http://dx.doi.org/10.3745/KIPSTD.2012.19D.2.161

임의쓰기 성능향상을 위한 로그블록 기반 FTL의 효율적인 합병연산

이 준 혁[†] · 노 홍 찬[††] · 박 상 현[†††]

요 약

최근 플래시 메모리의 꾸준한 용량 증가와 가격 하락으로 인해 대용량 SSD(Solid State Drive)가 점차 대중화 되고 있다. 하지만, 플래시 메모리는 하드웨어적인 제약사항이 존재하며, 이러한 제약사항을 보완하기 위해 FTL(Flash Translation Layer)이라는 특별한 미들웨어 계층을 필요로 한다. FTL은 플래시 메모리의 하드웨어적인 제약사항을 효율적으로 운용하기 위해 필요한 계층으로서 파일 시스템으로부터의 논리적 섹터 번호(logical sector number)를 플래시 메모리의 물리적 섹터 번호(physical sector number)로 변환해주는 역할을 한다. 특히, 플래시 메모리의 여러 제약사항 중 "쓰기 전 지우기(erase-before-write)"는 플래시 메모리 성능 저하의 주요한 원인이 되고 있으며, 이와 관련하여 로그블록 기반의 여러 연구들이 활발히 진행되어 왔지만, 대용량의 플래시 메모리를 효율적으로 운용하기 위해서는 몇몇 문제점들이 존재한다.

로그블록 기반의 FAST는 넓은 지역에 임의쓰기(random writing)가 빈번하게 발생하면 데이터 블록 내 사용되지 않은 섹터들로 인해 효율적이지 못한 합병 연산이 발생한다. 즉, 효율적이지 못한 블록 쓰레싱(thrashing)이 빈번하게 발생하고, 플래시 메모리의 성능을 저하시킨다.

로그블록은 덮어쓰기(overwriting) 발생 시 일종의 캐쉬처럼 운영되며, 이러한 기법은 플래시 메모리 성능 향상에 많은 발전을 주었다. 본 연구에서는 임의쓰기에 대한 성능 향상을 위해 로그 블록만을 캐쉬처럼 운영하는 것이 아니라 플래시 메모리 전체를 캐쉬처럼 운용하고, 이를 위해 별도의 오프셋이라는 매핑 테이블을 운용하여 플래시 메모리 성능 저하의 주요한 원인이 되는 합병연산과 삭제연산을 줄였다.

새로운 FTL은 XAST(eXtensively-Associative Sector Translation)이라 명명하며, XAST에서는 공간지역성과 시간지역성에 대한 기본적인 이론을 바탕으로 오프셋 매핑 테이블을 효율적으로 운용한다.

키워드 : 플래시 메모리, FTL, 합병 연산, 로그 블록, XAST, SSD

The Efficient Merge Operation in Log Buffer-Based Flash Translation Layer for Enhanced Random Writing

Junhyuk Lee[†] · Hongchan Roh[††] · Sanghyun Park[†††]

ABSTRACT

Recently, the flash memory consistently increases the storage capacity while the price of the memory is being cheap. This makes the mass storage SSD(Solid State Drive) popular. The flash memory, however, has a lot of defects. In order that these defects should be complimented, it is needed to use the FTL(Flash Translation Layer) as a special layer. To operate restrictions of the hardware efficiently, the FTL that is essential to work plays a role of transferring from the logical sector number of file systems to the physical sector number of the flash memory.

Especially, the poor performance is attributed to Erase-Before-Write among the flash memory's restrictions, and even if there are lots of studies based on the log block, a few problems still exists in order for the mass storage flash memory to be operated.

If the FAST based on Log Block-Based Flash often is generated in the wide locality causing the random writing, the merge operation will be occur as the sectors is not used in the data block.

In other words, the block thrashing which is not effective occurs and then, the flash memory's performance get worse.

If the log-block makes the overwriting caused, the log-block is executed like a cache and this technique contributes to developing the flash memory performance improvement. This study for the improvement of the random writing demonstrates that the log block is operated like not only the cache but also the entire flash memory so that the merge operation and the erase operation are diminished as there are a distinct mapping table called as the offset mapping table for the operation.

The new FTL is to be defined as the XAST(extensively-Associative Sector Translation). The XAST manages the offset mapping table with efficiency based on the spatial locality and temporal locality.

Keywords : Flash Memory, FTL, Merge Operation, Log Block, XAST, SSD

※ 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2011-0004382).
† 준 회 원 : 연세대학교 컴퓨터공학과 석사과정
†† 준 회 원 : 연세대학교 컴퓨터과학과 박사과정
††† 종신회원 : 연세대학교 컴퓨터과학과 교수(교신저자)
논문접수 : 2011년 12월 26일
수 정 일 : 1차 2012년 3월 6일
심사완료 : 2012년 3월 7일

1. 서 론

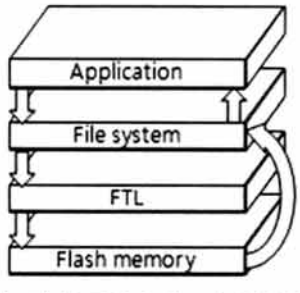
플래시 메모리는 저전력, 비휘발성, 저소음, 빠른 접근속도, 가벼운 무게, 강한 내구성 등으로 인해 휴대폰, MP3, PDA 등 여러 장치들에서 저장매체로 사용되고 있으며, 플래시 메모리의 가격이 하락하면서 이를 사용한 대용량 SSD(Solid State Drive)가 점차 대중화되어 노트북, PC, 스토리지 등에 널리 쓰여 지고 있다. 하지만, 플래시 메모리는 기존 하드디스크와 비교하여 하드웨어적인 측면에서 여러 제약사항 존재하고 있다.

플래시 메모리는 기존 하드디스크와는 달리 블록(Block)들로 구성이 되어있으며, 각각의 블록들은 섹터(Sector)들로 구성이 되어있다. 섹터는 읽기(Read), 쓰기(Write)의 단위로 동작하고, 블록은 삭제(Erase)의 단위로 동작한다. 즉, 읽기, 쓰기와 삭제의 대역폭이 다르며, 삭제가 읽기, 쓰기와 비교하여 상대적으로 연산속도가 느리다[13]. 또한, 전통적인 하드디스크와는 달리 해당 섹터에 접근 시 기계적인 지연이 없기 때문에 임의 접근(random access)이 매우 빠르다[2]. <표 1>은 삼성 K9F1G08U0B NAND 플래시의 Access time을 나타내며, 페이지(page)가 섹터를 말한다.

〈표 1〉 NAND 플래시 Access time (K9F1G08U0B)

Media	Page Size	Access time		
		Read (1 pages)	Write (1 page)	Erase (1 block)
NAND Flash	(2K + 64)Byte	25 ns ~ 25 us	200 us	1.5 ms

플래시 메모리의 "쓰기 전 지우기(erase-before-write)" 제약사항으로 인해 덮어쓰기(Overwrite)연산이 요청되었을 경우 기존 섹터에 데이터가 존재하면 새로운 블록을 찾아 새로운 섹터에 쓰기연산을 수행하고, 기존 블록을 삭제하여야 한다[2, 13]. 이때, 기존 블록에 유효한 데이터가 존재하면 이들까지도 새로운 블록으로 옮기는 작업을 수행해야 하며, 이러한 일련의 작업을 합병(Merge)연산이라고 한다. 위처럼 플래시 메모리의 덮어쓰기에 대한 제약 때문에 비효율적인 연산을 효율적으로 처리하기 위해 (그림 1)과 같이 FTL(Flash Translation Layer)이라는 특별한 소프트웨어 계층이 존재한다.



(그림 1) 플래시 메모리 아키텍쳐

FTL은 파일 시스템과 플래시 메모리 사이에 위치하며, 파일 시스템으로부터 처리하는 논리 주소를 물리 주소로 변환시켜주는 역할을 수행한다[13]. 특히, 덮어쓰기 연산 시 발생하는 가장 느린 삭제연산의 횟수를 줄이기 위한 여러 연구들이 진행되어져 왔다. 또한, 플래시 메모리의 블록들은 삭제 횟수에 제한이 있는데, 일정 횟수가 넘어가면 더 이상 해당 블록을 사용할 수 없기 때문에 이러한 블록들의 삭제에 대한 삭제평준화(wear-leveling)도 운영하며, 갑작스럽게 끊어지는 전원으로부터의 데이터 안정성도 보장한다[13]. 결국, 효과적인 FTL 알고리즘을 개발하는 것이 플래시 메모리를 이용하는 시스템에서 매우 중요하다.

FTL은 여러 주소변환 기법이 있으며, 기본적으로 섹터 매핑 기법, 블록 매핑 기법, 하이브리드 매핑 기법이 있다. 이러한 FTL 기법은 매핑 테이블을 운용하기 위해서는 RAM을 필요로 하는데 각각의 기법들에 따라 요구하는 RAM의 크기가 서로 다르며, SRAM의 경우 요구되어지는 양이 많아질수록 제품의 가격이 증가하는 단점이 있다. 여러 NAND플래시 칩으로 구성된 CompactFlash 시스템에서 FTL코드는 컨트롤러에 의해 ROM에 저장되어 수행되며, SRAM에 매핑 테이블을 저장한다[2].

섹터 매핑 기법은 논리적 섹터 번호(LSN : Logical Sector Number)와 물리적 섹터 번호(PSN : Physical Sector Number)로 매핑 되어 있고, LSN과 PSN이 1:1로 매핑 되어 있기 때문에 속도 측면에서는 빠르나, 너무 많은 양의 SRAM을 필요로 한다는 단점이 있다.

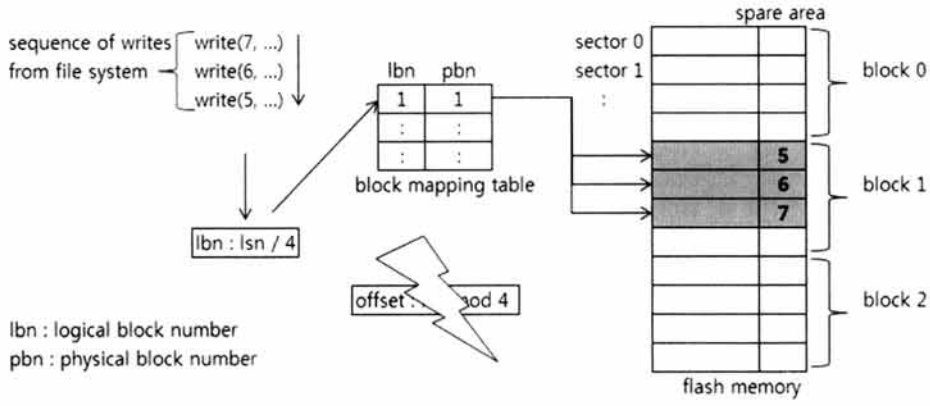
블록 매핑 기법에서는 논리적 블록 번호(LBN : Logical Block Number)와 물리적 블록 번호(PBN : Physical Block Number)로 매핑 되어 있고, 각각의 오프셋으로 섹터들에 접근한다. 블록 매핑 기법은 섹터 매핑 기법에 비해 상대적으로 적은 양의 SRAM을 필요로 하지만 덮어쓰기 발생 시 빈번한 블록 쓰레싱(Thrashing) 현상이 발생하는 단점이 있다.

하이브리드 매핑 기법은 섹터 매핑 기법의 장점과 블록 매핑 기법의 장점을 서로 융합한 기법으로 블록 매핑 테이블을 기반으로 섹터들에 대한 접근을 오프셋이 아닌 Spare Area라는 별도의 공간을 사용하여 접근한다. 하지만, 블록 내 해당 LSN에 접근하기 위해 첫 번째 섹터부터 순차적으로 검색해야 하는 단점이 있으며, SRAM이 아닌 플래시 메모리 내부에서 검색하기 때문에 속도가 느리다.

이처럼 기본적인 주소변환 기법들이 있으며, 플래시 메모리의 비효율적인 덮어쓰기 요청 시 발생되는 합병(Merge)연산의 삭제횟수 최소화를 목적으로 여러 기법들이 제안되어 왔다. 특히, 일종의 캐쉬처럼 동작하는 로그 블록 기반의 기법들이 꾸준히 연구되어 왔으며, 이 중 BAST, FAST의 동작 과정을 살펴보고, 본 논문에서는 이러한 합병 연산의 효율적인 운영을 위한 기법을 제시한다.

2. 이론적 배경

하이브리드 매핑 기법은 섹터 매핑 기법의 장점과 블록



(그림 2) 하이브리드 매핑 기법

매핑 기법의 장점을 서로 융합한 기법이다. 블록 매핑 기법에서의 오프셋을 제거하고 플래시 메모리의 spare area를 활용하여 섹터에 접근하는 방식이다[13]. 일단 spare area에 대해 먼저 설명하자면 하나의 섹터는 데이터를 저장하는 data area영역과 여분의 정보를 저장하기 위한 spare area로 구성되어 있다[2]. 이 spare area에는 여러 관련 정보들이 저장되는데 논리 섹터 번호를 함께 저장함으로써 오프셋과 관계없이 원하는 섹터에 접근할 수 있다.

(그림 2)를 보면 논리 섹터 번호 5번, 6번, 7번이 오프셋과 관계없이 해당 블록에 순차적으로 쓰기 연산이 수행된 것을 볼 수 있다. 만약 더 이상 블록 내 비어있는 섹터가 존재하지 않을 경우 합병 연산이 발생한다.

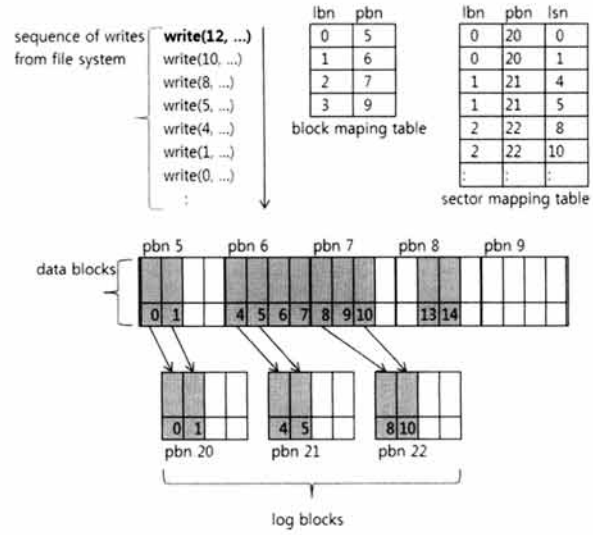
하이브리드 매핑 기법은 기존 블록 매핑 기법에 비해 공간의 효율성을 높이고 빈번한 블록 쓰레싱 현상을 개선하였지만 해당 논리 섹터 번호에 접근하기 위해서는 해당 블록의 첫 번째 섹터부터 검색을 수행해야 하고, 접근하고자 하는 논리 섹터 번호가 블록 내 뒤에 위치하였을 경우 느리다. 또한, 이러한 검색이 SRAM이 아닌 플래시 메모리 내부에서 수행되기 때문에 기존 섹터 매핑 기법과 블록 매핑 기법에 비해 상대적으로 느리다는 단점이 있다[13].

3. 플래시 메모리 관련 연구

위에서 기본적인 주소 매핑 기법에 대해 살펴보았다. 섹터 매핑 기법은 논리 섹터 번호와 물리 섹터 번호가 1:1로 매핑되어 있기 때문에 접근이 빠르고, 덮어쓰기 발생 시 유연하게 동작하지만, 많은 양의 SRAM을 필요로 하고, 블록 매핑 기법은 섹터 매핑 기법에 비해 작은 양의 SRAM을 요구하지만, 덮어쓰기 발생 시 빈번한 블록 쓰레싱이 발생한다. 하이브리드 매핑 기법은 spare area를 활용하여 오프셋과 상관없이 운용이 가능하지만, 접근하고자 하는 논리 섹터 번호를 해당 블록 내에서 순차 검색을 통해 접근하기 때문에 원하는 데이터가 블록 내 뒤에 있을 경우 느리다는 단점이 있다. 또한, 이러한 검색이 SRAM이 아닌 플래시 메모리 내부에서 수행되기 때문에 상대적으로 느리다.

이러한 단점들을 해결하고, 플래시 메모리 성능 저하의 주요한 원인이 되는 합병연산을 효율적으로 운용하기 위해 로그 블록 기반의 여러 기법들이 제안되어 왔다. 로그블록은 숨겨진 영역으로서 데이터 저장을 위해 쓰이며, 섹터 매핑 수준의 테이블로 관리 및 운영이 된다. 로그블록들의 집합을 로그버퍼라 하며[9], 그 중 BAST와 FAST에 대해 살펴본다.

3.1 BAST 기법

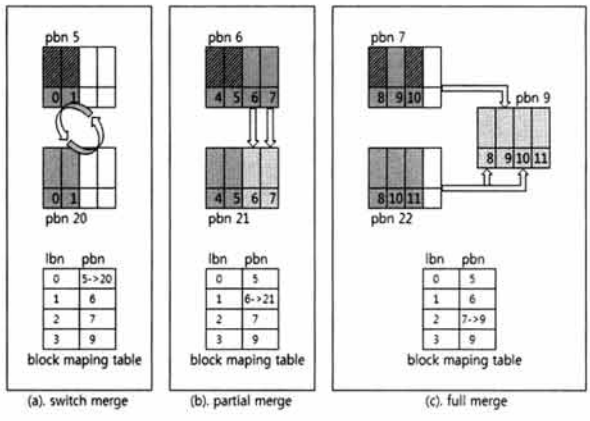


(그림 3) BAST 기법

BAST는 기본적으로 블록 매핑 기법을 사용하며, 일종의 캐쉬 역할처럼 동작하는 로그블록을 사용하여 덮어쓰기 발생 시 로그블록에 쓰기 연산을 수행함으로써 플래시 메모리 성능 저하의 주요한 역할을 하는 합병 연산의 횟수를 줄였다. 또한, 각각의 로그블록들은 별도의 섹터 매핑 테이블에서 논리 섹터 번호를 관리 및 운영을 하여 해당 논리 번호 섹터에 접근이 가능하도록 한다. 접근 방향은 마지막 섹터로부터 거꾸로 검색하여 최신 데이터를 읽을 수 있다[2].

BAST는 각각의 데이터 블록과 로그 블록이 1:1로 연관

되어 있으며, (그림 3)은 BAST의 동작을 보여준다. (그림 3)에서는 데이터블록에 이미 데이터들이 쓰여 있다고 가정하고, 각각의 데이터블록에 그림과 같이 순차적으로 덮어쓰기가 발생하였을 경우 로그블록에 해당 데이터가 어떻게 쓰여 지는지 알 수 있다. 즉, 덮어쓰기 발생 시 1:1로 연관된 로그블록에 순차적으로 갱신되는 데이터를 쓰고, 로그블록 섹터 매핑 테이블의 정보를 갱신하게 된다. 만약 로그블록 내 더 이상 비어있는 섹터가 없으면 합병 연산을 수행한다. 또한, BAST는 덮어 쓰기 발생 시 이미 모든 로그 블록이 사용되고 있다면, 가장 먼저 사용된 로그블록이 선택되어 연관된 데이터 블록과 합병 연산을 수행하는데 그림에서 write(12, ...)가 요청되면 모든 로그 블록이 데이터 블록과 1:1로 연관되어 사용되고 있기 때문에 합병 연산이 수행된다. 이러한 합병연산은 로그 블록 기반의 FTL에서 (그림 4)에서처럼 3가지로 나뉜다[2].



(그림 4) 합병 연산의 종류

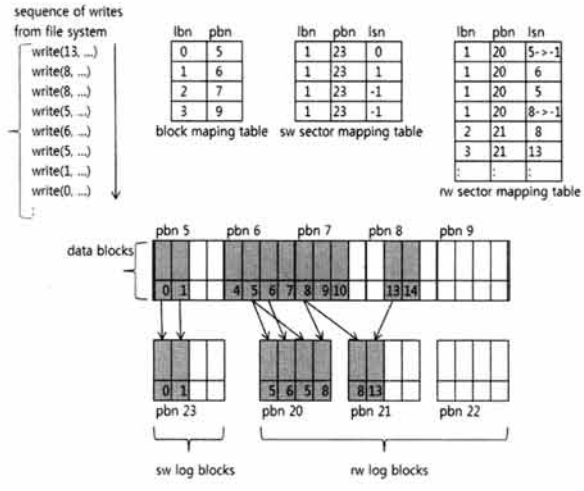
(a) switch merge : 스위치 합병은 데이터 블록과 로그 블록의 논리 섹터 번호가 순차적으로 모두 1:1로 동일한 경우 매핑 테이블의 블록 정보만 변경하고, 기존 데이터블록은 삭제함으로써 다른 합병 연산에 비해 상대적으로 가장 빠르게 합병 연산을 수행한다. 즉, 최신 데이터가 모두 로그 블록에 위치 한 경우다.

(b) partial merge : 부분 합병은 데이터 블록의 논리 섹터 번호가 로그 블록에 일정 부분 순차적으로 1:1로 동일한 경우 데이터 블록 내 유효한 데이터만 로그블록으로 복사하고, 매핑 테이블의 블록 정보를 변경한다. 그리고 기존 데이터블록은 삭제함으로써 일련의 합병 연산을 수행한다. 복사해오는 추가 연산이 발생함으로 스위치 합병 보다 느리다.

(c) full merge : 전체 합병은 데이터블록과 로그블록 내 유효한 데이터를 비어있는 블록에 복사하고, 기존 데이터블록과 로그블록을 삭제하는 연산이다. 복사와 두 번의 삭제 연산이 수행되기 때문에 가장 성능이 나쁘고, 이러한 전체 합병 연산을 줄이기 위해 여러 기법들이 제안되어왔다.

BAST는 로그블록을 활용하여 연속적인 순차 쓰기 요청 발생 시 스위치 합병 연산을 통해 플래시 메모리 성능을 향상 시켰다. 하지만 데이터 블록과 로그블록과의 관계가 1:1로 제한되어있기 때문에 로그 블록의 효율이 좋지 않다. 즉, 모든 로그블록이 사용 중이고, 연관되지 않은 데이터 블록에 덮어쓰기가 발생하면 가장 먼저 쓰인 로그 블록을 선택해서 합병 연산이 수행되어지는데, 이때 로그 블록 내 비어있는 섹터가 있음에도 불구하고, 합병 연산이 수행되기 때문에 로그 블록의 효율이 좋지 않다. 또한, 플래시 메모리 내 임의 쓰기가 여기저기 발생 할 경우 데이터 블록과 로그 블록과의 1:1 관계로 인하여 합병 연산이 빈번하게 발생한다. (그림 3)에서 물리 블록 번호 5번, 6번, 7번, 8번에 반복적인 덮어쓰기가 발생하면 로그 블록은 3개밖에 없기 때문에 4개의 데이터 블록과 1:1 연관 관계를 위해서는 빈번한 합병 연산이 계속 발생한다. 이러한 현상을 블록 쓰레싱이라 한다[6].

3.2 FAST 기법



(그림 5) FAST 기법

FAST는 로그블록과 데이터 블록의 관계를 1:N으로 확장하여 로그블록의 효율성을 높이고, 합병 연산의 횟수를 대폭 줄여 플래시 메모리 성능 저하의 삭제 연산을 획기적으로 줄인 기법이다[6].

FAST는 순차쓰기(sequential writing)와 임의쓰기(random writing)용 로그블록이 구분되어 운용되기 때문에 순차쓰기 패턴의 쓰기 연산을 스위치 합병이나 부분 합병으로 효율적으로 운용하고, 임의쓰기 패턴은 전체 합병은 통해 운용된다.

(그림 5)를 보면 물리 블록 번호 5번은 순차 쓰기 로그 블록과 연관이 되어있음을 볼 수 있다. 순차 쓰기 로그 블록의 선택 기준은 덮어쓰기가 발생 하였을 때 논리 섹터 번호의 오프셋이 0일 경우 선택되어 쓰여 진다. 이후 해당 데이터 블록에 덮어쓰기가 발생하였을 경우 해당 논리 섹터 번호의 오프셋이 순차쓰기 로그 블록 내 현재 쓰여 있는 섹터 바로 뒤의 오프셋과 동일할 경우 쓰기 연산을 수행한다. 만약, 오프셋이 서로 다를 경우 순차 쓰기 로그 블록은 합병 연산이 수행된다. 또한, 현재 순차 쓰기 로그 블록이 사

용 중이고, 다른 데이터 블록의 오프셋 0번에 대한 덮어쓰기가 발생하여도 합병 연산이 수행된다. 이러한 순차 쓰기용 로그블록은 기본적으로 스위치 합병이나 부분합병이 발생하기 때문에 효율이 좋다.

임의 쓰기용 로그 블록은 BAST와 달리 로그 블록과 데이터 블록과의 관계를 1:N으로 확장함에 따라 좀 더 복잡하게 운용되며, 1개의 로그 블록이 여러 데이터 블록과 관계가 맺어지기 때문에 합병 연산 시 관련 된 모든 데이터 블록과 합병 연산이 수행된다. 즉, 합병 연산 중 가장 성능이 나쁜 전체 합병 연산이 수행된다[12].

(그림 5)를 보면 FAST는 기본적으로 블록 매핑 기법을 통해 해당 블록 번호에 접근한다. 그리고 순차 쓰기용 로그블록과 임의 쓰기용 로그블록을 위한 섹터 매핑 수준의 테이블을 별도로 운용한다. 데이터에 접근할 때는 이러한 매핑 테이블을 참조하여 원하는 데이터에 접근할 수 있다. 만약 모든 로그 블록이 다 사용되고 물리 번호 20번에 대한 합병 연산이 수행되면 해당 테이블을 참조하여 관련된 데이터 블록과 합병 연산이 수행된다. 이때, 임의 쓰기 용 매핑 테이블은 최신 데이터를 기준으로 같은 논리 섹터 번호에 대한 정보를 -1의 값으로 변경하여 유효하지 않은 논리 섹터 번호를 구분하여 합병 연산을 수행한다.

FAST의 핵심 아이디어는 일종의 캐쉬처럼 운용 되는 로그 블록을 캐쉬의 사상 방식 측면에서 완전 연과 사상(fully associative mapping)으로 운영하여 플래시 메모리의 성능을 대폭 개선하였다[6]. 즉, BAST에서의 빈번한 임의 쓰기 요청 시 계속적으로 발생하는 합병 연산과 삭제 연산의 횟수를 줄였다. 하지만, FAST는 순차 쓰기 패턴과 랜덤 쓰기 패턴이 동시에 섞여서 요청될 경우 패턴을 효율적으로 구분 및 처리하지 못하고, 임의 쓰기에 대한 지역성이 고려되지 않았다[9].

4. 제안기법

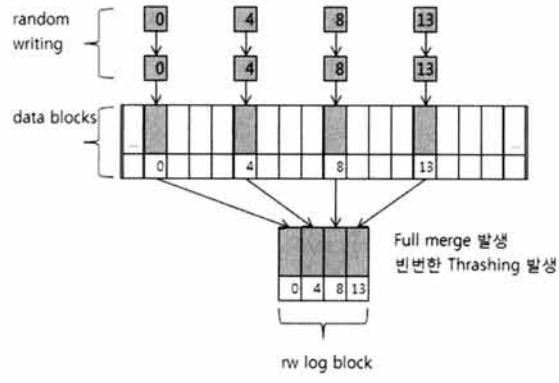
4.1 In-place scheme와 Out-place scheme

위에서 여러 매핑 기법들에 대해 살펴보면서 데이터 블록 내 어떤 방식으로 쓰기 연산이 수행되는지 보았다. 쓰기 연산의 방식은 두 가지로 구분되는데 해당 논리 섹터 번호의 오프셋과 동일한 위치에 쓰기 연산을 수행하는 기법을 In-place scheme이라 하고, 오프셋과 상관없는 위치에 쓰기 연산을 수행하는 기법을 out-place scheme이라 한다. FAST에서의 동작 방식을 보면 데이터 블록은 In-place scheme방식으로 수행되고, 로그 블록은 out-place scheme로 수행된다.

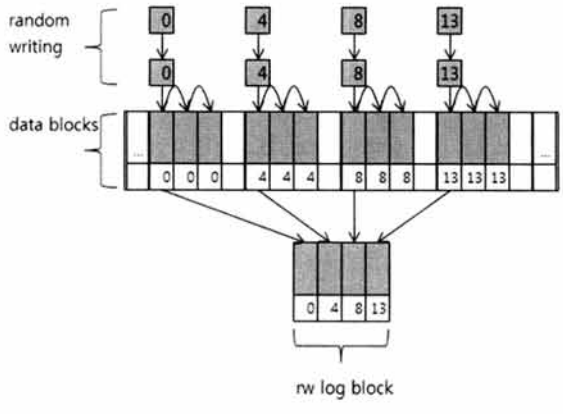
out-place scheme은 별도의 섹터 매핑 테이블을 운용하거나[5], spare area에 해당 논리 섹터 번호를 기입함으로써 데이터 접근을 가능하게 한다. out-place scheme은 별도의 매핑 테이블을 운용해야 하는데 In-place scheme에 비해 공간 효율이 좋기 때문에 합병 연산과 삭제 연산의 횟수를 줄일 수 있는 장점이 있다.

본 논문에서 제안하는 FTL에서는 데이터 블록 내 쓰기 연산을 In-place scheme과 Out-place scheme를 함께 사용한다.

4.2 핵심 아이디어



(그림 6) FAST에서 비효율적인 합병 연산



(그림 7) XAST의 데이터 블록 공간 효율성

FAST는 로그블록과 데이터블록과의 관계를 1:N으로 운용하여 로그 블록의 효율성을 최대화하고, 합병연산을 최소화하여 BAST의 빈번한 블록 쓰레싱 현상을 대폭 개선하였다. 또한, 순차쓰기용 로그블록을 별도로 운용하여 순차쓰기 패턴에 대해서도 고려하였다. 그러나 (그림 6)에서 임의 쓰기용 로그블록(rw log block)의 합병 연산 시 데이터블록 내 비어있는 섹터가 존재할 수 있으며, 본 논문에서는 이러한 공간까지 모두 활용하여 FTL의 성능을 극대화 하고자 한다. 즉, 로그 블록만 캐쉬처럼 사용하는 것이 아니라 (그림 7)처럼 플래시 메모리 전체를 캐쉬처럼 사용하고자 한다.

본 논문에서 제안하는 새로운 FTL은 XAST(eXtensively Associative Sector Translation)라 명명한다.

4.3 XAST 전체 구조

XAST는 블록 매핑 기법을 기반으로 동작하며, 로그블록들은 FAST와 동일하게 섹터 매핑 테이블을 통해 운용된다. 또한, FAST처럼 순차 쓰기용 로그 블록과 임의 쓰기용 로그블록을 활용하고, 순차 쓰기용 로그 블록의 동작에 대해

서는 더 이상 개선의 여지가 없어 보이기 때문에 임의 쓰기용 로그블록에 대해서만 다루도록 하고, 이후 로그블록은 모두 임의 쓰기용 로그 블록을 말한다.

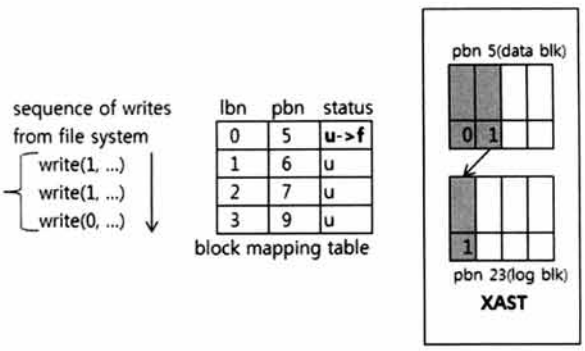
XAST에서는 블록 매핑 테이블에 현재 블록에 대한 상태값을 추가하고, 플래시 메모리 내부를 캐쉬처럼 사용하기 위해 별도의 오프셋 섹터 매핑 테이블을 운용한다. 블록들의 상태값은 f(fluent)와 u(unfluent)로 구분하고, 블록이 f이면 데이터블록 내 비어있는 곳에 out-place schema 방식으로 쓰기연산이 수행되며, 상태값이 u이면 in-place schema 방식으로 동작한다. 블록의 상태값은 덮어쓰기 발생 시 로그블록에 쓰기 연산을 수행하고, 해당 데이터 블록에 비어있는 섹터가 존재하면 f값으로 갱신되며, 해당 블록이 합병 연산 수행 후에는 다시 u값으로 초기화 된다. 상태값이 u인 경우에는 FAST와 동일하게 동작하고, f값으로 변경된 데이터 블록들은 오프셋 섹터 매핑 테이블의 주어진 공간만큼 out-place schema 방식으로 쓰여 졌던 섹터들에 대한 정보가 관리되어지며, 데이터 접근 시 해당 테이블을 참조하여 하이브리드 기법처럼 spare area을 검색하여 접근한다. 상태값이 f인 블록들은 모두 로그 블록들과 연관되어있으며, 합병 연산 발생 시 별도의 오프셋 섹터 매핑 테이블을 참조하여 수행한다. 합병 연산 시에는 오프셋 섹터 매핑 테이블에 관리되어진 데이터를 삭제하여 다시 활용될 수 있도록 한다. XAST에서는 오프셋 매핑 테이블을 다시 재사용하는 것이 매우 중요하며, 합병 연산을 통해 사용 공간을 다시 확보하는 흐름은 (그림 8)과 같다. 일반적인 어플리케이션의 빈번한 갱신(update)들은 높은 공간 지역성을 갖고 있기 때문에[5, 9] 오프셋 매핑 테이블은 효율적으로 운용이 된다.

(그림 8) 오프셋 테이블 흐름

이처럼 데이터 블록의 비어있는 섹터들을 활용하고, 로그블록의 합병 연산을 일종의 이벤트로 인식하여 오프셋 섹터 매핑 테이블에 관리되어지는 데이터를 효율적으로 운용함으로써 플래시 메모리의 전체적인 성능을 개선한다.

4.4 XAST 매핑 테이블 운용

XAST에서는 임의 쓰기에 대한 효율을 높이고자 데이터 블록 내 비어있는 섹터를 이용한다. 이를 위해 블록 매핑 테이블에 각각의 블록들에 대한 상태값을 추가하고, 상태값에 따라 비어있는 섹터를 활용한다.

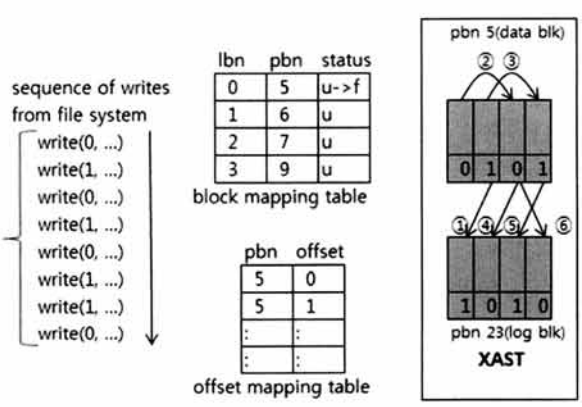


(그림 9) XAST 블록 매핑 테이블

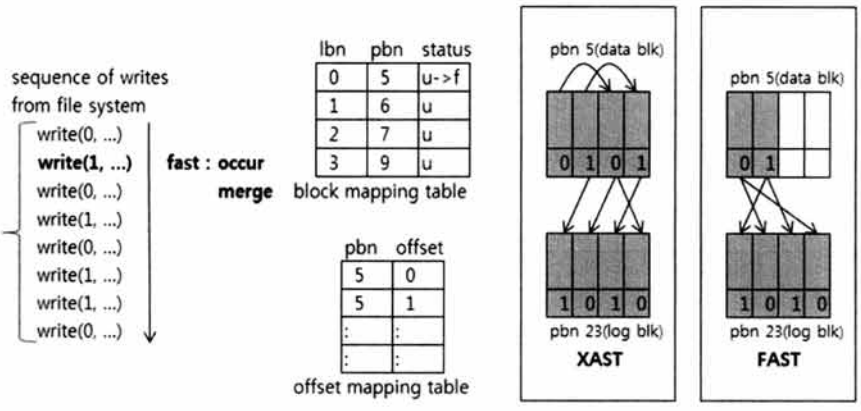
데이터블록 내 비어있는 섹터를 활용하기 위해서는 블록의 상태값이 f값이어야 하는데 값이 f가 되기 위해선 해당 데이터 블록에 덮어쓰기가 한번이라도 발생해야 하고, 동시에 데이터블록에 비어있는 섹터가 있어야 상태값이 f로 변경된다. (그림 9)을 보면 3번째 write(1, ...)에서 해당 데이터 블록의 상태값이 f로 변경된 것을 볼 수 있다. 데이터 블록의 상태값이 f로 변경되면 블록 내 비어있는 섹터에 out-place schema 방식으로 데이터를 쓰기 위해 별도의 섹터 매핑 수준의 오프셋 매핑 테이블을 사용한다. 더 이상 비어있는 섹터가 존재하지 않으면 FAST와 동일하게 로그블록에 쓰기 연산을 수행한다.

이처럼 상태값이 f인 데이터블록들은 모두 로그블록과 연관되어 있으며 이러한 연관관계는 XAST에서 운용되는 오프셋 매핑 테이블의 관리에 매우 중요한 역할을 한다. 즉, 합병 연산을 수행하면 더 이상 해당 데이터 블록에 대한 정보는 오프셋 매핑 테이블에서 필요가 없어지기 때문에 삭제되고 다시 삭제된 공간만큼 사용할 수 있는 것이다. 만약, 오프셋 매핑 테이블의 용량을 모두 사용했을 경우에는 더 이상 사용할 수 없고, 합병 연산이 수행될 때까지 대기해야 하지만 제한된 로그블록의 개수와 빈번한 덮어쓰기 발생은 오프셋 매핑 테이블을 빠르게 재사용 할 수 있게 해준다. (그림 10)은 오프셋 매핑 테이블을 운용을 보여준다.

(그림 10)의 오프셋 매핑 테이블을 보면 물리블록 5번의 0번과 1번이 저장되어 있다. 즉, 데이터블록 내 0번째와 1번



(그림 10) XAST의 오프셋 매핑 테이블 운용

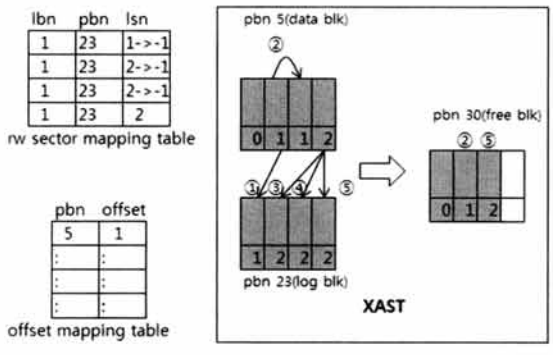


(그림 11) 데이터 블록 기준의 XAST와 FAST의 동작 비교

째 오프셋은 이미 유효하지 않은 데이터를 말한다. 이러한 매핑 정보가 어떻게 만들어지는 해당 그림을 통해 알 수 있다. 우선, 3번째 write(1, ...) 요청이 수행되면 로그 블록에 논리 섹터 번호 1번의 데이터가 로그 블록 첫 번째 섹터에 쓰기 연산을 수행하고, 데이터 블록에는 비어있는 섹터가 존재하기 때문에 블록 매핑 테이블의 상태값이 f로 갱신된다. 그리고 4번째 write(0, ...)가 발생하면 해당 데이터 블록의 상태값이 f이기 때문에 데이터 블록 내 비어있는 3번째 섹터에 쓰기 연산을 수행하고, 오프셋 매핑 테이블의 정보를 갱신한다. 5번째 write(1, ...)에 대해서도 동일하게 수행된다. 6번째 write부터는 데이터 블록 내 더 이상 비어있는 섹터가 없기 때문에 로그블록에 쓰기 연산을 수행한다. 덮어쓰기 연산에 대한 순서는 그림 오른쪽의 XAST 블록 그림을 보면 알 수 있다. 이후 그림에서 해당 로그블록에 합병 연산이 발생하면 오프셋 매핑 테이블에서 운영되는 물리 블록 번호 5번에 대한 데이터가 삭제되고 비어있는 공간만큼 또다시 사용될 수 있다. 이러한 오프셋 매핑 테이블의 재활용에 대한 가능성은 작은 크기의 임의쓰기들은 높은 공간 지역성을 갖기 때문이다[5, 9].

XAST는 해당 데이터에 접근하기 위해서는 우선 해당 블록의 상태값을 판별하고 상태값이 u라면 기존 블록 매핑 기법처럼 접근하고, 상태값이 f라면 로그블록 매핑 테이블을 검색해서 해당 논리 섹터 번호를 찾는다. 만약 로그 블록 매핑 테이블에 접근하고자 하는 논리 섹터 번호가 없으면 오프셋 테이블을 참조하여 하이브리드 기법처럼 접근한다. 어떻게 보면 데이터블록의 상태값이 f인 블록들에 대해 기존 하이브리드 방식과 비슷하게 동작하지만 오프셋이라는 별도의 매핑 테이블을 운용하여 빠른 접근이 가능하다.

(그림 11)은 XAST와 FAST의 쓰기 연산을 간략히 비교한 것이다. FAST에서는 7번째 write(1, ...)에서 합병 연산이 수행되지만 제안하는 XAST에서는 발생하지 않는 것을 볼 수 있다. 이러한 부분은 기존 BAST에서의 로그블록 공간 활용의 부족한 점을 FAST에서 개선했듯이 FAST의 임의쓰기 요청이 다발적으로 발생하였을 경우 데이터 블록의 비어있는 공간의 활용을 극대화 시켰다.



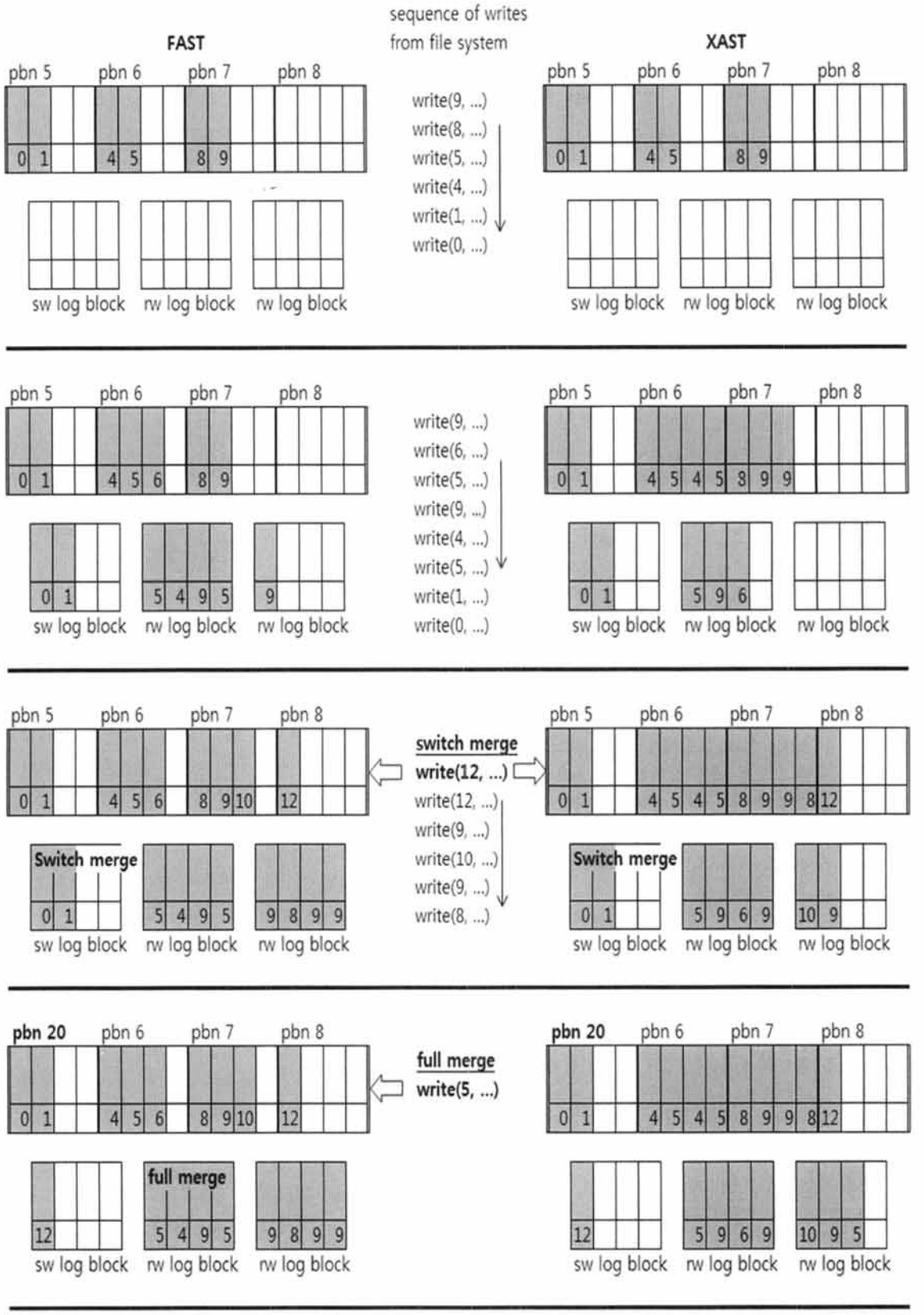
(그림 12) XAST 합병 연산

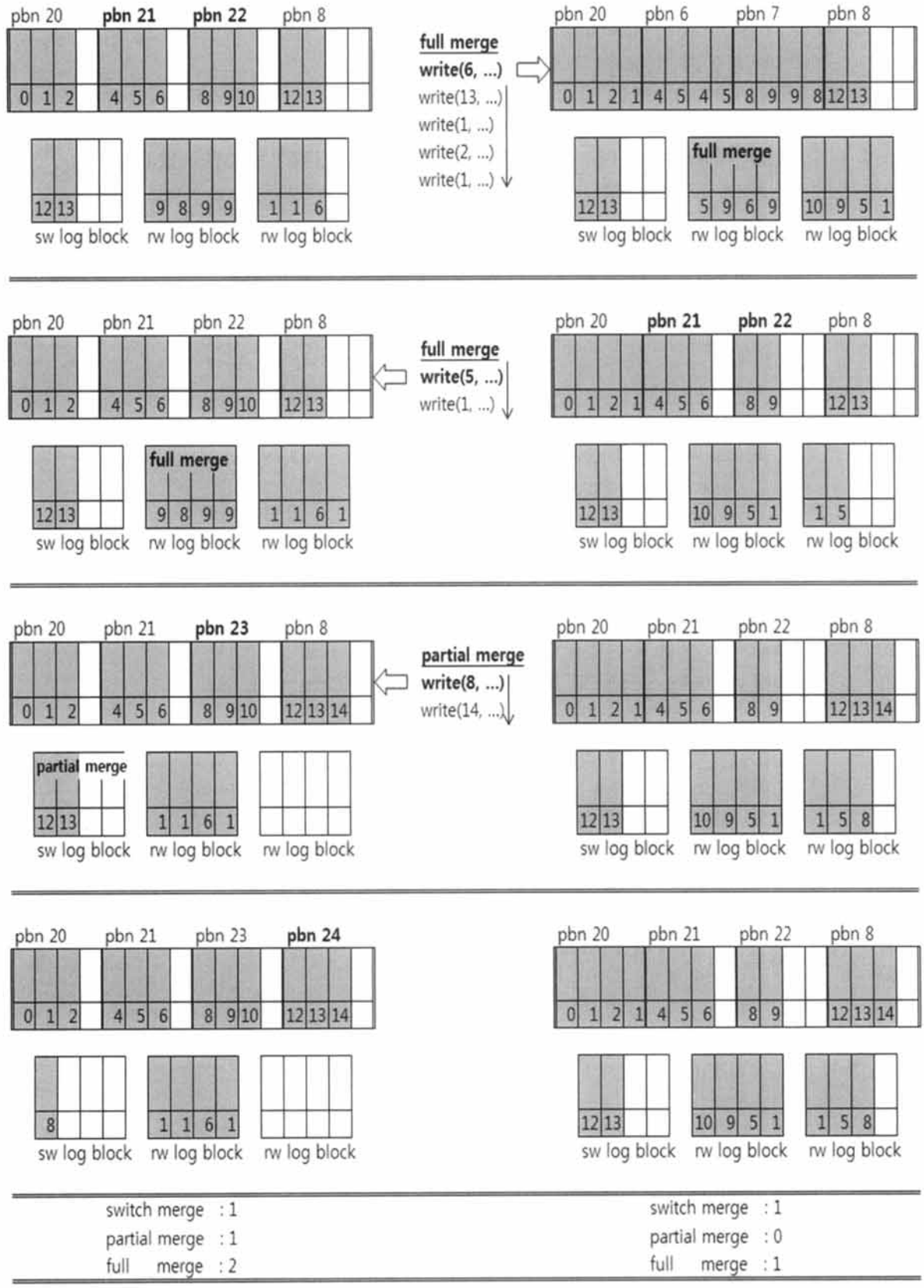
XAST에의 합병 연산은 FAST와 비슷하다. FAST에서는 합병 연산 시 유효하지 않은 섹터를 섹터 매핑 테이블에 -1로 표기함으로써 최신 데이터만을 참조한다. 이러한 작업을 XAST에서는 상태값이 f인 블록에서 덮어쓰기 발생 시 동일하게 수행함으로써 최신 데이터가 데이터 블록과 로그 블록 중 어느 위치에 있는지를 판별한다. (그림 12)에서 XAST의 합변 연산을 보여주며, 논리 섹터 번호 1번은 데이터 블록에 있고, 논리 섹터 번호 2번은 로그 블록에 위치하고 있는 것을 알 수 있다.

XAST에서는 FAST와 비교하여 오프셋 매핑 테이블을 별도로 운영하기 때문에 해당 테이블을 참조해야 하는 오버헤드가 발생한다. 하지만 이러한 매핑 테이블은 SRAM 등과 같이 빠른 액세스가 가능한 RAM에서 운용된다. 즉, 플래시 메모리 삭제연산을 줄이는 것이 플래시 메모리의 성능을 높이는 중요한 요소가 된다.

FTL 알고리즘의 가장 중요한 메타데이터 정보는 주소매핑 정보들이다. 즉, 전원이 나갔을 경우 주소매핑 테이블을 다시 만들 수 있도록 지속적으로 어딘가에 보관해야 하며, 이러한 기법에는 map block method와 per block method가 있다[2]. XAST의 오프셋 매핑 정보들도 이러한 기법들을 기반으로 수행되어질 경우 해당 매핑 정보들의 관리가 가능하다.

(그림 13)는 XAST와 FAST의 상세한 연산에 대해 보여주며, (그림 14)은 XAST의 write 알고리즘을 보여준다.


FAST
sequence of writes
from file system
XAST
pbn 5
pbn 6
pbn 7
pbn 8
write(9, ...)
write(8, ...)
write(5, ...)
write(4, ...)
write(1, ...)
write(0, ...)
sw log block
rw log block
rw log block
write(9, ...)
write(6, ...)
write(5, ...)
write(9, ...)
write(4, ...)
write(5, ...)
write(1, ...)
write(0, ...)
switch merge
write(12, ...)
write(12, ...)
write(9, ...)
write(10, ...)
write(9, ...)
write(8, ...)
Switch merge
pbn 20
full merge
write(5, ...)



(그림 13) XAST와 FAST의 비교 연산

```
Algorithm write(lsn, data)

1  if search for lsn in BMT = false then
2     'write data at offset in the data block of pbn;
3  else
4     if a collision occurs at offset of the data block of pbn = ture then
5        if offset = 0 and search for lsn in RMT = false and pbn is FLUENT in BMT = false then
6           if search for lsn in SMT = false and all sectors is empty in SW log block = true then
7              'write data to the SW log block;
8              'update the SMT;
9           else
10             'merge the SW log block with its corresponding data block;
11          end if
12       else
13          if search for lbn of lsn in SMT = ture then
14             if (last lsn in SMT  + 1) = lsn then
15                'append data to the SW log block;
16                'update the SMT;
17             else
18                'merge the SW log block with its corresponding data block;
19             end if
20          else
21             if pbn of lsn is FLUENT in BMT = true and empty in pbn = true then
22                'write data to the pbn;
23                'update the FMT;
24             else
25                if there are empty sectors in RW log block = true then
26                   'write data to the RW log block;
27                   'update the RMT;
28                   if there are empty sectors of data block with its corresponding RW log block = true then
29                      'pbn is update to FLUENT in BMT;
30                   end if
31                else
32                   'merge the RW log block with its corresponding data block;
33                end if
34             end if
35          end if
36       end if
37    else
38       'write data at offset in the data block of pbn;
39    end if
40 end if
```

(그림 14) XAST의 write 알고리즘

5. 검증 및 평가

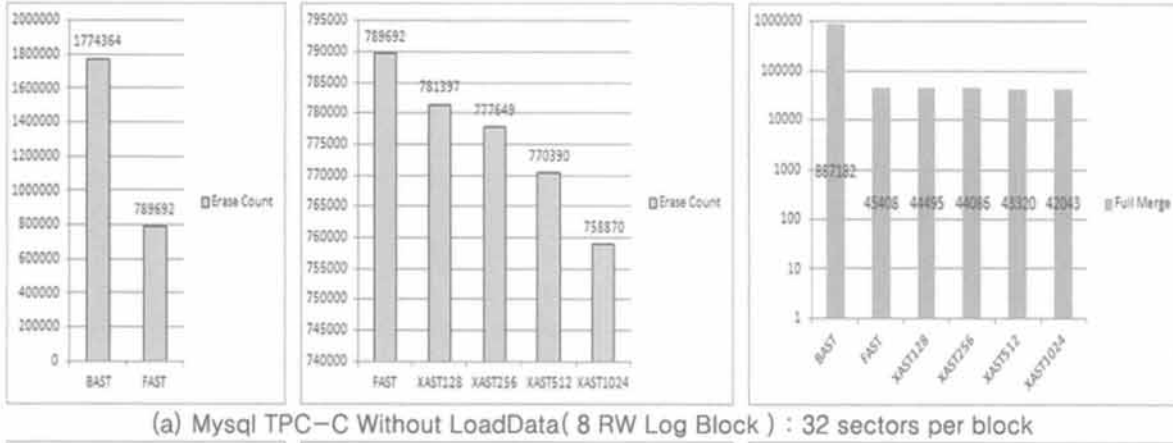
XAST는 임의 쓰기에 대해 데이터 블록 내 비어있는 섹터를 활용하는 것이 핵심 아이디어이기 때문에 오프셋 매핑 테이블의 크기를 얼마만큼 정하는지에 대해 여러 시뮬레이션이 필요하며 이를 수행하도록 한다. 또한, 오프셋 매핑 테이블은 블록 당 섹터의 수와 밀접한 관계를 맺고 있기 때문에 본 실험에서는 블록 내 섹터의 수를 32, 64, 128, 256개로 구분하고, TPC-C 실험들의 로그블록 수는 8, 16개로 나뉘어 실험을 수행한다. 그밖에 실제 사용자 데이터들에 대한 실험은 로그블록의 수를 8개로만 정하여 수행한다. 오프셋 매핑 테이블의 크기는 블록 내 섹터의 수와 로그블록의 수를 고려하여 4가지 경우에 따라 BAST, FAST와의 비교 평가를 수행하며, 시뮬레이션을 위한 실험데이터는 <표 2>와 같다.

실험 데이터 a~d는 리눅스 기반에서 추출한 데이터이며, 그 외 모든 실험 데이터는 윈도우 기반에서 얻어진 데이터이다. 실험 데이터 a~d는 mysql 5.0 데이터베이스 기반에 Direct IO를 적용하여 TPC-C 벤치마킹을 수행하여 얻은 데이터이다. 실험 데이터 e~h는 오라클9i 데이터베이스 기반에 TCP-C 벤치마킹을 수행하면서 얻는 데이터이다. TPC-C벤치마킹은 별도의 디스크에서 수행하여 얻은 데이터이며, 실험데이터 c, d, g, h는 TPC-C벤치마킹 수행 전 로드(Insert)되는 데이터 레코드들을 포함하고 있고, a, b, e, f는 포함하지 않는다. 블록 내 데이터의 존재유무에 대해 실험을 구분한 이유는 최근의 SSD는 Over-provisioning 기능을 제공하여 비어있는 블록들을 사전에 준비하는 등 버퍼 사용과 함께 SSD의 성능을 극대화 시키고 있다[18]. 즉, Over-provisioning 기능을 제공받지 못하는 가정하에 블록 내 데이터가 존재하는 상태에서의 실험을 구분하였다. 실험데이터 i와 j는 사용자들이 실제 사용하고 있는 컴퓨터에서 얻는 데이터이며, 주로 엑셀작업과 웹서핑을 하는 기업 내 재무팀 사용자들에게서 수집한 데이터이다.

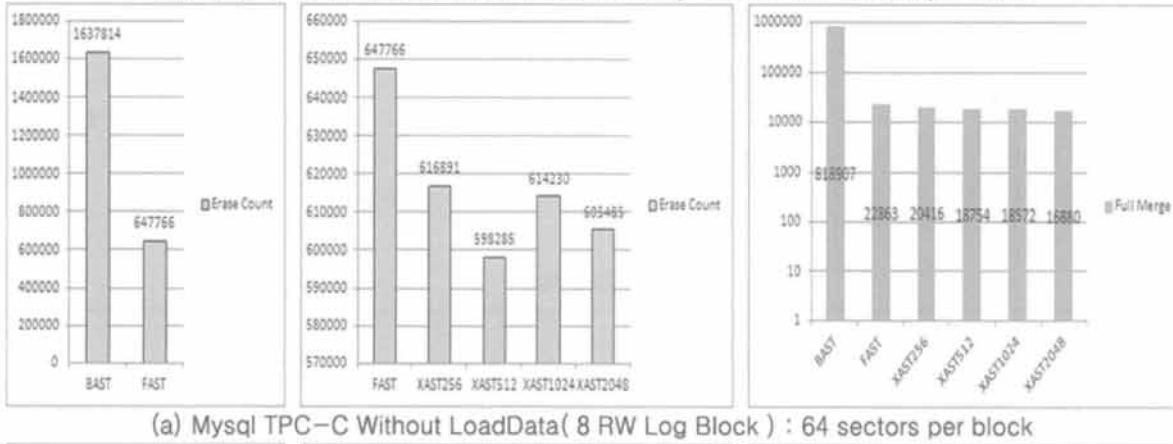
리눅스 기반에서의 데이터 추출은 리눅스의 요청 큐 연산을 관찰할 수 있는 blktrace를 사용하였고, 윈도우 기반에서의 데이터 추출은 DiskMon을 사용하였다. DiskMon은 Microsoft Sysinternals에서 제공한다[16].

〈표 2〉 실험 데이터

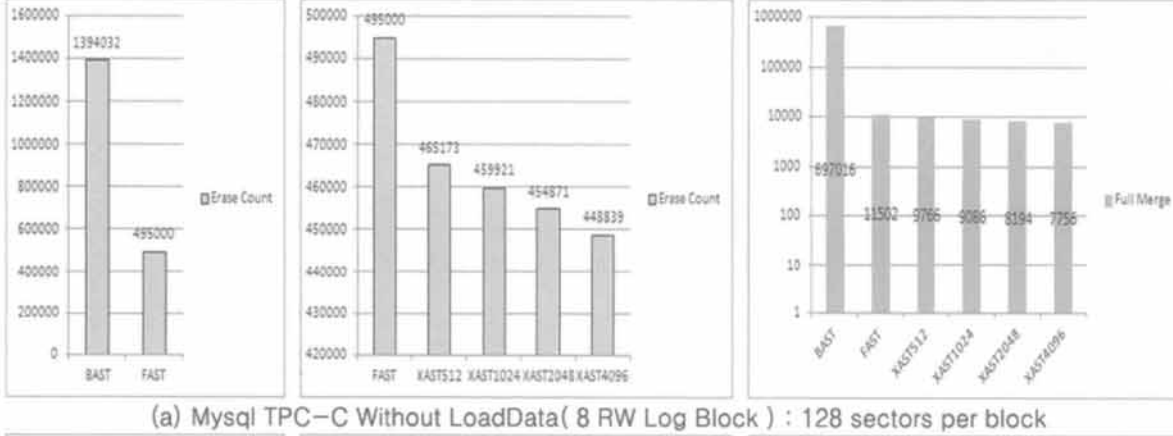
구분	로그블록수	설명	추출 데이터 수
(a) Mysql TPC-C Without LoadData	8	리눅스 기반 TPC-C 추출 데이터 (Insert 데이터 미포함)	1,571,478
(b) Mysql TPC-C Without LoadData	16		
(c) Mysql TPC-C With LoadData	8	리눅스 기반 TPC-C 추출 데이터 (Insert 데이터 포함)	1,869,466
(d) Mysql TPC-C With LoadData	16		
(e) Oracle TPC-C Without LoadData	8	오라클 기반 TPC-C 추출 데이터 (Insert 데이터 미포함)	1,135,882
(f) Oracle TPC-C Without LoadData	16		
(g) Oracle TPC-C With LoadData	8	오라클 기반 TPC-C 추출 데이터 (Insert 데이터 포함)	1,251,537
(h) Oracle TPC-C With LoadData	16		
(i) Real Data 1	8	윈도우 기반 실제 사용자 데이타	483,909
(j) Real Data 2	8	윈도우 기반 실제 사용자 데이타	199,811



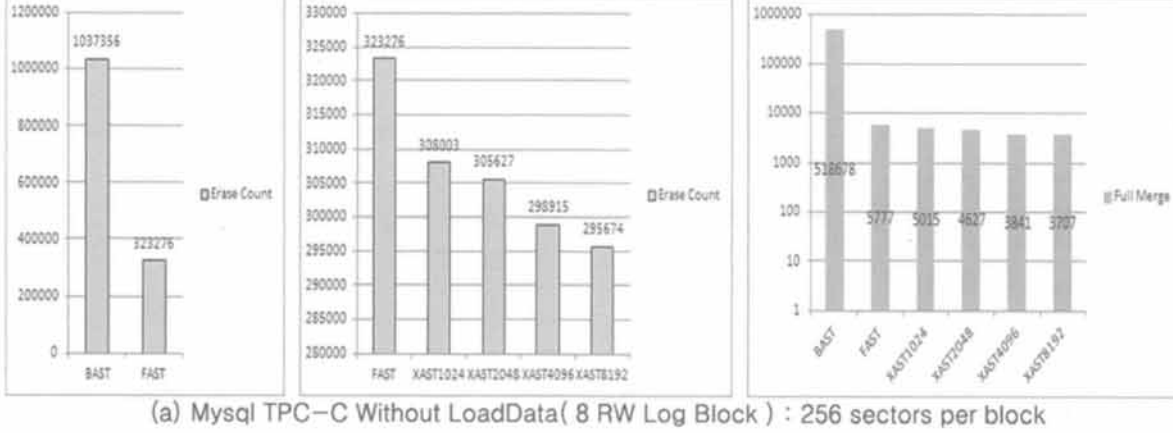
(a) Mysql TPC-C Without LoadData(8 RW Log Block) : 32 sectors per block



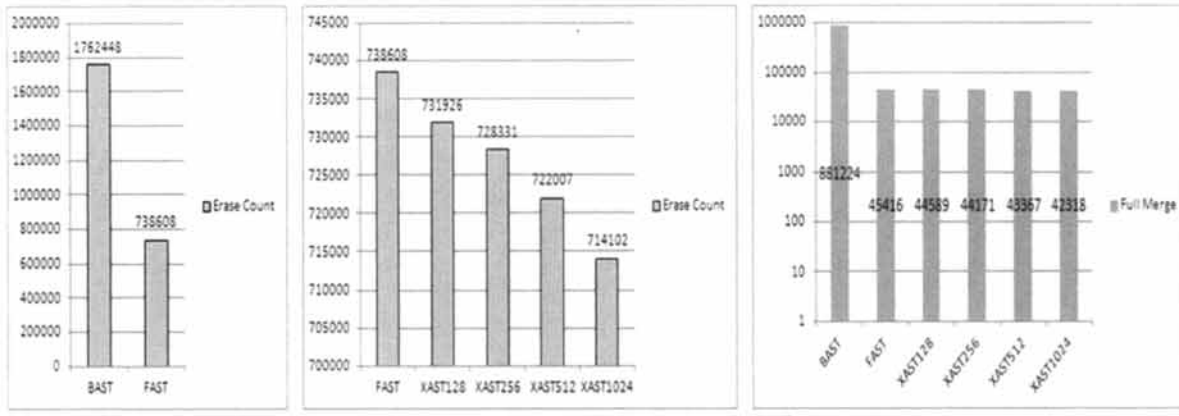
(a) Mysql TPC-C Without LoadData(8 RW Log Block) : 64 sectors per block



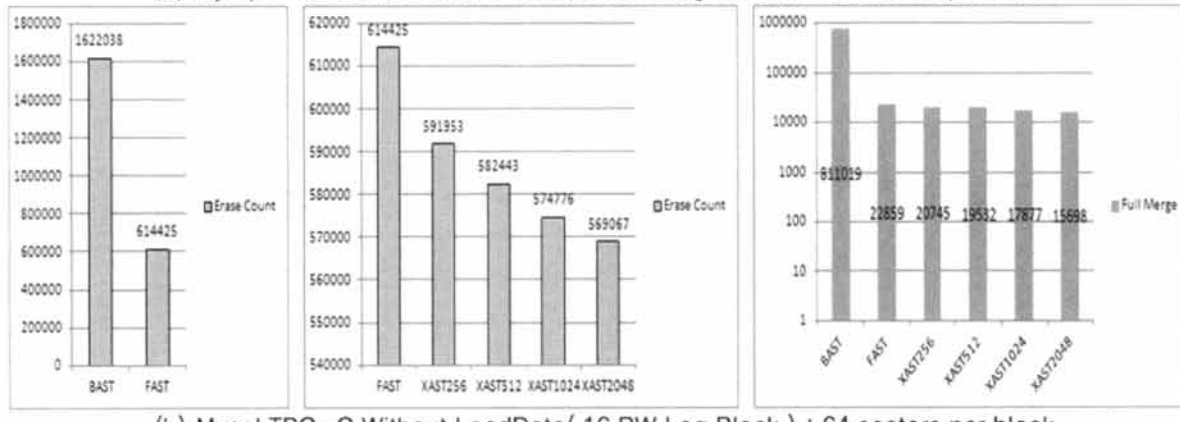
(a) Mysql TPC-C Without LoadData(8 RW Log Block) : 128 sectors per block



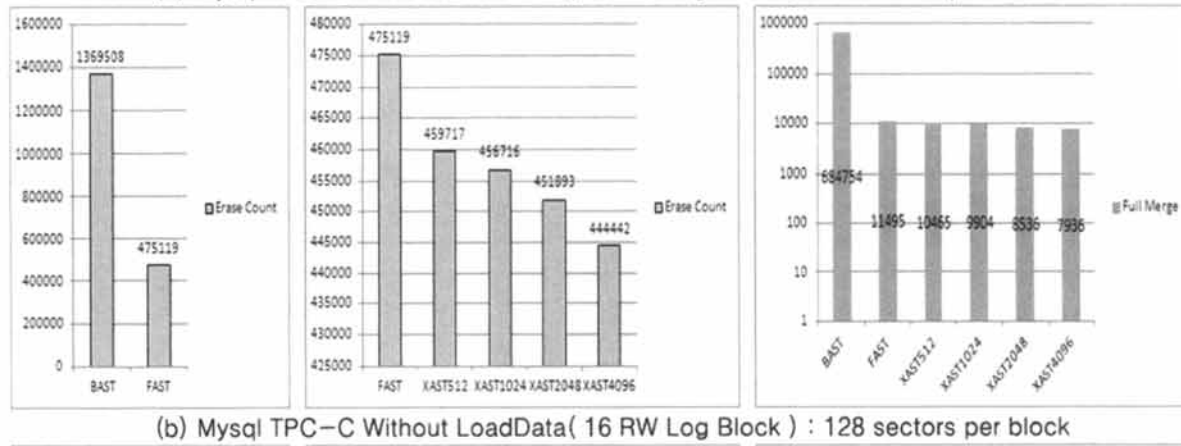
(a) Mysql TPC-C Without LoadData(8 RW Log Block) : 256 sectors per block



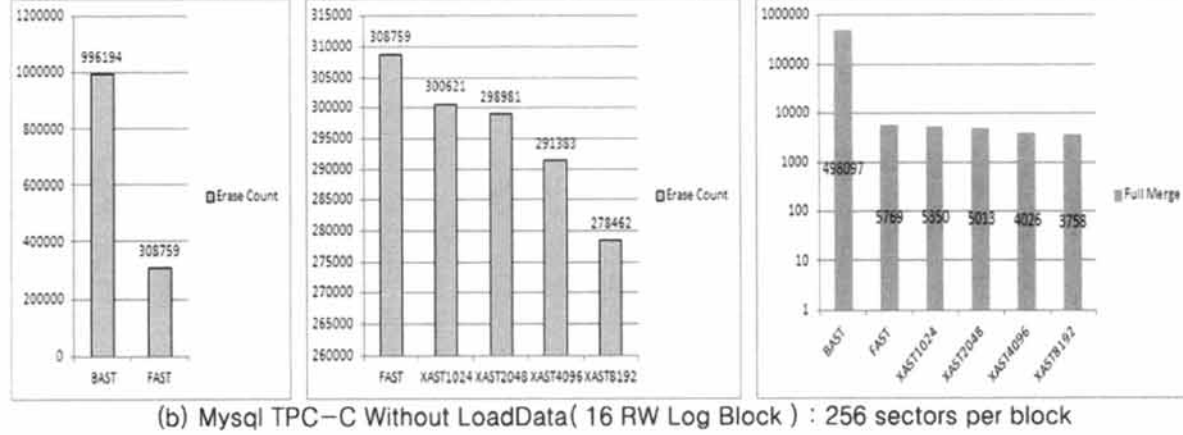
(b) Mysql TPC-C Without LoadData(16 RW Log Block) : 32 sectors per block



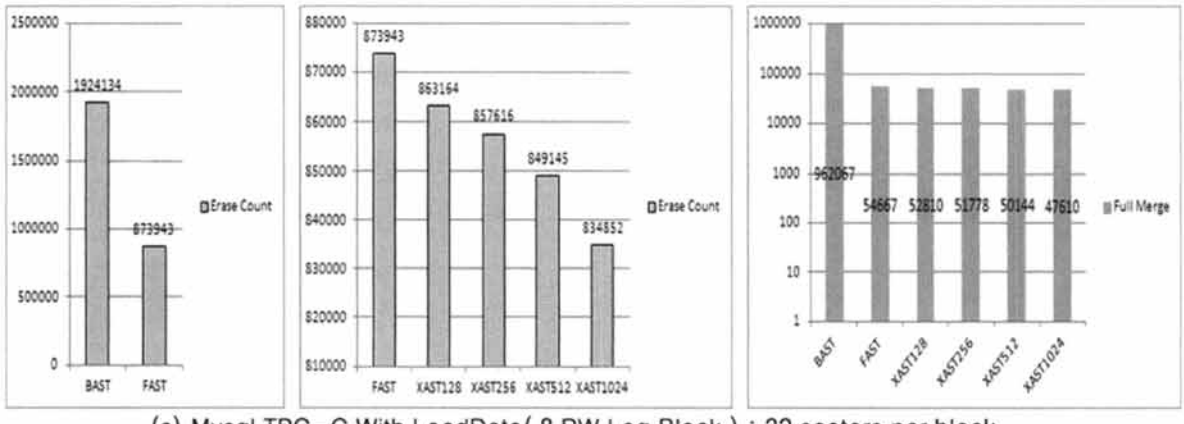
(b) Mysql TPC-C Without LoadData(16 RW Log Block) : 64 sectors per block



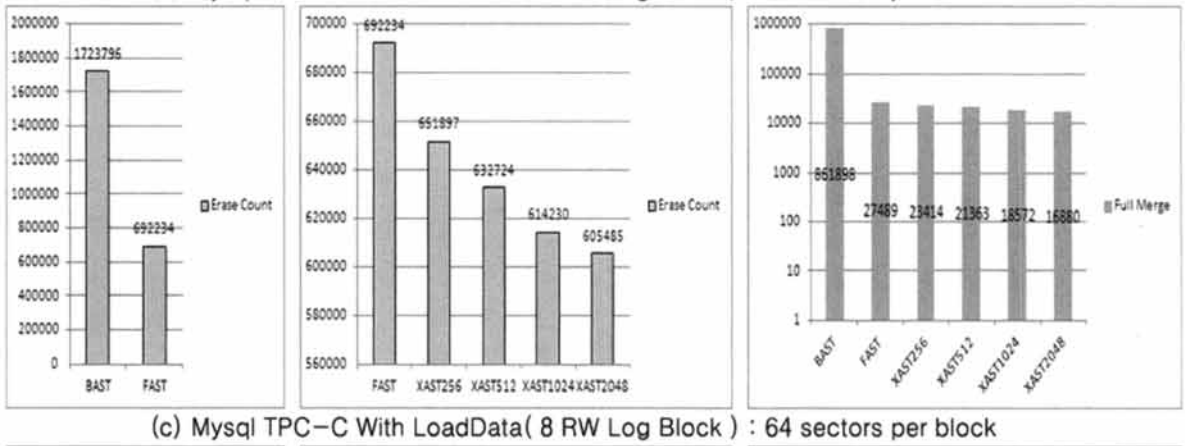
(b) Mysql TPC-C Without LoadData(16 RW Log Block) : 128 sectors per block



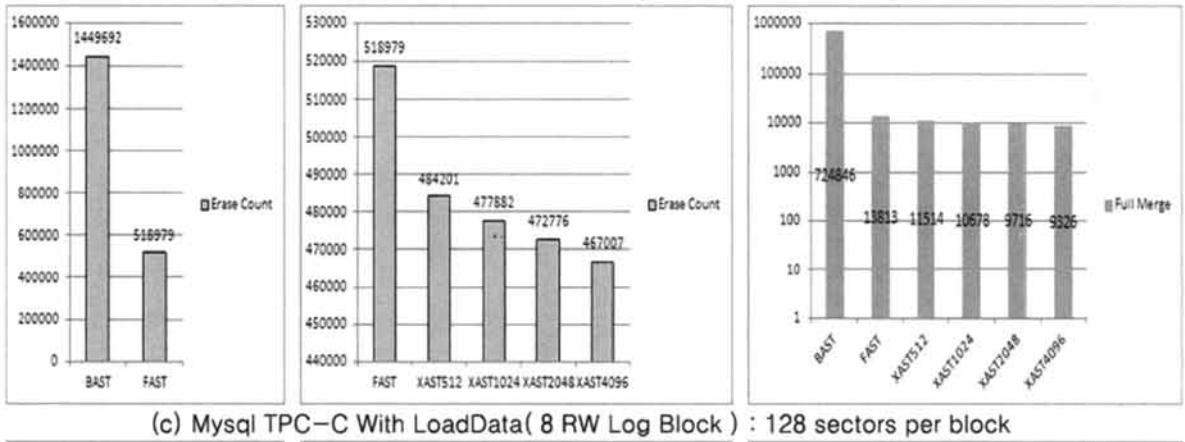
(b) Mysql TPC-C Without LoadData(16 RW Log Block) : 256 sectors per block



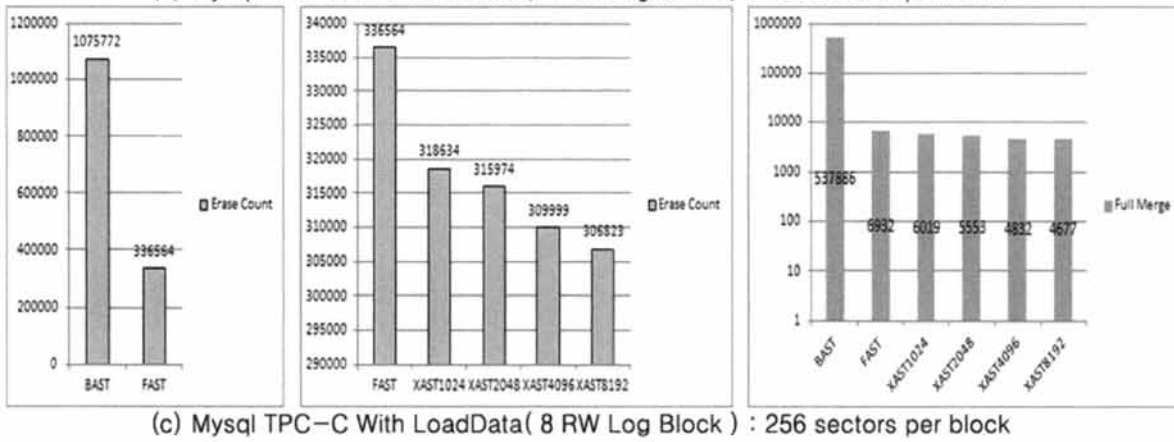
(c) Mysql TPC-C With LoadData(8 RW Log Block) : 32 sectors per block



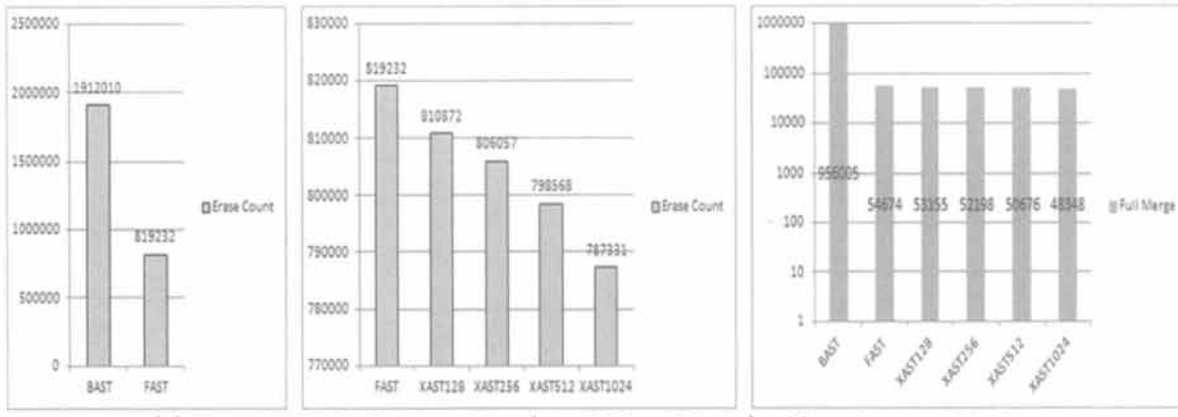
(c) Mysql TPC-C With LoadData(8 RW Log Block) : 64 sectors per block



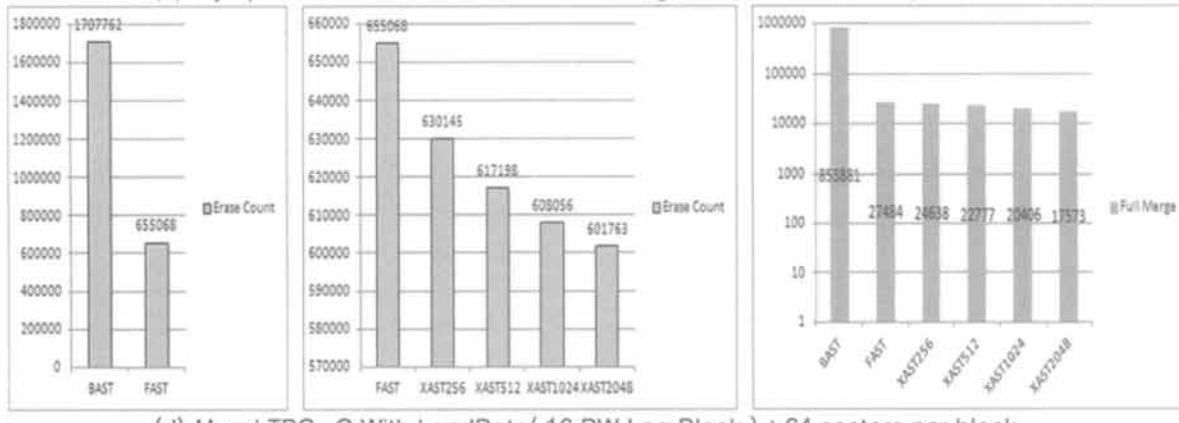
(c) Mysql TPC-C With LoadData(8 RW Log Block) : 128 sectors per block



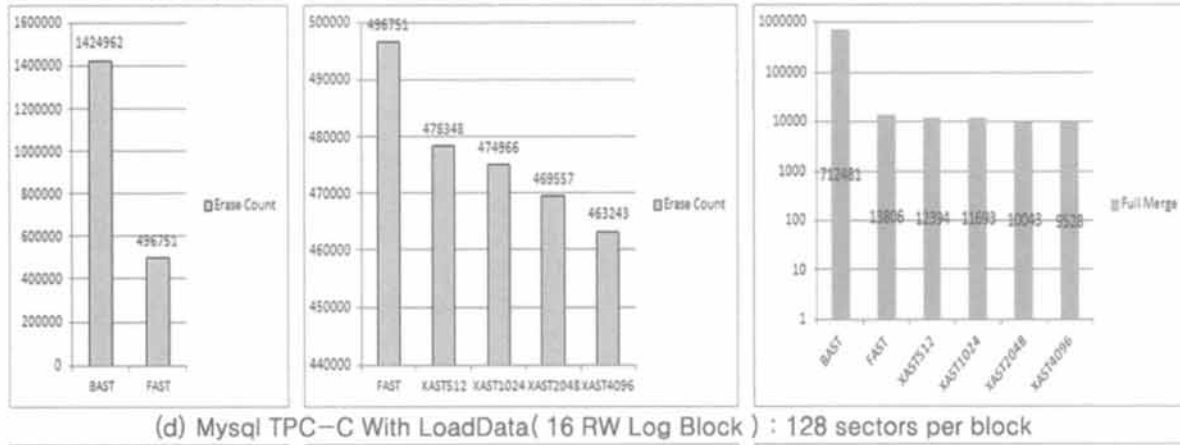
(c) Mysql TPC-C With LoadData(8 RW Log Block) : 256 sectors per block



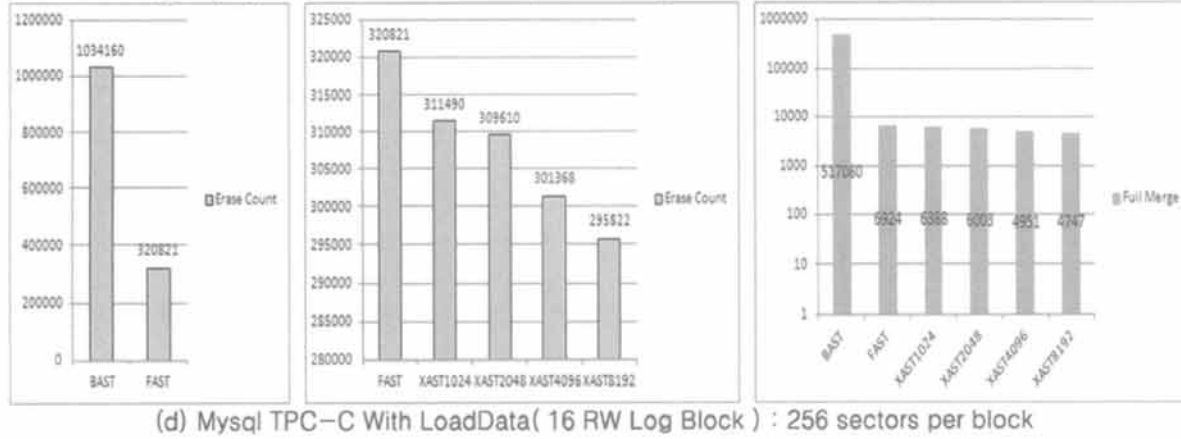
(d) Mysql TPC-C With LoadData(16 RW Log Block) : 32 sectors per block



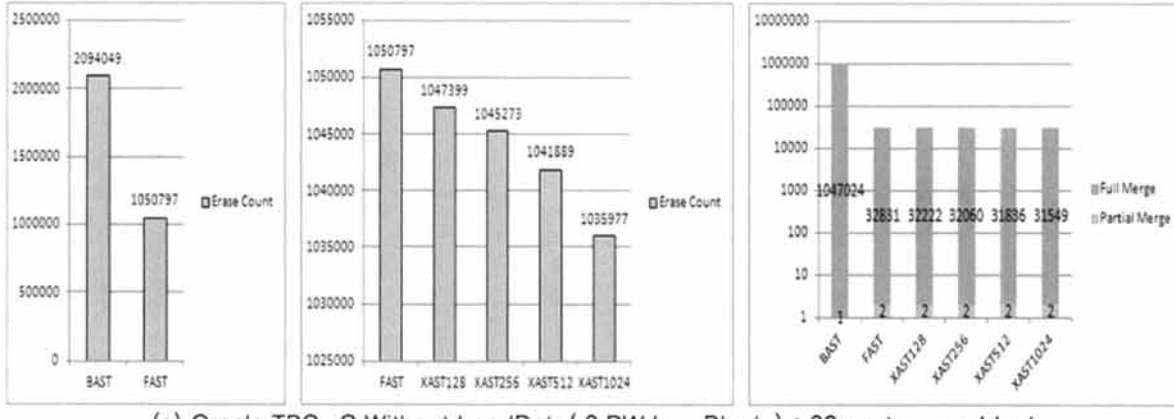
(d) Mysql TPC-C With LoadData(16 RW Log Block) : 64 sectors per block



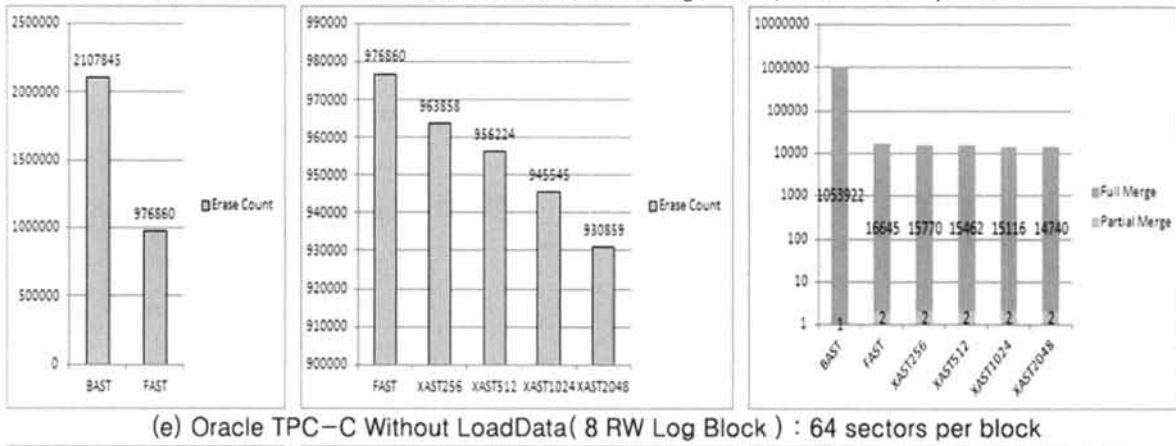
(d) Mysql TPC-C With LoadData(16 RW Log Block) : 128 sectors per block



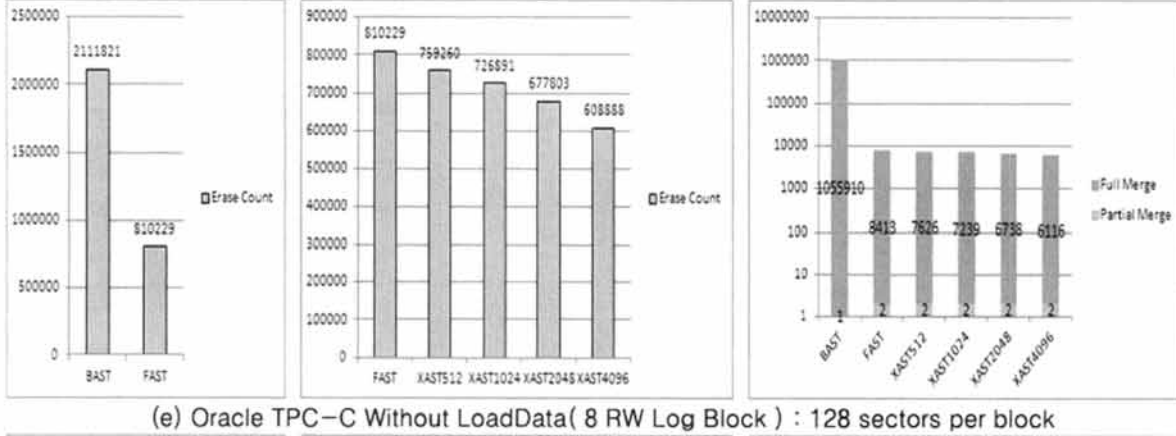
(d) Mysql TPC-C With LoadData(16 RW Log Block) : 256 sectors per block



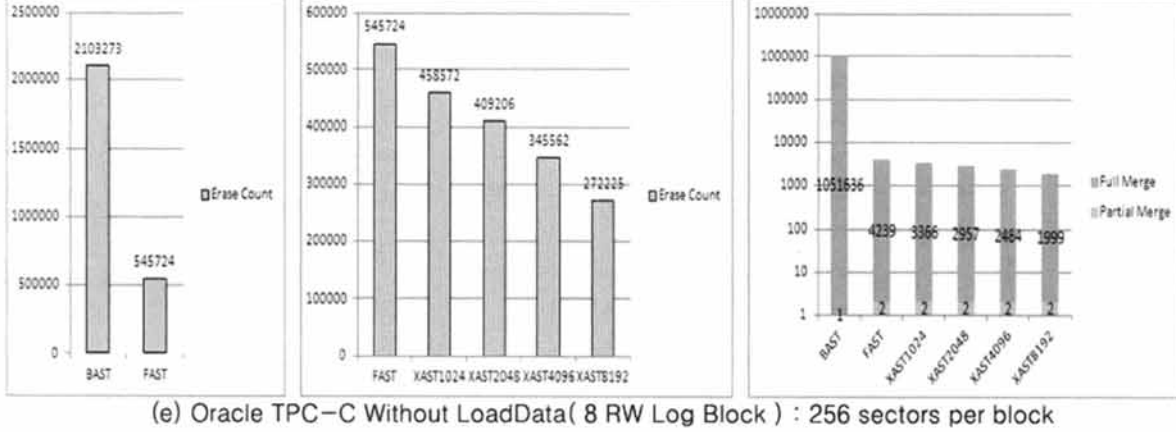
(e) Oracle TPC-C Without LoadData(8 RW Log Block) : 32 sectors per block



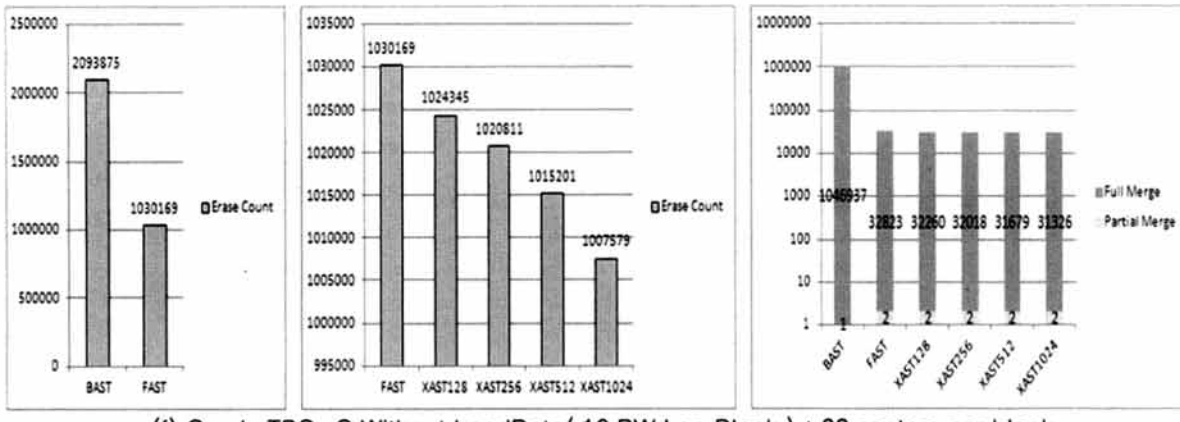
(e) Oracle TPC-C Without LoadData(8 RW Log Block) : 64 sectors per block



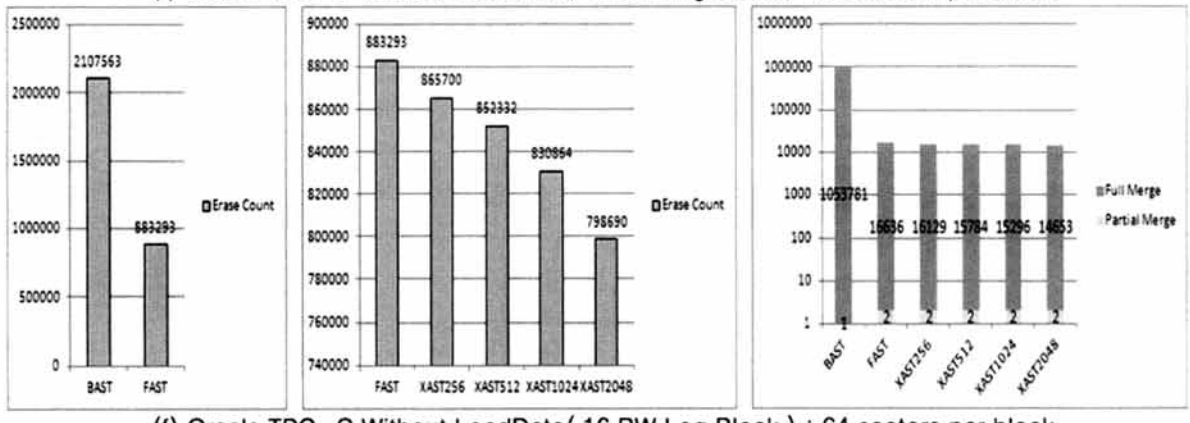
(e) Oracle TPC-C Without LoadData(8 RW Log Block) : 128 sectors per block



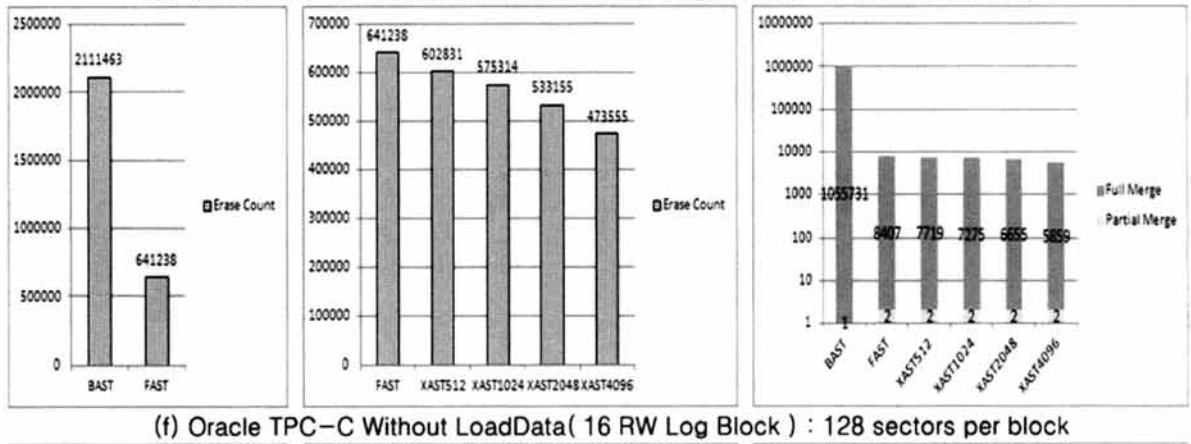
(e) Oracle TPC-C Without LoadData(8 RW Log Block) : 256 sectors per block



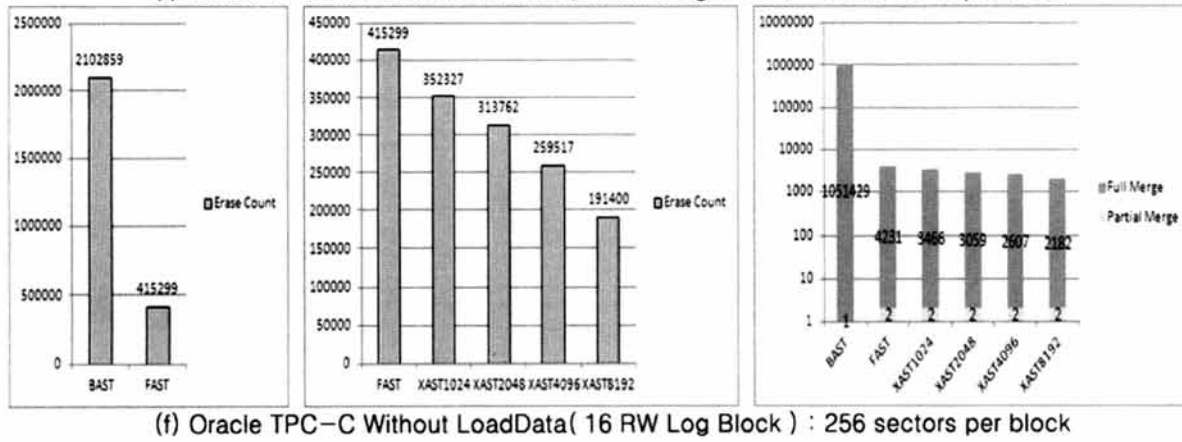
(f) Oracle TPC-C Without LoadData(16 RW Log Block) : 32 sectors per block



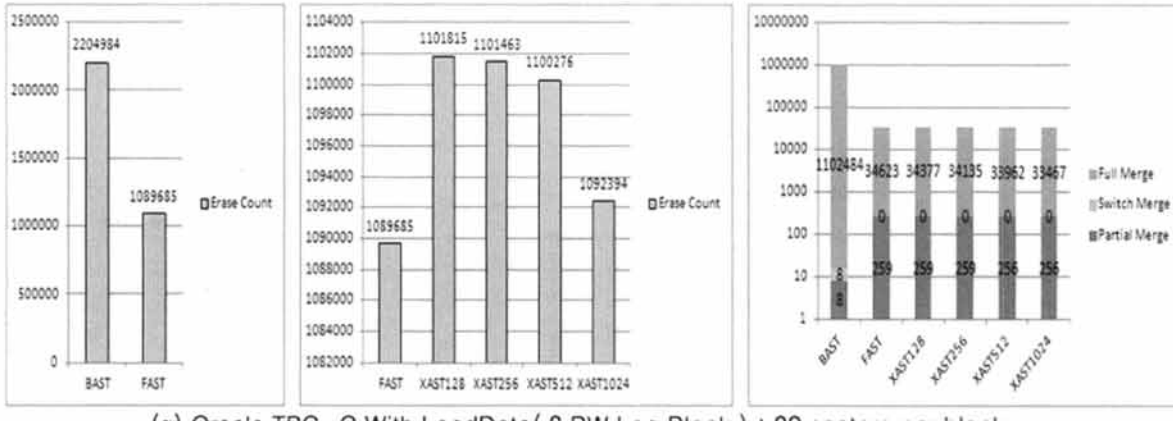
(f) Oracle TPC-C Without LoadData(16 RW Log Block) : 64 sectors per block



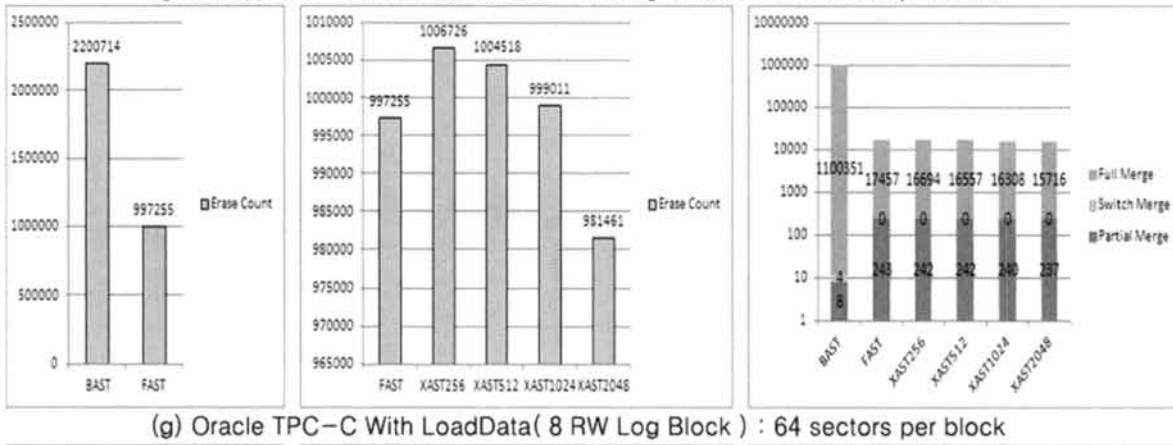
(f) Oracle TPC-C Without LoadData(16 RW Log Block) : 128 sectors per block



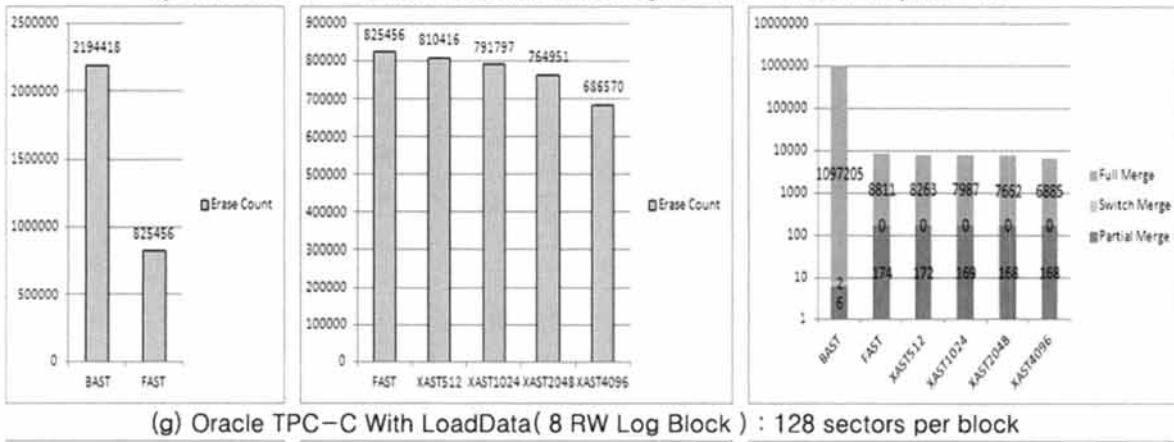
(f) Oracle TPC-C Without LoadData(16 RW Log Block) : 256 sectors per block



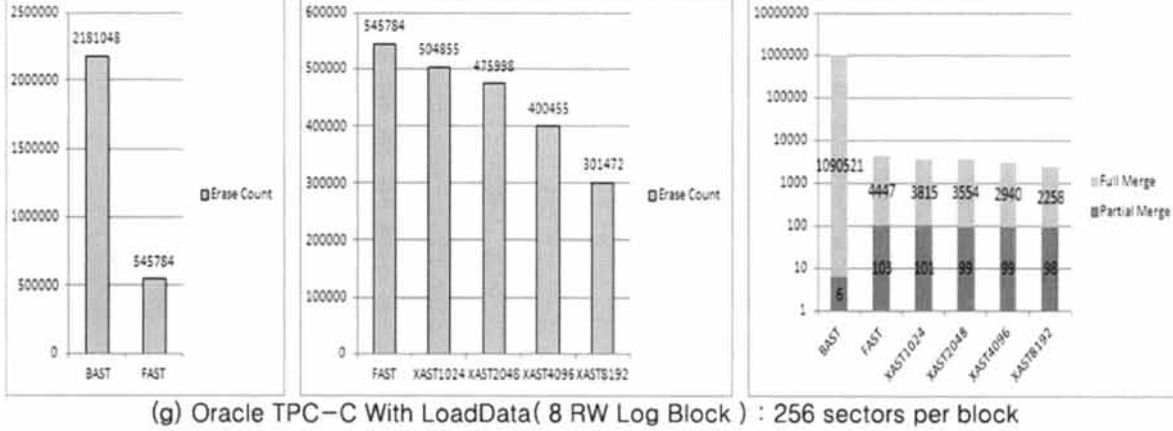
(g) Oracle TPC-C With LoadData(8 RW Log Block) : 32 sectors per block



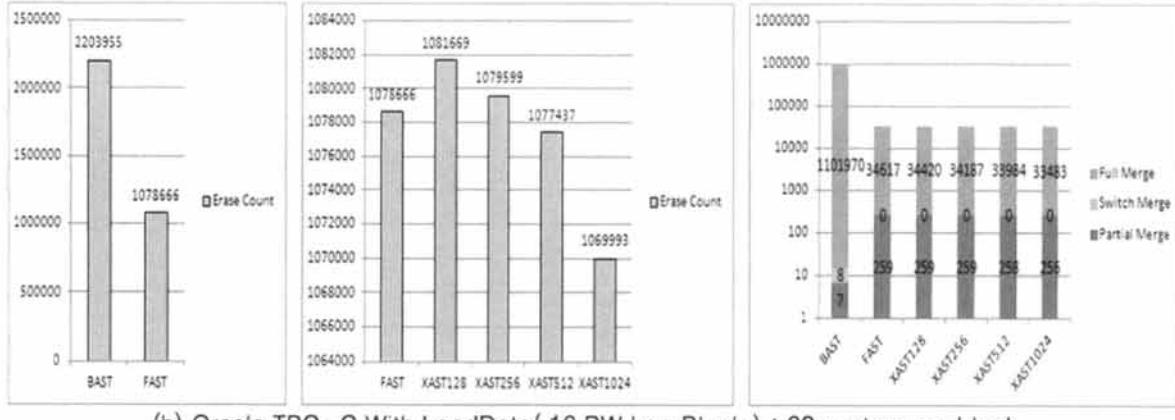
(g) Oracle TPC-C With LoadData(8 RW Log Block) : 64 sectors per block



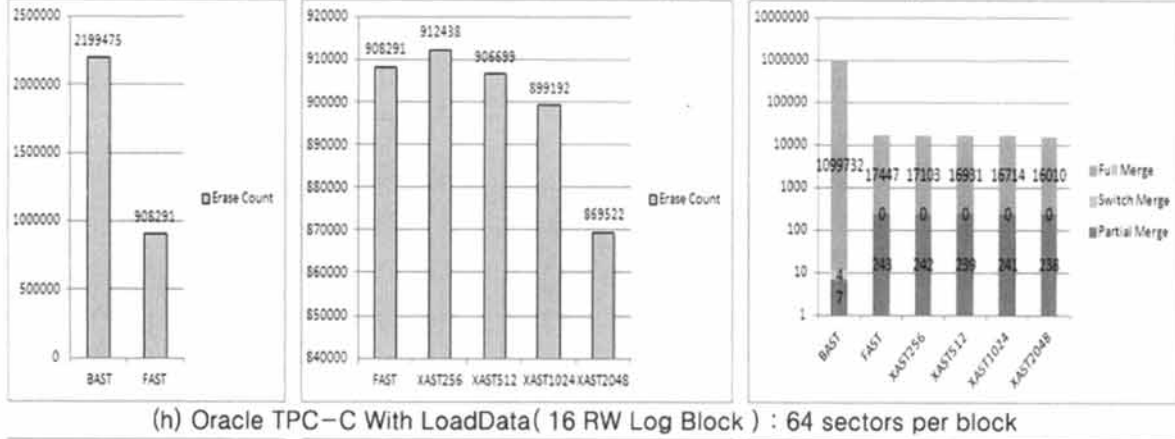
(g) Oracle TPC-C With LoadData(8 RW Log Block) : 128 sectors per block



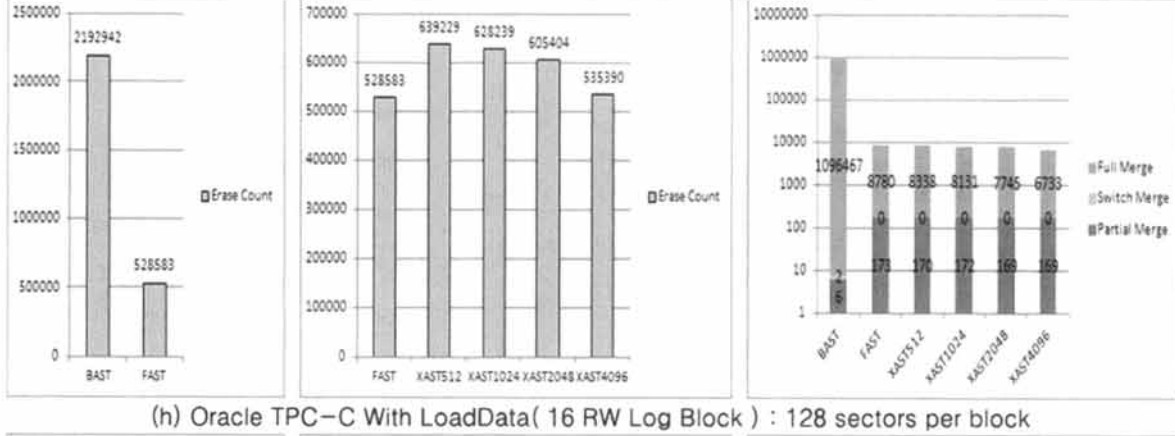
(g) Oracle TPC-C With LoadData(8 RW Log Block) : 256 sectors per block



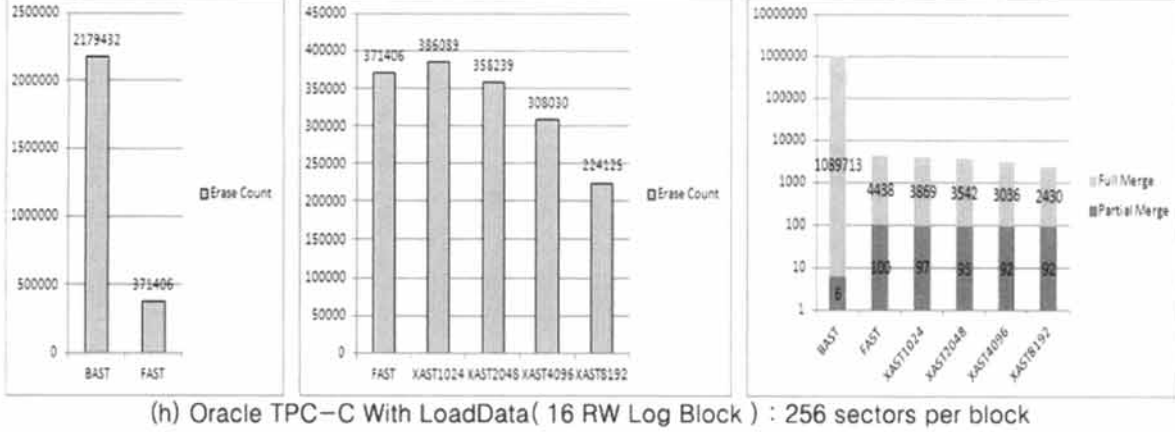
(h) Oracle TPC-C With LoadData(16 RW Log Block) : 32 sectors per block



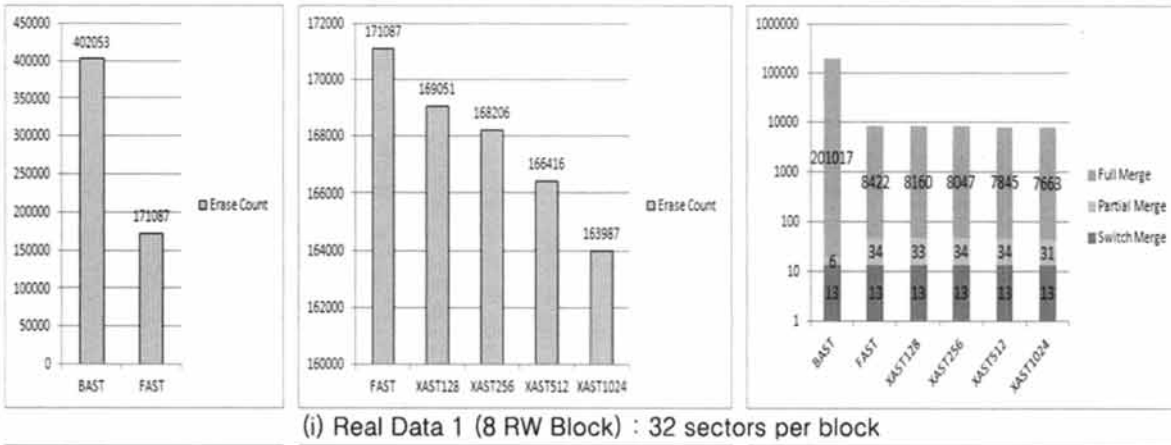
(h) Oracle TPC-C With LoadData(16 RW Log Block) : 64 sectors per block



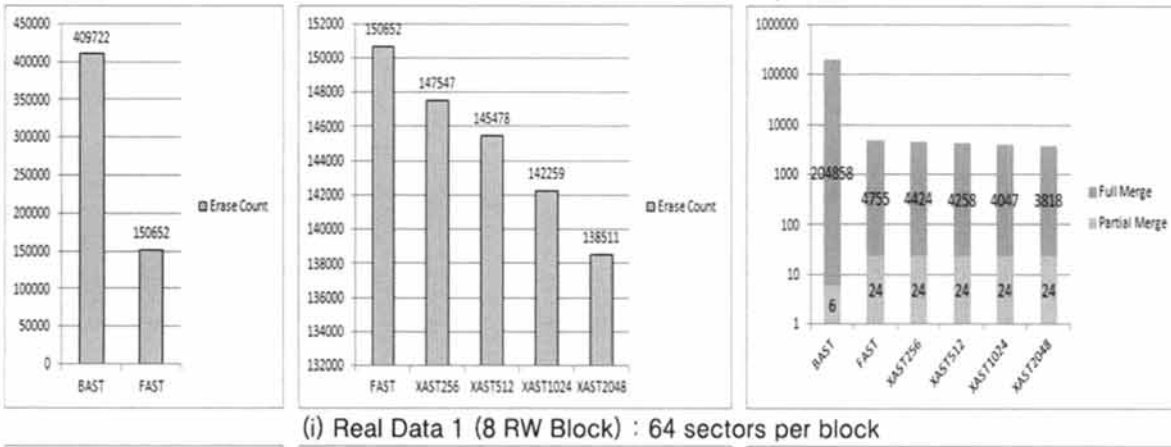
(h) Oracle TPC-C With LoadData(16 RW Log Block) : 128 sectors per block



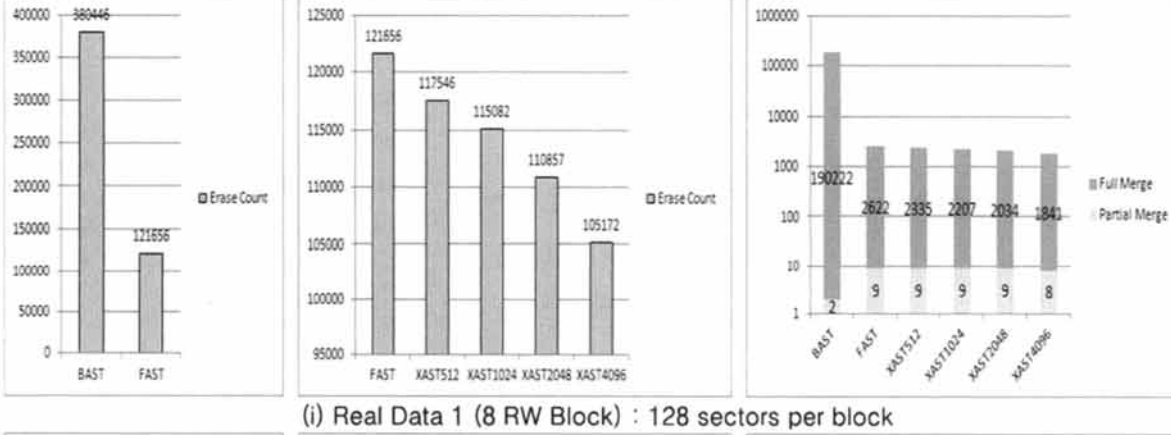
(h) Oracle TPC-C With LoadData(16 RW Log Block) : 256 sectors per block



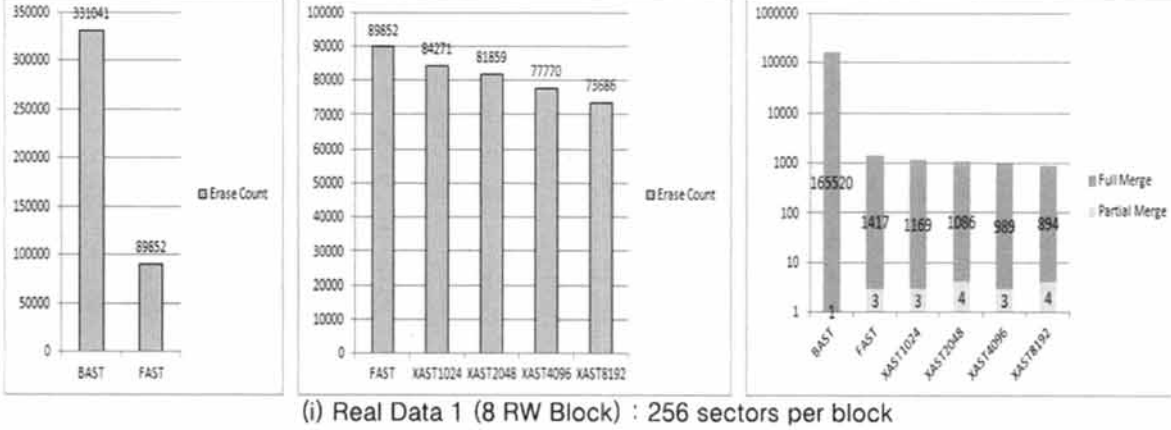
(i) Real Data 1 (8 RW Block) : 32 sectors per block



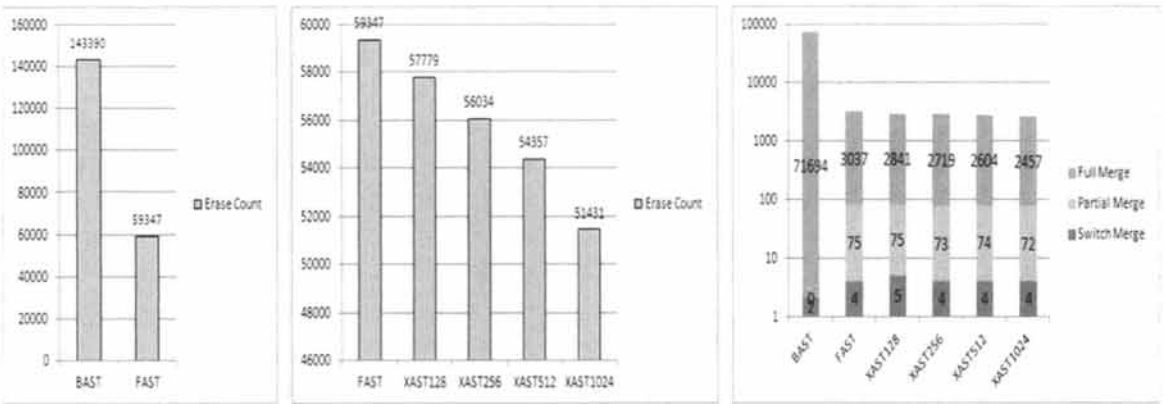
(i) Real Data 1 (8 RW Block) : 64 sectors per block



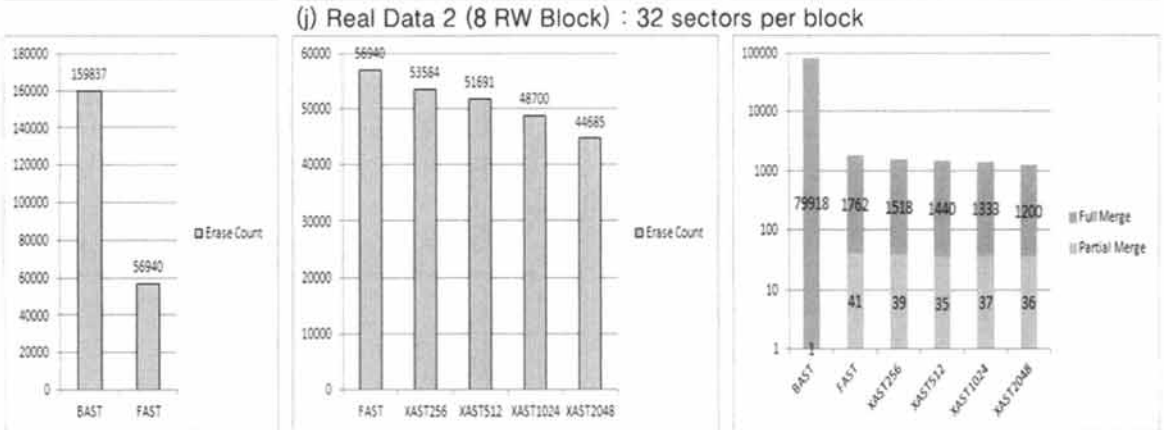
(i) Real Data 1 (8 RW Block) : 128 sectors per block



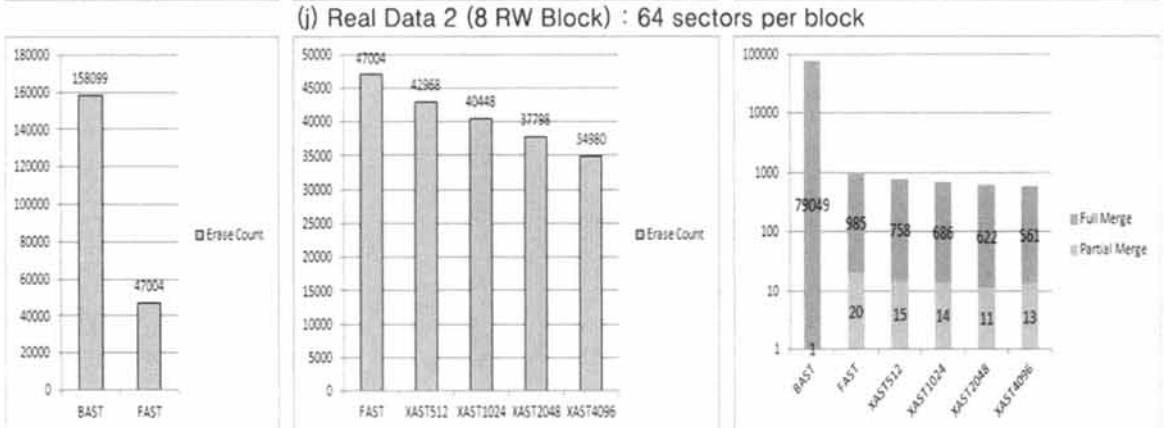
(i) Real Data 1 (8 RW Block) : 256 sectors per block



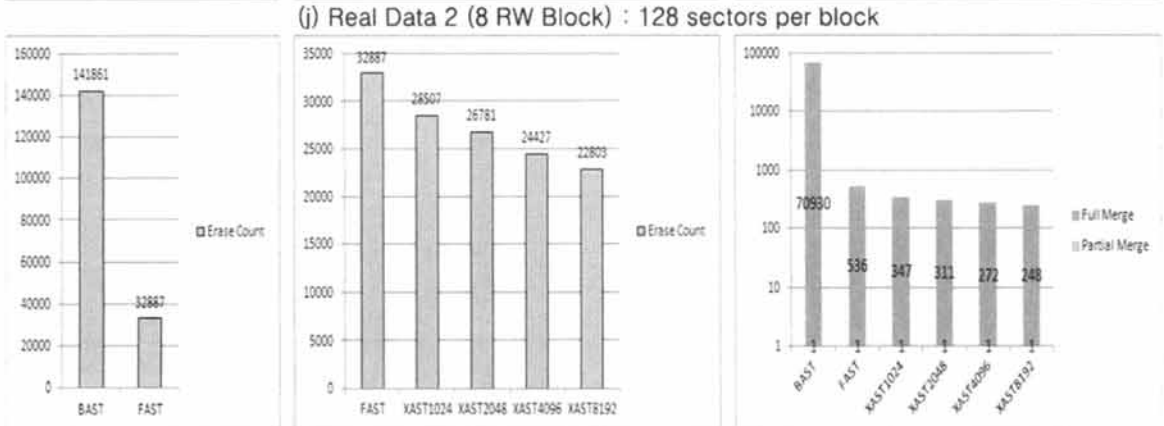
(j) Real Data 2 (8 RW Block) : 32 sectors per block



(j) Real Data 2 (8 RW Block) : 64 sectors per block



(j) Real Data 2 (8 RW Block) : 128 sectors per block



(j) Real Data 2 (8 RW Block) : 256 sectors per block

(그림 15) 성능 평가 결과

<표 3>은 각각의 실험들에 대한 플래시 메모리의 정보를 나타낸다.

〈표 3〉 실험들에 대한 플래시 메모리 정보

구분	블록 당 섹터 수	총 블록 수	총 섹터 수
OS : Linux (a~b) Mysql TPC-C Without LoadData	32	66,461	2,126,752
	64	37,871	2,423,744
	128	20,961	2,683,008
	256	11,040	2,826,240
OS : Linux (c~d) Mysql TPC-C With LoadData	32	67,196	2,150,272
	64	38,471	2,462,144
	128	21,478	2,749,184
	256	11,490	2,941,440
OS : Windows XP (e~f) Oracle TPC-C Without LoadData	32	76,408	2,445,056
	64	39,591	2,533,824
	128	20,439	2,616,192
	256	10,456	2,676,736
OS : Windows XP (g~h) Oracle TPC-C Without LoadData	32	54,360	1,739,520
	64	28,882	1,848,448
	128	15,299	1,958,272
	256	9,113	2,332,928
OS : Windows XP (i) Real Data 1	32	157,174	5,029,568
	64	109,756	7,024,384
	128	74,139	9,489,792
	256	45,873	11,743,488
OS : Windows XP (j) Real Data 2	32	86,130	2,756,160
	64	58,283	3,730,112
	128	39,282	5,028,096
	256	22,781	5,831,936

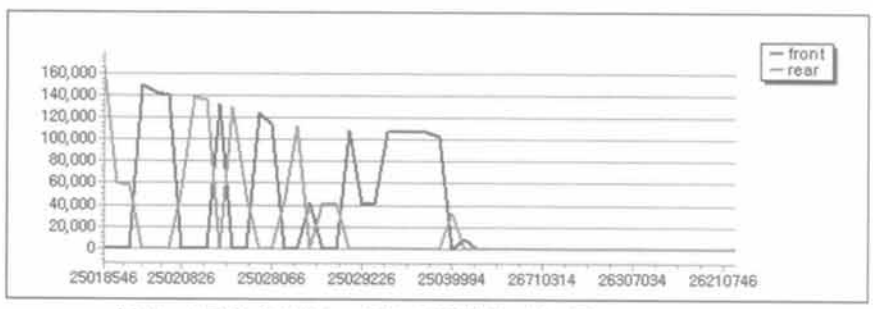
(그림 15)의 실험그래프에서 XAST 뒤의 숫자는 오프셋 매핑 테이블에서 관리되어지는 섹터의 수를 의미하며 1개 섹터에 대한 정보는 4Bytes면 충분하다. 예를 들면 <표 3>에서 실험(c~d)의 섹터 수가 64개였을 경우에는 총 물리 블록의 수가 38,471이기 때문에 물리 블록의 정보를 저장하기 위해 2Bytes가 필요하다. 그리고 블록 내 섹터 오프셋에 대한 위치 정보는 1Bytes면 충분하다. 즉, 1개의 섹터를 오프셋 매핑 테이블에 저장하기 위해 총 3Bytes가 필요하며, 오프셋 섹터 정보를 1024개를 관리한다고 하면 3Bytes * 1024 = 3072Bytes 용량이 된다. 실험(c~d)에서 섹터의 크기를 2Kbytes라 가정하고, 총 섹터 수를 가지고 총 용량을 계산하면 38,471Mbit이다. 이는 약 4.8Gbytes로써 3072Bytes의 오프셋 매핑 테이블을 별도의 램에서 운용하는 것은 총 용량 대비 적은 수준의 크기이다.

<표 4>는 FAST와 비교한 XAST의 성능 분석표이다. 실험(g)는 섹터의 수가 32개일 경우 기존 FAST와 비교하여 더 많은 삭제연산이 발생하였다. 이러한 이유는 로그블록과 데이터블록의 관계가 FAST와 비교하여 XAST는 더욱 커지기 때문에 발생하였다. 예를 들면 물리 블록 번호 4, 5, 6번의 덮어쓰기 대한 데이터를 갖고 있는 4개의 섹터로 구성된 로그블록이 있다고 가정하자. FAST에서는 물리 번호 6번에 덮어쓰기가 발생하면 로그블록에 쓰기연산을 수행하고, 로그블록은 4, 5, 6번의 물리 블록과 관계를 맺고 합병연산 발생 시 4번의 삭제연산이 발생한다. 하지만, XAST에서 물리 번호 6번 블록이 f인 경우에는 자신의 블록 내에 쓰기 연산을 수행한다. 이후 8번 물리 블록에 덮어쓰기가 발생하고 해당 블록의 상태가 u인 경우에 로그블록 쓰기 연산을 수행하며, 로그 블록은 물리 블록 4, 5, 6, 8번과 관계를 맺고 있다. 즉, 합병 연산 수행 시 5번의 삭제연산이 수행된다. 하지만 <표 4>에서 섹터의 수가 늘어날수록 XAST는 FAST와 비교하여 성능향상을 나타내고 있다.

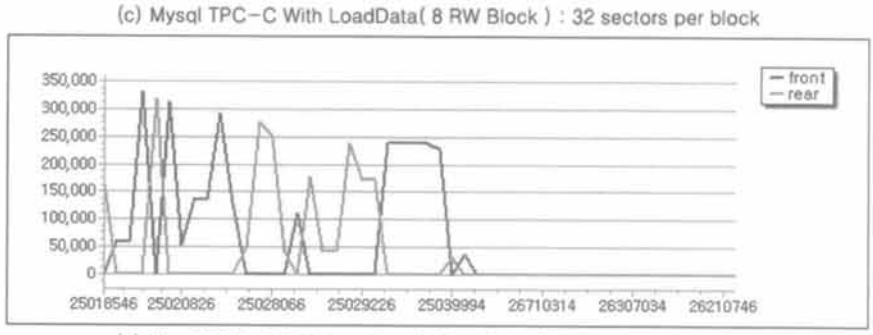
〈표 4〉 FAST 대비 XAST의 삭제 연산에 대한 성능 개선 결과

구분	pages	xast(k/2)	xast(k)	xast(k*2)	xast(k*4)
(a) Mysql TPC-C Without LoadData (8RW Log Block)	32	1.05%	1.52%	2.44%	3.90%
	64	4.77%	7.64%	5.18%	6.53%
	128	6.03%	7.09%	8.11%	9.33%
	256	4.72%	5.46%	7.54%	8.54%
(c) Mysql TPC-C With LoadData (8RW Log Block)	32	1.23%	1.87%	2.84%	4.47%
	64	5.83%	8.60%	11.27%	12.53%
	128	6.70%	7.91%	8.92%	10.01%
	256	5.33%	6.12%	7.89%	8.84%
(e) Oracle TPC-C Without LoadData (8RW Log Block)	32	0.32%	0.53%	0.85%	1.41%
	64	1.33%	2.11%	3.21%	4.70%
	128	6.29%	10.29%	16.34%	24.85%
	256	15.97%	25.02%	36.68%	50.12%
(g) Oracle TPC-C With LoadData (8RW Log Block)	32	-1.11%	-1.08%	-0.97%	-0.25%
	64	-0.95%	-0.73%	0.18%	1.58%
	128	1.82%	4.08%	7.33%	16.83%
	256	7.50%	12.79%	26.63%	44.76%
(i) Real Data 1 (8RW Log Block)	32	1.19%	1.68%	2.73%	4.15%
	64	2.06%	3.43%	5.57%	8.06%
	128	3.38%	5.40%	8.88%	13.55%
	256	6.21%	8.90%	13.45%	17.99%
(j) Real Data 2 (8RW Log Block)	32	2.64%	5.58%	8.41%	13.34%
	64	5.89%	9.22%	14.47%	21.52%
	128	8.59%	13.95%	19.59%	25.58%
	256	13.32%	18.57%	25.72%	30.66%

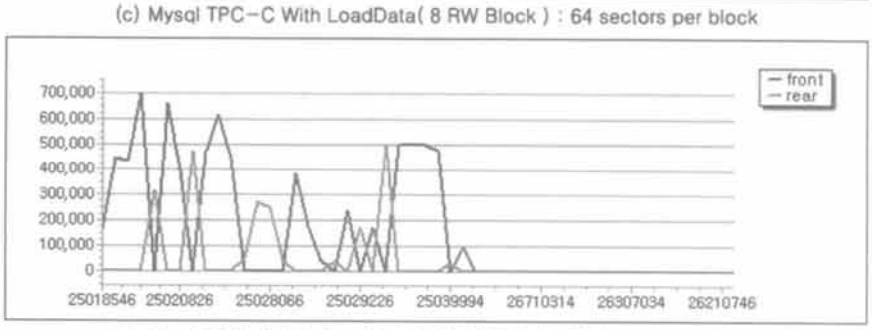
블록 내 최초 쓰기 연산이 블록의 앞부분(front) 혹은 뒷부분(rear)에서 시작하는지에 따라 성능차이가 생긴다. 그러한 이유는 최근 플래시 메모리는 고집적도에 고성능의 Large Block NAND 플래시 메모리가 생산되고 있으며, 이러한 플래시 메모리는 블록 내 쓰기 연산이 순차적으로 이루어져야 하는 추가적인 제약사항이 존재한다[5]. 위 모든 실험들은 이러한 제약사항을 반영한 결과로서 XAST는 쓰기 연산의 위치랑 밀접한 관계를 갖고 있다. 즉, 블록 당 섹터의 수가 64개라고 가정하면 최초 쓰인 위치로부터 순차적으로 쓰기 연산이 수행되어야 한다. 최초에 쓰기 연산이 수행된 섹터 이전에 비어있는 섹터가 존재하여도 쓰기 연산을 수행할 수 없고, 이러한 제약사항이 각 실험에서 섹터수가 변경됨에 따라 최초 쓰기 연산이 앞이나 뒤로 변경 될 수밖에 없다. (그림 16)은 각각의 섹터로 구성된 블록을 반으로 나누어 앞부분(front)과 뒷부분(rear)에 대해서 실험(c)와 실험(e)의 데이터 분포도를 나타낸다. x축은 물리 블록 번호를 나타내고, y축은 해당 블록에 수행된 쓰기 연산을 횟수를 수치화 하여 나타낸다. (그림 16)에서 실험(e) 그래프를 보면 256개 섹터 수에서는 앞부분(front)에 쓰기연산이 더 많이 분포되어 있는 것을 볼 수 있기 때문에 <표 4>의 실험(e)의 256개 섹터 수가 64개 섹터 수보다 성능이 좋다. 다시 말해 똑같은 실험데이터라 하여도 섹터의 수에 따라 블록 내 앞부분과 뒷부분의 분포도가 서로 달라지며 성능에 영향을 미치는 것을 알 수 있다.



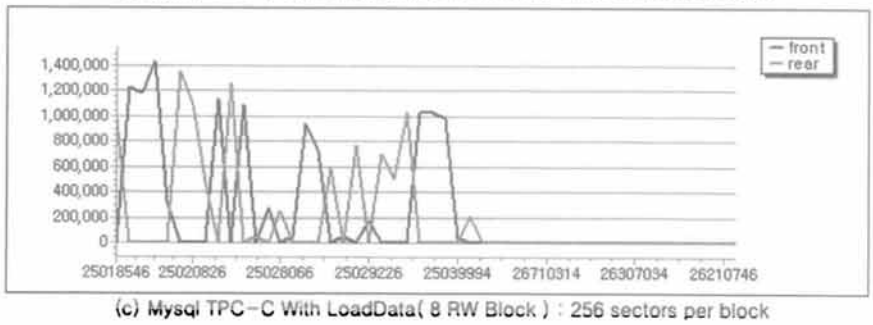
(c) Mysql TPC-C With LoadData(8 RW Block) : 32 sectors per block



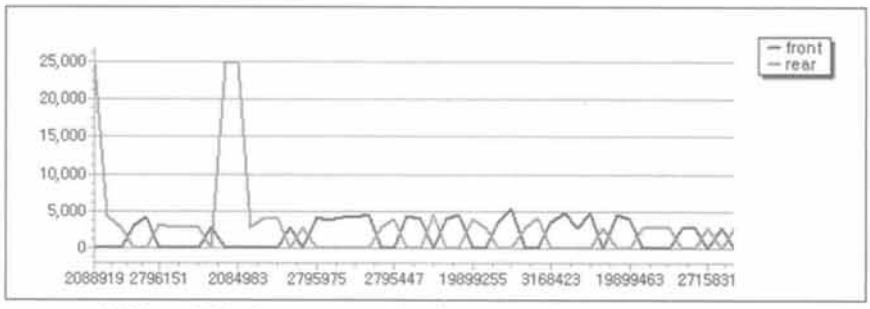
(c) Mysql TPC-C With LoadData(8 RW Block) : 64 sectors per block



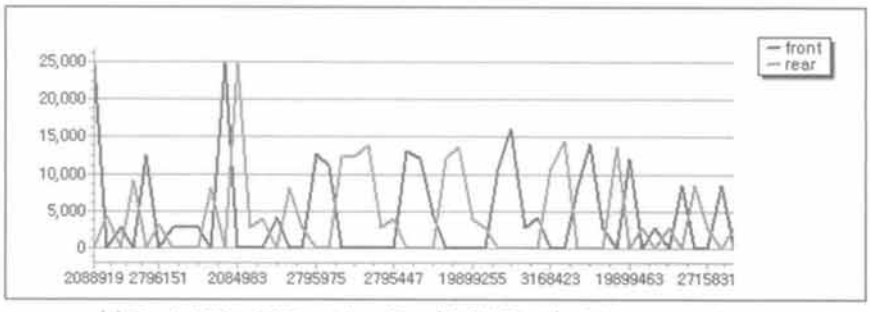
(c) Mysql TPC-C With LoadData(8 RW Block) : 128 sectors per block



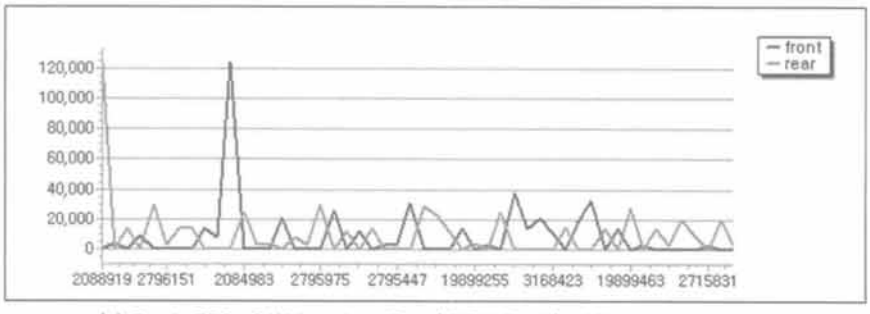
(c) Mysql TPC-C With LoadData(8 RW Block) : 256 sectors per block



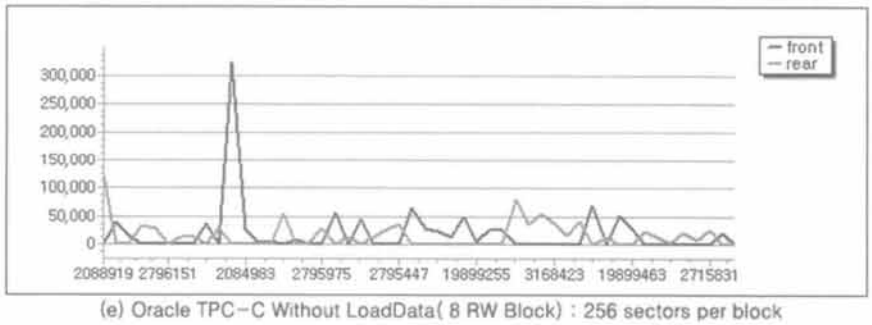
(e) Oracle TPC-C Without LoadData(8 RW Block) : 32 sectors per block



(e) Oracle TPC-C Without LoadData(8 RW Block) : 64 sectors per block



(e) Oracle TPC-C Without LoadData(8 RW Block) : 128 sectors per block



(e) Oracle TPC-C Without LoadData(8 RW Block) : 256 sectors per block

(그림 16) 실험 데이터 분포도(front/rear)

6. 결론 및 향후 연구

실험 결과를 통해 XAST는 FAST와 비교하여 임의쓰기에 대해 전체 합병과 삭제 연산이 감소하였다. 즉, 플래시 메모리 내 덮어쓰기로 발생하는 합병연산과 이를 이벤트처럼 인식하여 동작하는 오프셋 매핑테이블을 통해 데이터 블록 내 비어있는 섹터의 활용이 어떠한 영향을 미치는지 확인 할 수 있었으며, 기존 FTL과 비교하여 향상된 성능을 보였다.

본 실험에서는 오프셋 매핑 테이블의 크기를 단순히 로그블록의 개수 * 블록 내 섹터의 수를 계산한 값을 k라 할 때 k/2, k, k*2, k*4로 수행되었다. 만약 로그블록과 관계를 맺고 있는 모든 데이터블록에 대한 섹터 정보를 저장하고 운용하기 위해서는 오프셋 매핑 테이블의 크기는 로그 블록당 섹터의 수 * 데이터 블록 당 섹터의 수 * 로그블록의 개수 가 된다. 하지만 실험에서처럼 XAST는 작은 양의 SRAM을 요구하면서도 FAST와 비교하였을 때 상당 수 삭제 연산이 감소하였고, <표 4>를 통해 알 수 있었다. <표

4>에서 실험(c) 리눅스 기반의 데이터베이스에서는 블록 당 섹터의 수가 64개일 경우 오프셋 매핑 테이블의 크기가 k*2(1,024개)일 경우 최상의 성능을 발휘하였고, 블록 당 섹터의 수에 따라 성능 향상이 다른 걸 볼 수 있었다. 이러한 이유는 현재의 Large Block NAND 플래시 메모리가 갖는 특성 때문이다.

XAST가 더욱 발전하기 위해서는 오프셋 매핑 테이블의 최적화된 크기를 찾고, 로그블록과 데이터블록과의 관계 개선과 함께 Large Block NAND 플래시 제약사항을 극복하여야 한다. 또한, 임의쓰기와 순차쓰기가 동시 다발적으로 발생할 경우에 대해 필터링 기법을 적용하는 것을 제안한다.

참 고 문 헌

[1] D. Roselli, J. R. Lorch, T. E. Anderson. "A comparison of file system workloads," *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000.

[2] J. S. Kim, J. M. Kim, S. H. Noh, S. L. Min and Y. K. Cho, "A space-efficient flash translation layer for CompactFlash systems," I*EEE Transactions on Consumer Electronics*, Vol.48, No.2, pp.366-375, 2002.

[3] T.S. Chung, S. Park, M. J. Jung and B. S. Kim, "STAFF: State Transition Applied Fast Flash Translation Layer," *Lecture Notes in Computer Science, Organic and Pervasive Computing - ARCS 2004*, Vol.2981, pp.67-80, 2004.

[4] 정태선, 박형석, "플래시 메모리를 위한 효율적인 사상 알고리즘," *정보과학회논문지*, Vol.32, No.9, 2005.

[5] J. U. Kang, H. S. Jo, J. S. Kim and J. W. Lee, "A superblock-based flash translation layer for NAND flash memory," *In Proceedings of the 6th ACM & IEEE International conference on Embedded software*. 2006.

[6] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. W. Park and H. J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems*, Vol.6, No.3, July, 2007.

[7] S. J. Kwon and T. S. Chung, "An efficient and advanced space-management technique for flash memory using reallocation blocks," *IEEE Transactions on Consumer Electronics*, Vol.54, No.2, pp.631-638, May, 2008.

[8] C. I. Park, W. M. Cheon, J. U. Kang, K. H. Roh, W. H. Cho and J. S. Kim, "A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications," A*CM Transactions on Embedded Computing Systems*, Vol.7, No.4, July, 2008.

[9] S. J. Lee, D. K. Shin, Y. J. Kim and J. H. Kim, "LAST: locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Operating Systems Review*, Vol.42, No.6, October, 2008.

[10] H. J. Kim and S. J. Ahn, "BPLRU: a buffer management scheme for improving random writes in flash storage," *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[11] A. Gupta, Y. J. Kim and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, March, 2009.

[12] H. J. Cho, D. K. Shin and Y. I. Eom, "KAST: K-associative sector translation for NAND flash memory in real-time systems," *Design, Automation & Test in Europe Conference & Exhibition*, pp.507-512, April, 2009.

[13] T. S. Chung, D. J. Park, S. W. Park, D. H. Lee, S. W. Lee and H. J. Song, "A survey of Flash Translation Layer," *Journal of Systems Architecture*, Vol.55, No.5-6, pp.332-343, May, 2009.

[14] 윤태현, 김광수, 황선영, "섹터 매핑 기법을 적용한 효율적인 FTL 알고리듬 설계," *한국통신학회논문지*, Vol.34, No.12, pp.1418-1425, 2009.

[15] 조혜원, 한용구, 이영구, "플래시메모리 DBMS를 위한 블록의 비고정적 로그 영역 관리 기법," *정보과학회논문지*, Vol.37, No.5, 2010.

[16] M. Russinovich, "DiskMon for Windows v.2.01," http://www.microsoft.com/technet/sysinternals/utilities/diskmon.mspx, 2006.

[17] Samsung Electronic, "128M x 8 Bit NAND Flash Memory," http://www.datasheet4u.com, 2006.

[18] S. W. Lee, B. K. Moon, C. I. Park, "Advances in Flash Memory SSD Technology for Enterprise Database Applications," *Proceedings of the 35th SIGMOD international conference on Management of data*, 2009.

이 준 혁

e-mail : mediassu@naver.com

2007년 숭실대학교 미디어학부(학사)

2007년~2008년 (주)현대성우종합건설 전산팀 근무

2009년~현 재 연세대학교 컴퓨터공학과 석사과정

관심분야 : 데이터베이스시스템, 플래쉬메모리, SSD, 소프트웨어 공학

노 홍 찬

e-mail : fallsmal@cs.yonsei.ac.kr

2006년 연세대학교 컴퓨터과학과(학사)

2008년 연세대학교 컴퓨터과학과 (공학석사)

2008년~현 재 연세대학교 컴퓨터과학과 박사과정

관심분야 : 데이터베이스시스템, 플래쉬메모리, SSD

박 상 현

e-mail : sanghyun@cs.yonsei.ac.kr

1989년 서울대학교 컴퓨터공학과(학사)

1991년 서울대학교 컴퓨터공학과 (공학석사)

2001년 UCLA 대학원 컴퓨터과학과 (공학박사)

1991년~1996년 대우통신 연구원

2001년~2002년 IBM.J.Watson Research Center Post-Doctoral Fellow

2002년~2003년 포항공과대학교 컴퓨터공학과 조교수

2003년~2006년 연세대학교 컴퓨터과학과 조교수

2006년~2011년 연세대학교 컴퓨터과학과 부교수

2011년~현 재 연세대학교 컴퓨터과학과 교수

관심분야 : 데이터베이스, 데이터마이닝, 바이오인포매틱스, 적응적 저장장치 시스템, 플래쉬메모리 인덱스, SSD