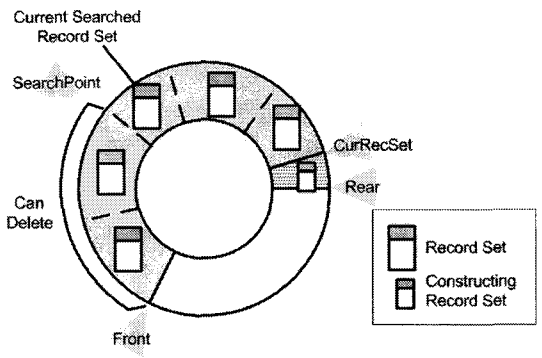


그림 8. 반-동시 검색/삭제를 고려한 순환버퍼

알고리즘 7과 8은 레코드 셋 검색과 플러시 연산에 대한 알고리즘을 나타내고 있다. 만약 플러시가 진행 중이면 검색 연산은 실패가 된다. 그러나 플러시 연산의 경우는 검색이 진행되는 도중에도 진행될 수 있다. 하지만 완전히 동시에 수행이 되는 것은 아니고 검색이 완료된 레코드 셋만이 플러시를 시킬 수 있다.

```
Algorithm 7: Select
Input: query: select query statement
Output:
      procedure result
Procedure: Select( query )
1: if it is flushing;
     Selection Fail;
     return FAIL;
2: SearchPoint := Front;
3: repeat
     recset := record set pointed by
     SearchPoint;
     query to recset;
     nextSearchPoint := (SearchPoint+size of
     recset) % buffer_size;
     SearchPoint := nextSearchPoint;
4: until
     SearchPoint <= Rear;
5: return SUCCESS;
```

```
Algorithm 8: FlushBuffer
Input: no
Output:
      procedure result
Procedure: FlushBuffer()
1: recset_count := the number of record set in
   DSB;
2: repeat
3: if Front = SearchPoint;
       Block until SearchPoint != Front;
     end if
     recset := record set pointed by Front;
     flush recset;
     DeleteOneRecSet();
4: until
     Front < CurRecSet;
5: return SUCCESS;
```

그림 9는 연산들의 동시 수행 여부를 나타낸 것이다. 앞에서 설명한 바와 같이 검색과 플러시 연산은 반-동시 수행이 가능하다. 그러나 데이터 입력은 어떠한 연산하고도 동시에 수행이 가능하다.

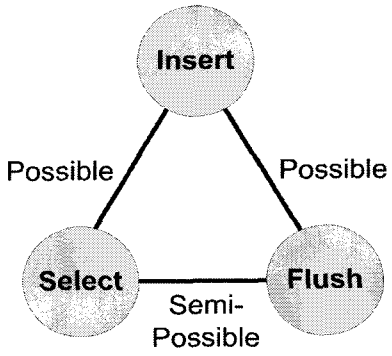


그림 9. 연산들 간의 동시성 여부

2.4 압축 알고리즘

데이터 스트림의 입력은 무한하고 크기도 방대하다. 따라서 입력되는 모든 데이터를 디스크에 저장하면 디스크는 곧 꽉 차게 된다. 따라서 한 파일에 데이터를 넣어 파일이 원하는 크기 이상이 되면 파일을 압축을 하여 디스크 공간을 절약하여야 한다.

순환버퍼에서 저장 될 레코드 셋들은 파일로 합쳐진다. 파일은 인덱스 부분, 데이터 부분으로 나뉘어져 있다. 인덱스 부분은 한 레코드 셋의 오프셋과 크기와 시간 범위를 가지고 있다. 레코드 셋에 있는 레코드들은 시간 순서대로 정렬되어 있고 따라서 파일 안에 있는 레코드 셋들도 시간 순서대로 정렬되어 있다. 결과적으로 각 파일들도 시간 순서대로 정렬이 되게 된다. 검색 요청이 있을 시 원하는 시간 범위에 해당하는 파일을 찾아 해당하는 파일안의 인덱스 부분만을 파일에서 읽어서 원하는 레코드 셋 한 개를 쉽게 읽을 수 있다.

파일이 일정 크기 이상으로 만들어지면 생성을 중단하고 새로운 파일의 구축을 시작한다. 이때 구축이 완료된 파일은 압축 쓰레드가 압축을 시작한다. 2.2절에서 언급한대로 파일을 레코드 셋 별로 나누어 압축한다. 그 이유는 통째로 압축하면 파일 검색 시 다시 통째로 압축을 풀어야 하기 때문에 검색 성능이 떨어지고, 압축을 한 후 파일에 입력하면 압축 알고리즘의 속도가 너무 느려서 입력 속도가 떨어지기 때문이다. 압축을 할 때 파일의 헤더와 인덱스 부분은 압축을 하지 않는다. 헤더와 인덱스 부분은 레코드 셋 부분에 비하여 크기가 상대적으로 매우 작고, 검색 시 인덱스 부분만을 읽어 원하는 압축된 레코드 셋을 읽어내어 압축을 풀고 검색을 할 수 있기 때문

이다. 그림 10은 레코드 셋을 디스크에 입력하고 완성된 파일을 압축하는 과정을 나타낸 것이다.

파일에 레코드 셋을 입력하여 파일이 최소 크기가 넘으면 CompleteFileList에 입력을 한다. 압축 쓰레드에서는 CompleteFileList에 파일이 있으면 압축을 실행한다. 이 두 과정은 각기 다른 쓰레드에서 실행이 된다. 또한 한 쓰레드는 쓰기만 하고 다른 쓰레드는 읽기만 한다. 이 과정은 앞에서 설명한 NBB (Non-Blocking Buffer)의 경우와 비슷하여 CompleteFileList는 NBB로 구현이 가능하다. 알고리즘 9는 파일에 레코드 셋을 쓰는 알고리즘이고 알고리즘 10은 압축 쓰레드의 알고리즘이다. CompleteFileList에서 읽고 쓰는 과정은 NBB에서 읽고 쓰는 과정과 유사하게 구현될 수 있다.

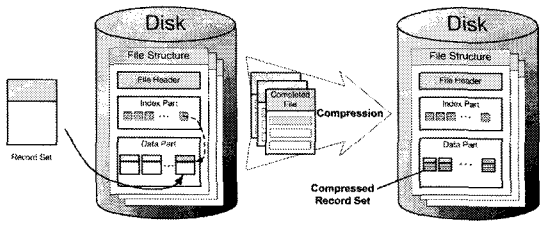


그림 10. FAST의 파일 압축

Algorithm 9: Write
Input: record_set: record set to write
Output:
record set to write into disk
Procedure: Write(record_set)
1: **if** The size of current_file > min_file_size
insert current_file into CompleteFileList;
current_file := make new file;
end if
2: insert record_set into data_part of current_file;
3: insert one index into index_part of current_file;
4: modify time information in file_header of current_file;
5: return SUCCESS;

Algorithm 10: CompressionThread
Input: no
Output:
procedure result
Procedure: CompressionThread()
1: completed_file := read one file from CompleteFileList;
2: compress each record set in completed_file;
3: return SUCCESS;

3. 성능평가

본 논문에서 제안한 FAST는 C++로 개발이 되었으며, 시스템은 Intel Pentium 4 3.00GHz CPU, 3.24GB의 램을 가지고 있고 Microsoft Windows XP가 OS로 설치되어 있다. 연산은 삽입, 검색, 플러시로 이루어져 있다. 따라서 이 세 가지에 대한 테스트를 진행한다.

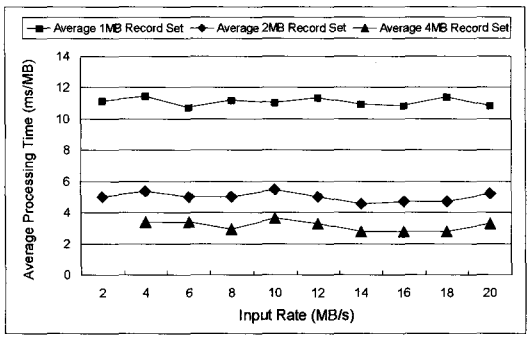
삽입 테스트는 다양한 입력속도에 따라 일정한 크기의 데이터가 얼마나 빨리 처리되는가는 측정한다. 입력 량은 1초당 2MB에서 20MB로 다양하게 조절된다. 측정되는 입력시간은 30초 동안 레코드 셋들이 입력되는 시간을 측정하고 그 시간의 합과 총 입력양의 계산으로 구해낸다. 다음은 계산되는 총 입력시간, $avgProcessingTime$의 공식이다.

$recsetSize_n$: n번째 레코드 셋의 크기
$inputTime_n$: n번째 레코드 셋이 입력되는데 걸린 시간

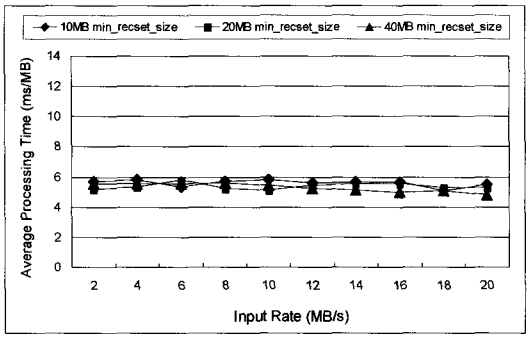
$$avgProcessingTime = \sum_{n=1}^{30} recsetSize_n \Big/ \sum_{n=1}^{30} inputTime_n$$

버퍼풀은 총 1GB의 버퍼를 가지며 파일 압축의 구현은 일반적으로 쓰이는 압축 라이브러리인 zLibrary를 사용하였다. 레코드는 평균 100Byte의 가변길이를 가지며 값은 압축 효율이 5%, 압축 속도는 10MB/s가 되도록 설계를 하였다. 데이터의 입력은 레코드 셋 단위로 이루어지며 이러한 레코드 셋의 입력이 30초 동안 이루어진다. 그림 11은 다양한 입력 속도에 따른 레코드 입력 시간의 변화를 나타낸 그래프이다. 그림 11의 (a)는 한 번에 입력되는 평균 레코드 셋의 크기를 1MB, 2MB, 4MB로 다양하게 하여 실험을 한 것이고 (b)는 순환버퍼 안의 최소 레코드 셋 크기인 min_recset_size를 다양하게 하여 실험을 한 것이다.

(a)를 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 많아지고 결과적으로

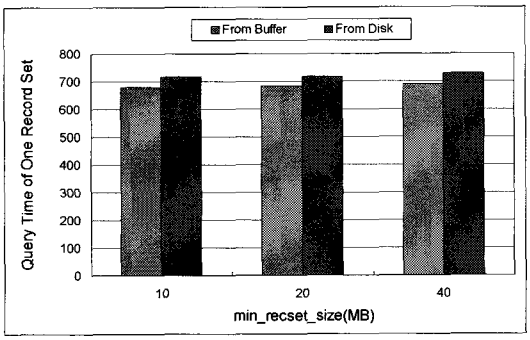


(a)

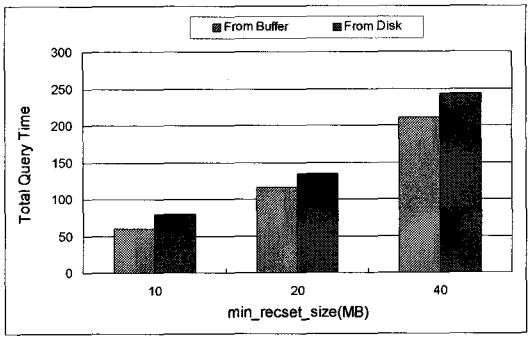


(b)

그림 11. 입력 속도의 변화



(a)



(b)

그림 12. 질의 시간의 변화

(a)와 같은 결과가 나온다. 또한 레코드 셋 입력 처리 속도는 입력 비율이 아니라 얼마나 많은 개수의 레코드 셋이 입력되는가에 따라 달라진다는 것을 알 수 있다. 평균 4MB의 레코드 셋 입력 시에 그래프가 중간에서부터 시작되는 것은 초당 2MB/s의 레코드 입력이 이루어 질 수 없기 때문이다. (b)에서는 입력되는 평균 레코드 셋 크기를 2MB로 고정하여 실험을 하였다. 결과는 min_recset_size에 상관없이 일정한 입력 시간을 보였다. 그 이유는 입력을 할 때는 입력되는 크기하고는 상관없이 CurRecSet과 Rear 두 변수의 갱신 말고는 다른 연산이 없기 때문이다.

레코드 검색은 다양한 min_recset_size에 따라서 변화하는 0.01% 검색 속도를 측정하였다. From Buffer는 버퍼에서 검색된 결과이고 From Disk는 데스크에서 검색된 결과이다. 그림 12의 (a)는 200MB의 입력을 주었을 때 버퍼 또는 디스크에서 한 레코드 셋을 가져와 검색하는데 걸리는 시간을 측정한 것이다. 데이터의 양이 똑같으면 min_recset_size의 변화에 상관없이 비슷한 검색 속도를 보인다. 그러나 디스크에서만 검색을 할 경우엔 버퍼에서만 검색을 할 경우보다 약간 많은 시간이 소모되었다. (b)는 한 레코드 셋의 검색 속도 변화이다. min_recset_size가 클수록 많은 양의 레코드를 가지고 있으므로, 한 레코드 셋의 검색 시간은 min_recset_size에 비례하여 증가하였다.

플러시 속도는 min_recset_size의 크기에 따라 달라지는 압축 속도와 디스크 입력 시간을 측정한다. 레코드 값의 변화를 이용하여 초당 압축 속도를 5MB/s로, 압축률을 5%로 고정하였다. 그림 13과 같이 실험 결과 레코드 셋의 압축 속도보다 압축된 레코드 셋의 디스크 입력 시간이 훨씬 빨랐다. 2.2절에서 설명한 압축의 두 번째 방법, 레코드 셋을 압축 후 디스크에 저장하는 방법을 사용하면 플러시 시간은 전적으로 압축 알고리즘에 의하여 결정된다는 것을 알 수 있었다. 따라서 압축 방법은 파일 완성 후 레코드 셋 별로 따로 압축하여 새로운 파일을 생성하는 압축의 세 번째 방법을 선택하는 것이 옳다.

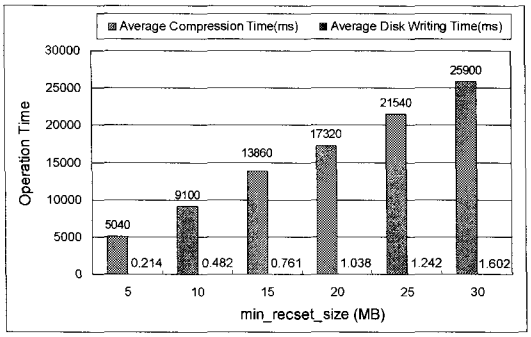


그림 13. 압축 시간과 디스크 입력 시간의 비교

제안 방법은 하이브리드질의를 처리하기 위하여 데이터 스트림을 저장하는 OSCAR와 제안방법의 삽입, 검색 속도를 비교한다. 제안 방법은 min_recset_size는 20MByte, max_recset_size는 40MByte로 하고 버퍼 풀의 크기는 500MByte로 하였다. 두 시스템 모두 입력은 레코드 블록 단위로 받으며 입력되는 한 블록의 크기는 5MByte로 고정한다. 또한 한 레코드의 크기는 평균 100Byte의 가변 길이를 가진다. 그림 14는 이러한 환경에서 총 200MByte의 입력이 발생하였을 때 OSCAR와 제안 방법의 입력 시간 비교를 나타낸 것이다.

실험 결과 OSCAR의 입력속도는 제안 방법보다 최대 20배 이상 차이가 났다. 그 이유는 OSCAR는 데이터가 입력될 때 마다 각 레코드를 현재 입력을 받는 런의 어느 블록에 삽입할 것인지 임의로 선택을 하게 된다. 샘플링 비율이 0%라더라도 이러한 샘플링을 고려한 레코드 단위의 연산은 많은 연산 부하를 가져온다. 그러나 제안 방법은 데이터의 입력을 데이터 블록 단위로 하기 때문에 입력이 블록단위로 발생하면 OSCAR보다 빠른 입력 처리가 가능하다. 따라서 입력되는 블록의 평균 크기를 늘릴수록 그 크기에 반비례하여 입력속도는 빨라진다. 이 이유는 블록의 크기에는 상관없이 한 블록의 입력이 발생 할 때마다 순환버퍼에는 단지 Currecset과 Rear의 갱신만 일어나기 때문이다.

검색은 200MByte의 입력을 주고 검색되는 데이터의 양을 다양하게 하여 결과 레코드가 얻어지는 첫 시간과 마지막 시간을 측정한다. OSCAR의 구현 방법은 데이터 입력 실험 때와 동일하다. 제안 방법의 min_recset_size는 10MByte, min_recset_size는 15MByt로 일정하게 한다.

그림 15에 나타나는 것과 모든 결과 레코드를 가져오는 데 걸리는 시간은 OSCAR보다 제안 방법이 현저하게 높게 나타난다. 제안 기법은 제안 방법은 되도록 많은 데이터를 디스크에 저장하는데 디스크 관리의 초점을 두었기 때문에 디스크에 데이터를 기록하고 나서 데이터를 압축한다. 따라서 디스크에 입력된 데이터를 검색하려면 다시 압축을 풀어야 하는 번거로움이 있다. 그러나 검색은 디스크뿐만 아니라 버퍼에서도 발생하기 때문에 버퍼에서 검색된 첫 결과 레코드를 얻는 데는 OSCAR만큼 적은 시간이 소요된다. 따라서 사용자는 기다리는 시간없이 계속적으로 결과를 볼 수 있다.

그림 16은 FAST가 동작하는 화면을 캡쳐한 모습이다. [Buffer Mgr]는 회부에서 입력된 데이터 셋을 순환버퍼에 저장하는 역할을 하고, [Write Thread]는 순환버퍼에 있는 완성된 레코드 셋을 디스크에 쓰는 역할을 한다. 순환버퍼가 비동기적으로 입출력을 진행하므로 [Buffer Mgr]와 [Write Thread]는 비동기적으로 데이터를 순환버퍼에 입력하고 또한 읽고 제거할 수 있다. 중간에 표시되는 <Select

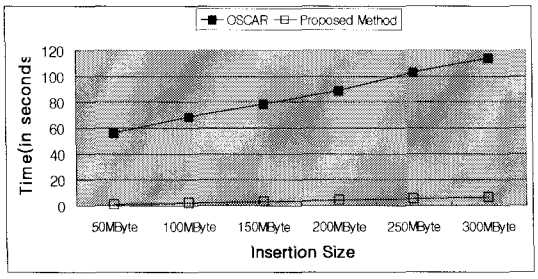


그림 14. OSCAR와 제안 방법의 데이터 삽입속도 비교

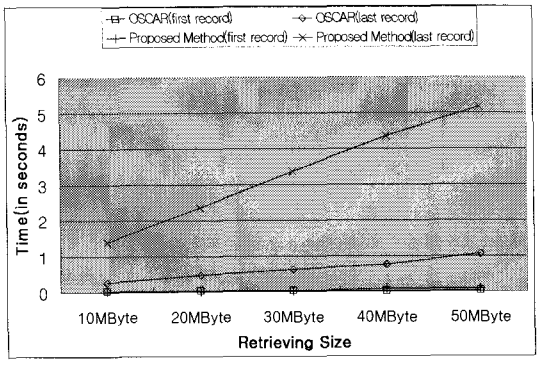


그림 15. OSCAR와 제안 방법의 검색 시간 비교

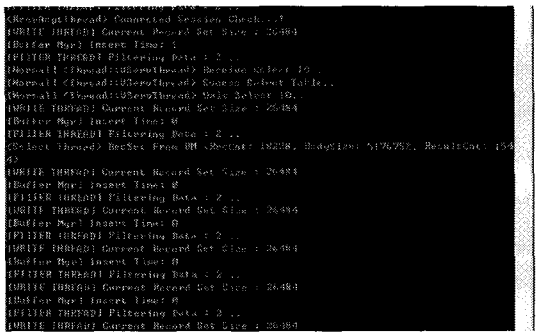


그림 16. FAST의 동작 모습

Thread>는 클라이언트의 Select 질의 시에 순환버퍼의 내용을 읽는 역할을 한다. II장에서 설명한 바와 같이 Select 연산은 Flush와 반-동시 수행, Insert와는 동시 수행이 가능하다. 따라서 <Select Thread>도 [Write Thread]와는 반-동시 수행이 가능하고 <Write Thread>와는 동시 수행이 가능하다.

[Buffer Mgr]에서 나타나는 Insert Time은 데이터가 입력되는 밀리초 단위의 시간을 나타낸다. 앞서 설명한 바와 같이 데이터는 셋 단위로 입력되어 단순히 메모리 복사연산을 하므로 1ms보다도 빠른 입력 시간을 보이므로 0이라고 표시된다. <Select Thread>가 연산을 할 때는 읽어가는 레코드 셋에 있는 레코드 의 수 RecCnt, 레코드 셋의 크기를 나타내는 BodySize, 그리고 연산 후 결과 레코드를 나타내는 ResultCnt를 콘솔창에 나타낸다. [Write Thread]가 디스크에 레코드 셋을 입력할 때는 정해진 크기가 넘는 완성된 레코드 셋만이 디스크에 쓴다. 위 그림에서 나타나는 완성된 레코드 셋의 기준은 20KByte로 작게하여 실험을 하였다. 따라서 20KByte의 레코드 셋이 입력되는 즉시 레코드 셋은 디스크에 쓰인다.

[Filter Thread]는 데이터 스트림의 연속질의를 실험하기 위해 구현을 했으나 연속질의에 대한 내용은 본 논문에서 다루지 않으므로 설명을 생략한다.

4. 결론 및 향후 연구

본 논문은 하이브리드 질의 처리를 위해 데이터 스트림을 빠르게 저장할 수 있는 기술인 FAST (FAst Storing Technique)를 제안한다. 데이터 스트림의 입력을 받는 버퍼는 가변길이 레코드의 저장과 입력과 플러시의 동시성을 위하여 순환버퍼로 구현된다. 데이터 스트림은 많은 입력을 가지기 때문에 디스크 용량의 한계를 고려하여 입력이 완성된 파일은 압축이 된다. 이러한 방법들은 레코드의 갱신이 없고 검색보다 삽입의 성능을 최대화하기 위함이다. FAST는 센서, RFID, 물류, 네트워 이벤트 분석 등 빠르고 많은 양의 데이터를 저장할 필요가 있는 모든 분야에 적용이 가능하다.

FAST는 빠르게 데이터 스트림을 저장 할 수 있게 한다. 성능 분석 결과 데이터를 저장하는데 가장 큰 장애는 디스크 I/O가 아니라 더 많은 데이터를 저장하기 위한 압축 알고리즘이다. 만약 입력 속도에 비하여 압축 알고리즘의 느리면, 디스크에 압축이 안된 파일이 많아져서 디스크가 금방 꽉 차게 된다. 따라서 데이터 스트림의 특성에 맞추어진 빠른 압축 알고리즘의 연구가 필요하다.

참 고 문 헌

[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] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring Streams - A New Class of Data Management Applications," *VLDB 2002*, pp. 215-226.

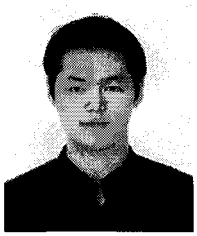
[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," *In Proceedings of CIDR*, 2003.

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

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

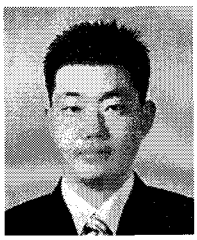
[6] S. Chandrasekaran and M. J. Franklin, "Remembrance of Streams Past: Overload-

Sensitive Management of Archived Streams," *VLDB*, pp. 348-359, 2004.
[7] B. Babcock, M. Datar, and R. Motwani. "Sampling from a Moving Window over Streaming Data," *SODA*, pp. 633-634, 2002.
[8] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. "Sampling algorithms in a stream operator," *SIGMOD Conference*, 2005.
[9] 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*, Vol.32, No.3, pp. 197-211, 2006.
[10] Kopetz, H. and Reisinger, J., "NBW: A Non-Blocking Write Protocol for Task Communication in Real-Time Systems," *Proc. IEEE CS 1993 Real-Time Systems Symp.*, pp. 131-137, Dec. 1993.
[11] P. Muth, P. O'Neil, A. Pick, and G. Weikum, "The LHAM Log-Structured History Data Access Method," *VLDB 2000*. Vol.9, No.3-4, pp. 199-221, 2000.
[12] M. OverMars, "The design of dynamic data structures," *LNCS*, 1983.
[13] U. Srivastava, and J. Widom, "Flexible time management in data stream systems," *PODS*, pp. 263-274, 2004.



신 재 진

2005년 인하대학교 컴퓨터공학과(공학사)
2007년 인하대학교 컴퓨터정보공학과(공학석사)
2007년~현재 (주)한국공간정보통신
관심분야 : 데이터베이스 관리 시스템, 데이터 스트림, 데이터 마이닝, 공간 데이터베이스, 베이스 기타

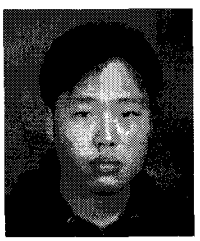


유 병 섭

2002년 인하대학교 전자·전기·컴퓨터공학부-컴퓨터공학(공학사)
2004년 인하대학교 대학원 전자계산공학과(공학석사)
2004년~현재 인하대학교 대학원 정보공학과(박사과정)
관심분야 : 공간 데이터베이스, 공간 데이터 웨어하우스, 센서 네트워크

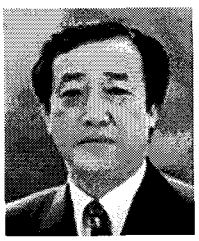
어 상 훈

2003년 인하대학교 컴퓨터공학부(공학사)
2003년~현재 인하대학교 대학원 컴퓨터정보공학과(통합과정)
관심분야 : 다중레벨 데이터베이스, LBS, 유비쿼터스 컴퓨팅, RFID 미들웨어



이 동 욱

2003년 상지대학교 전자계산공학과(이학사)
2005년 인하대학교 컴퓨터 정보공학과(공학석사)
2005년~현재 인하대학교 정보공학과(박사과정)
관심분야 : Spatial Data Warehouse, Ubiquitous 환경을 위한 SDBMS



배 해 영

1974년 인하대학교 공학사(응용물리학)
1978년 연세대학교 공학석사(전자계산학)
1990년 숭실대학교 공학박사(전자계산학)
1992년~1994년 인하대학교 전자계산소 소장
1982년~현재 인하대학교 컴퓨터공학부 교수
1999년~현재 지능형GIS연구센터 센터장
2000년~현재 중국 중경우전대학교 대학원 명예교수
2006년~현재 인하대 대학원원장
관심분야 : 분산 데이터베이스, 공간 데이터베이스, 지리정보 시스템, 멀티미디어 데이터베이스 등