Posted by : Jaeeun Kim at Jun 12, 2019

Disk/File Manager

본 시리즈는 CUBRID의 Disk Manager와 File Manager의 내부구조와 동작과정을 소스코드레벨에서 분석하여 정리한 내용을 담는다. Disk/File Manager는 큐브리드가 Heap, Index등의 데이터를 OS의 파일에 담고 이들을 위해 공간을 할당, 조정한다. 이 글은 시리즈의 첫번째 포스팅으로 전체적인 아키텍처에 관한 개요와 용어들을 설명한다. 이어지는 포스팅에서는 개념, 아키텍쳐 뿐만 아니라 구조체와 함수단위에서 각 컴포넌트가 어떻게 동작하는지 살펴볼 것이다.

> 모든 내용은 CUBRID github 저장소의 버전은 10.2.0, develop branch: 7094ba33f61c을 기준으로 한다.

> 설명 중 함수명이나 변수명은 이텔릭체로 *var1, funtion()* 등으로 표기한다. 특히 함수의 경우 *function()*과 같이 인자없이 괄호를 추가하여 표기한다.

용어정리

- **볼륨 (Volume)**: 큐브리드에서 사용하는 OS가 제공하는 파일(open() 시스템콜을 통해 생성하는)을 말한다. 볼륨은 데이터 볼륨, 로그 볼륨, 백업 볼륨 등 여러 종류가 있지만 앞으로 언급하는 볼륨은 모두 Index와 Heap등이 담기는 데이터 볼륨(혹은 디스크 볼륨)을 가리키는 것으로 한다.
- **페이지 (Page)**: 페이지는 고정된 크기의 연속적인 데이터 블록(block)으로 큐브리드가 스토리지를 관리할 때 사용하는 가장 작은 단위이다. 큐브리드에는 로그페이지와 데이터페이지 두종류가 있지만 여기서는 데이터페이지만을 가리키는 것으로 한다. 이 페이지는 메모리로 로드되면, 버퍼매니저(Buffer Manager)에서 사용되는 Page Buffer와 매핑된다.
- **섹터 (Sector)**: 큐브리드가 스토리지를 다룰 때 사용하는 또 다른 단위로, 64개의 페이지들의 묶음을 이야기한다. 페이지 단위로 관리하는 것은 자원소모가 심하므로 묶음인 섹터로 일차적인 관리를 한다.
- **파일 (File)**: 큐브리드에서 이야기하는 파일이란, OS가 제공하는 파일이 아닌 특정 목적을 위해 예약된 섹터들의 묶음을 말한다. 파일은 섹터단위로 볼륨의 공간을 예약하고, 필요에 따라 섹터내의 페이지를 할당하여 사용한다. 각각의 파일은 하나의 테이블, 하나의 인덱스, 나중에 이야기할 파일 트래커(File Tracker)등의 정보를 담는다.
- **섹터 예약 (Reservation)**: 볼륨의 섹터를 사용하기로 하는 행위를 말한다. 반대로 사용을 중지하고 반납하는 행위를 섹터 예약 해제 (Unreservation)이라고 한다.
- **페이지 할당 (Allocation)**: 파일에서 예약한 섹터중 한 페이지를 사용하기로 하는 행위를 말한다. 반대로 사용을 중지하고 반납하는 행위를 페이지 할당해제(Deallocation)라고 한다.

> 앞으로 파일은 상기에 서술된 큐브리드의 파일을 이야기하고, OS의 파일을 이야기할 때는 OS파일 혹은 볼륨으로 표현한다.

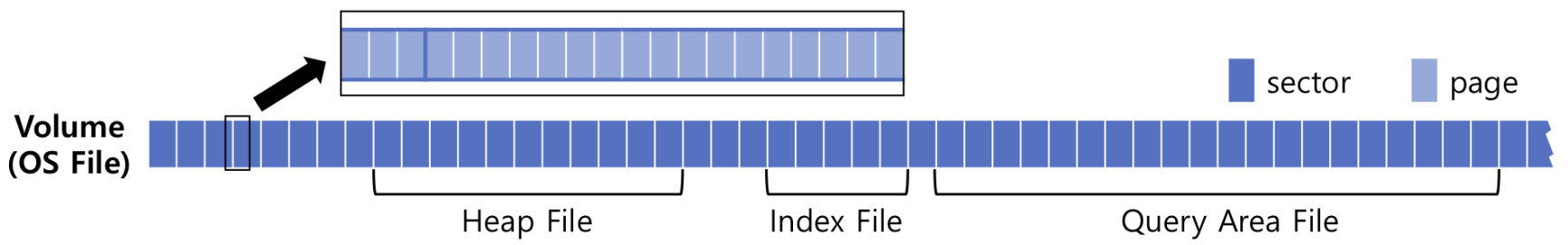


Figure 1: Volume Overview

디스크 매니저와 파일매니저

디스크 매니저: 볼륨공간 전체를 관리하며 섹터들의 예약여부를 트래킹한다. 섹터들의 예약관리가 주역할이며, 모든 섹터가 다 예약돼었을 경우 볼륨의 크기(OS파일의 크기)를 늘리거나 볼륨의 수를 늘림으로서 추가적인 섹터를 확보한다. 디스크매니저 관련함수들은 *disk_** prefix로 시작한다.

파일매니저: 큐브리드의 내부파일들을 관리하며 디스크매니저로부터 섹터들을 할당받고 파일 내에서의 페이지할당여부를 트래킹한다. 페이지들의 할당 관리가 주 역할이며, 모든 페이지를 할당하여 추가적인 공간이 필요할 경우 디스크 매니저에게 추가적인 섹터를 요청한다. 파일매니저 관련 함수들은 *file_** prefix로 시작한다.

> 앞으로 설명할 볼륨관련 데이터나 함수는 디스크 매니저, 파일관련 데이터나 함수는 파일매니저라고 보면 된다.

볼륨 (Volume)

데이터볼륨은 영구(Permanent)볼륨과 임시(Temporary)볼륨으로 나뉜다.

- **영구볼륨**: 테이블, 인덱스, 시스템데이터 등이 담기는 볼륨으로 한번 만들어지면 영구히 존재하는 볼륨을 말한다.
- **임시볼륨**: 정렬(sorting)과정이나 쿼리결과를 가져오는 과정에서의 임시데이터들을 담는 볼륨을 말하며, 데이터베이스 종료와 재시작시 모두 삭제된다.

그렇다고해서 영구볼륨의 데이터는 항상 보존되어야 하는 데이터만(즉, 리커버리의 대상이되는)이 담기는 것은 아니고 addvoldb로 임시목적(purpose)의 영구볼륨을 만들경우 임시데이터가 담길 수도 있다. 볼륨에 대한 개념적인 내용은 큐브리드 메뉴얼을 참고하자.

볼륨은 언제만들어질까?

이해를 돕기위하여 볼륨의 생성이 언제 일어나는지를 알아보자. 영구볼륨이 생성되는 경우는 다음과 같다.

1. 데이터베이스를 createdb명령을 통하여 생성할 때
2. 사용중인 영구볼륨들이 모두 가득차서 추가적인 공간이 필요할 때
3. addvoldb를 통하여 사용자가 직접 볼륨을 추가할 때

데이터가 지속적(durable)으로 보관되기 위해서는 영구볼륨이 필요하다. 지속성은 데이터베이스의 기본 속성이므로 데이터베이스가 처음 생성될 때 기본적으로 영구볼륨이 생성된다. 생성된 영구볼륨을 사용하는 과정에서 데이터가 가득차게 되면 새로운 볼륨이 추가로 생성되는데, 이 때는 시스템파라미터에 있는 정보를 바탕으로 최소한의 크기로 볼륨이 생성되고, 데이터가 채워짐에 따라서 볼륨의 크기가 늘어난다. 그러다가 다시 볼륨의 최대크기까지 데이터가 입력되면 새로운 볼륨이 생성되어 이를 처리한다.

> 영구볼륨의 제거나 축소는 없고, deletedb를 통해서 전체 데이터베이스를 제거할때만 볼륨들이 모두함께 제거되는 것으로 보인다.

그렇다면 임시볼륨은?

쿼리중간결과등의 임시데이터는 임시볼륨에만 담길 수 있는 것은 아니다. 임시목적의 영구볼륨이 존재할 경우에는 기본적으로 이 영구볼륨에 담기게 되며 공간이 부족해질 경우에 임시볼륨이 만들어져 데이터를 저장한다. 이렇게 생성된 임시볼륨은 이후에 데이터베이스가 종료되거나 재시작될 때 모두 제거된다.

> 만약 이미 임시 볼륨이 있다면?
>
> 임시데이터를 위한 섹터를 할당하려할 때 이미 이전에 만들어진 임시볼륨이 존재한다고 해도, 먼저 임시목적의 영구볼륨에서 섹터를 할당시도한다.

파일 (File)

앞서 이야기했듯이 큐브리드에서 말하는 파일은 OS에서 제공하는(open등의 시스템콜로 생성하는) OS file이 아니라, 큐브리드만의 독자적인 유닛으로 볼륨내의에서 **하나의 목적으로 할당된 섹터들의 논리적인 집합**이다. 파일을 생성하면 볼륨의 섹터들을 할당받고 파일내에서는 이를 가장 작은 단위인 페이지로 나누어서 관리하며, 할당받았던 섹터의 페이지를 모두 사용하면 추가적인 섹터들을 추가적으로 할당 받는다. 파일도 볼륨과 같이 영구파일과 임시파일 두가지로 분류할 수 있다.

- **영구파일**: 각각의 파일은 인덱스, 힙데이터, 파일트래커 등의 특정한 목적을 가지고 있으며, 변경이 일어나면 이는 로그로 기록되어 리커버리의 대상이 된다. 영속적으로 보관할 데이터이므로 업데이트 비용이 비싸며 관리 오버헤드가 있다.
- **임시파일**: 쿼리나 정렬의 중간결과들이 일시적으로 쓰여지는 파일로 기본적으로 임시파일을 사용하는 트랜잭션이 종료되면 제거된다. 혹은, 필요에 따라 트랜잭션종속에서 벗어나 쿼리매니저에서 관리되기도 한다. 임시파일은 사용하는 순간에만 유효하면 되므로 영구파일에 비해 연산자체도 단순하고 관리 오버헤드도 적다.

파일은 위의 두가지 분류 뿐만 아니라 파일의 목적에 따라 heap, btree, catalog등의 파일타입으로도 나눌 수 있다.

Numerable 속성

파일은 Numerable 속성을 지닐 수 있다. 기본적으로 파일에 할당되는 페이지는 순서가 없다. 물리적으로 연속적인 페이지만을 할당받는 것은 아니며 여러 섹터에 흩어져 있는 페이지들을 할당받는다. 심지어 할당받은 각 페이지가 속한 섹터들도 연속적이지 않을 수 있으며 여러볼륨에 흩어져 있을 수도 있다. 이러한 페이지들에 논리적인 순서를 부여하는 속성을 numerable이라 한다. 이 속성을 지니면 파일에 할당된 페이지(정확히는 유저페이지만)들을 할당된 순서대로 인덱스를 통하여 접근할 수 있다. 이는 extensible hash나 external sorting의 경우에 유용하게 사용될 수 있다.

> 이 중 extensible hash는 현재 deprecated되어 있고 호환성을 위해서만 남아있는 것으로 보인다.

이번 글에서는 디스크매니저와 파일매니저에서 쓰이는 개념들에 대하여 정리하였다. 앞으로의 글에서는 이를 바탕으로 어떤 방식으로 파일과 볼륨이 관리되는지 자세히 알아본다. 이후의 내용은 다음과 같다.

- 볼륨은 어떻게 관리될까?
- 섹터 예약은 어떻게 이루어질까?
- 섹터 예약시 볼륨에 섹터가 부족하면?
- 큐브리드 파일은 어떻게 관리될까?
- 페이지 할당은 어떻게 이루어질까?
- 파일의 생성과 제거

Posted by : Jaeeun Kim at Jun 25, 2019

볼륨은 어떻게 관리될까?

- 볼륨헤더(Volume Header)와 섹터테이블(Sector Table) -

앞선 포스터 (Overview)에서 볼륨 매니저가 섹터의 예약(reservation)을 관리한다고 이야기하였다. 이번 글에서는 볼륨내의 섹터들이 어떻게 관리되는지에 대한 구체적인 이야기와 이를 위해 볼륨이 어떻게 구성되어 있는지를 다룬다. 여기서 다루어지는 볼륨의 구조는 그대로 non-volatile memory (SSD, HDD 등)에 쓰여진다.

볼륨 구조

디스크매니저의 가장 큰 역할은 파일생성과 확장을 위해 섹터들을 제공해주는 것이다. 이를 위해 각 볼륨은 파일들에게 할당해줄 섹터들과 이를 관리하기 위한 메타(meta)데이터로 이루어져있다. 메타데이터들이 저장된 페이지를 볼륨의 **시스템페이지(System Page)**라고 하며, 볼륨에 대한 정보와 각 섹터들의 예약여부를 담고 있다. 시스템페이지는 다음과 같이 두가지로 나뉜다.

- **볼륨 헤더 페이지 (Volume Header Page, 이하 헤더페이지)**: 페이지 크기, 볼륨내 섹터의 전체/최대 섹터, 볼륨 이름 등, 볼륨에 대한 정보를 지니고 있는 페이지
- **섹터 테이블 페이지 (Sector Table Page, 이하 STAB페이지)**: 볼륨내의 각 섹터의 예약여부를 비트맵으로 들고 있는 페이지

이러한 시스템페이지들은 볼륨이 생성될 때 미리 볼륨 내의 정해진 공간에 쓰여지고, 이 페이지들이 포함된 섹터를 제외한 나머지 섹터들이 파일매니저로부터의 섹터예약요청을 처리하기위해 사용된다. 볼륨헤더는 볼륨의 첫번째 한 개의 페이지에 할당되고, STAB 페이지는 헤더페이지의 바로 다음 페이지부터 볼륨의 크기를 모두 커버할 수 있는 만큼의 양이 연속적으로 할당받는다 (*disk_stab_init()*). 이를 도식화 하면 다음과 같다.

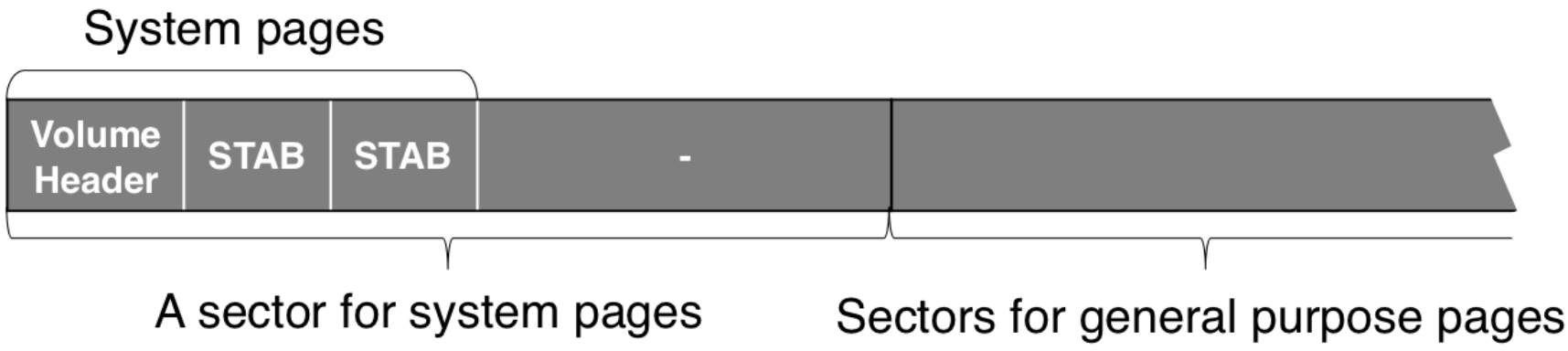


Figure 1: Volume Format

첫 섹터가 시스템페이지들을 위해 할당되어 있는 모습을 볼 수 있다. 볼륨의 시스템페이지들의 수가 한 섹터를 못 채울 경우 그림처럼 시스템페이지들을 위해 할당된 섹터 내의 페이지들이 일부 사용되지 않을 수 있고, 볼륨에 크기가 커지면 이에 따라 시스템페이지들을 위한 섹터가 둘 이상 할당될 수도 있다.

볼륨 헤더 (Volume Header)

볼륨헤더(*DISK_VOLUME_HEADER*)는 볼륨의 첫번째 페이지에 쓰여지며, 기본적으로 볼륨에 대한 정보들이 고정사이즈로 들어가고 나머지공간에는 가변길이 변수들이 들어간다. 볼륨 헤더가 담고 있는 정보는 크게 5가지 정도로 분류할 수 있다.

- 볼륨 정보: 볼륨 자체에 대한 정보로 볼륨전체에 공통적으로 적용되는 정보이다. 볼륨의 타입, 캐릭터셋, 생성시간, 섹터당 페이지수, 페이지의 크기 등이 저장된다.
- 섹터 정보: 볼륨의 현재 섹터의 정보이다. 볼륨내에 몇개의 섹터가 있는지, 얼마나 확장될 수 있는지 등이 저장된다.
- 시스템페이지 정보: 앞서 이야기한 시스템페이지에 대한 정보들이 저장된다.
- 체크포인트 정보: 마지막으로 체크포인트가 성공시 체크포인트의 시작 지점의 로그 레코드 LSA 정보가 저장된다. 이는 리커버리과정에서 사용된다.
- 가변길이 변수: 볼륨헤더페이지내에서 볼륨헤더의 모든 고정변수들을 제외한나머지 공간은 가변길이 변수들을 위한 공간이다. 볼륨의 full path나 사용자 정의 comment등이 저장된다.
- 기타: reserved 등 동작과 무관한 특수목적 변수들이 저장된다.

구체적으로 볼륨 헤더 구조체(*DISK_VOLUME_HEADER*)가 담고 있는 정보(변수)들은 다음과 같다.

분류	변수 타입	변수명	설명
볼륨	INT8	db_charset	데이터베이스의 캐릭터셋
	INT16	volid	해당 볼륨의 볼륨 식별자
	DB_VOLTYPE	type	볼륨의 타입, 볼륨이 어떻게 관리될지를 결정 Permanent: 영구적으로 볼륨유지 Temporary: 서버 종료/재시작시 제거. 임시데이터를 저장하는데 기존 볼륨의 공간이 부족할 경우 생성된다.
	DB_VOLPURPOSE	purpose	볼륨의 이용목적, 볼륨을 어떻게 사용할지를 결정 Permanent: 영구적인 데이터를 저장할 것. Temporary: 임시적인 데이터를 저장할 것. 임시데이터를 저장할 때에 임시타입의 볼륨을 만들기전에 임시목적의 영구타입볼륨이 있을 경우 먼저 사용한다.
	INT64	db_creation	데이터베이스 생성시간
	INT16	next_volid	여러 볼륨이 있을 경우 그들을 연결하는 포인터, 다음 볼륨의 식별자를 담음
	DKNPAGES	sect_npgs	한 섹터당 페이지 수
	INT16	iopagesize	한 페이지의 크기
	HFID	boot_hfid	볼륨 부팅과 멀티 볼륨관련된 정보를 담고있는 힙(Heap)파일의 식별자
섹터	DKNPAGES	nsect_total	볼륨의 현재 총 섹터 수, 볼륨파일의 크기를 결정
	DKNPAGES	nsect_max	볼륨이 확장될 수 있는 최대 크기의 섹터 수
	SECTID	hint_allocsect	섹터예약시 섹터테이블의 어디부터 탐색할지 캐싱해둔 값
시스템페이지	DKNPAGES	stab_npages	섹터테이블이 차지하는 페이지 수
	PAGEID	stab_first_page	섹터테이블의 시작페이지
	PAGEID	sys_lastpage	마지막 시스템 페이지 (현재 stab_first_page+stab_npages -1)
체크포인트	LOG_LSA	chkpt_lsa	체크포인트 시작점의 LSA, 리커버리분석의 시작점 (ARIES의 master record)
가변길이변수	char [1]	var_fields	가변길이 변수들의 시작점, var_fileds + offset_to_* 가 각 가변변수의 위치
	INT16	offset_to_vol_fullname	볼륨의 절대경로 이름의 offset
	INT16	offset_to_next_vol_fullname	next_volid 볼륨의 절대경로 이름의 offset
	INT16	offset_to_vol_remarks	볼륨에 대한 코멘트의 offset 코멘트는 볼륨포맷(*disk_format()*)시에 적히는 것으로 유저가 addvoldb를 실행하면서 적는 코멘트나 볼륨의 공간이 가득차 자동으로 새로운 볼륨을 만들어질 경우 적히는 코멘트 ("Automatic Volume Extension") 등이 들어간다.

기타	INT32	reserved0/1/2/3	미래 확장성을 위한 예약변수들
	INT8/32	dummy1/2	alignment를 위한 더미변수들
	char []	magic	볼륨파일의 매직넘버

> 각 변수들에 대한 설명을 달아두었긴 했지만 명확한 이해를 위해서는 각 변수들의 값들이 언제 설정되고, 어떻게 사용되는지 등을 알아야한다. 이에 대한 자세한 내용은 각 변수들이 이용되는 부분을 설명할 때에 다시 살펴보도록 한다.

섹터 테이블 (Sector Table)

섹터테이블(STAB)은 볼륨내 모든 섹터들의 사용여부(예약여부)를 저장하고 있는 비트맵이다. 섹터테이블페이지의 하나의 비트는 하나의 섹터의 예약 여부를 나타낸다. 섹터테이블은 볼륨헤더페이지의 바로 다음페이지(볼륨의 두번째 페이지, *stab_first_page*)부터 시작하여 볼륨의 최대크기(*nsect_max*)를 커버할 수 있는 만큼의 페이지(*stab_npages*)를 사용한다.

> 현재 코드는 볼륨헤더페이지의 다음페이지가 *stab_first_page*, STAB의 마지막 페이지가 *sys_lastpage* 와 같은 값을 가지고 있지만 데이터의 구조상 또 다른 시스템페이지가 추가될 수 있을 것으로 보인다.

섹터예약에 관한 연산을 수행할 때, 각 비트들을 하나씩 순회하며 연산을 수행할 수도 있지만 큐브리드는 비트들을 ***DISK_STAB_UNIT* (이하 unit, 유닛)**이라는 단위로 묶어 관리, 연산하고 불가피할 때에만 비트를 순회한다. 비트연산을 할 때에 CPU 아키텍쳐등을 고려하여 효율적인 방법으로 처리 할 수 있도록 이러한 처리단위를 제공하는 것으로 보인다. 정리하자면 섹터테이블의 비트맵은 여러페이지로 구성되며 각 페이지는 다시 유닛으로 나뉘고, 유닛의 비트들은 각각의 하나의 섹터의 예역여부를 나타낸다. 섹터테이블을 읽거나 조작하는 등의 연산은 모두 이 유닛을 기반으로 이루어진다.

> 현재 유닛은 다음과 같이 UINT64형이다. CPU아키텍처나 디자인에 맞춰 이 값을 변경시키면 STAB의 관리 단위를 변경 시킬 수 있다. 주석 또한 이 값의 변경을 통해 유닛단위를 쉽게 변경할 수 있을 것이라 이야기하고 있다.
>
> ```
> typedef UINT64 DISK_STAB_UNIT;
> // ... If we ever want to change the type of unit, this can be modified and should be handled automatically, ...
> ```
>
> 하지만, 유닛을 단위로 하는 연산들이 *bit64_count_zeros()* 등으로 64bit에 맞게 하드코딩되어 있기에 변경이 그렇게 단순해 보이지는 않는다. 각 바이트단위로 비트연산 함수들은 base/bit.c에 모두 구현되어 있기 때문에 유닛 사이즈에 맞춰 각 비트연산을 호출하게끔 수정한다면 코드의 유연성을 확보할 수 있을 것으로 보인다.

만약 sector_id가 32100인 섹터에 대한 예약여부를 확인하려할 때, STAB에서 해당 비트의 위치는 어떻게 구할 수 있을까? 이는 마치 초에서 (시,분,초)를 구하듯 (page_id, offset_to_unit, offset_to_bit) 으로 다음과 같이 계산된다.

```
page_id: (볼륨헤더의 stab_first_page) + sector_id / (페이지의 비트 수)
offset_to_unit: sector_id % (페이지의 비트 수) / (페이지내 유닛의 수)
offset_to_bit: sector_id % (페이지의 비트 수) % (페이지내 유닛의 수)
```

만약 1KB페이지, 64bit unit이라면 sector_id 32100인 (3, 117, 36)이 된다.

안타깝게도 페이지의 크기가 2^n형태가 아니기 때문에 OS의 페이지테이블이나 CPU 캐시처럼 단순 비트 쉬프트연산으로 유닛과 오프셋등을 구할 수 없다. 때문에 비싼 /, % 연산이 사용된다.

> IO 페이지의 크기는 4KB, 16KB 등 2^n형태이더라도 모든 페이지가 공통적으로 페이지타입, LOG_LSA 등의 공간을 이미 예약해두었기 때문에 실제 사용가능한 크기는 이 영역을 제외한 크기이다.

섹터 테이블의 연산

섹터의 예약정보를 조회하거나 예약하려면 섹터테이블의 비트맵을 조작해야한다. 이러한 연산들은 앞서말한 유닛단위를 기반으로 이루어지며, 하나의 섹터비트나 유닛을 참조할 일 보다는 여러 유닛들을 참조하는 경우가 대부분이기 때문에 **커서(Cursor, *DISK_STAB_CURSOR*)**와 이터레이션 인터페이스(*disk_stab_iterate_units()*)를 제공한다. 커서는 볼륨내 한 섹터의 STAB에서의 위치(page_id, offset_to_unit, offset_to_bit)를 가리킨다. 또, 커서가 가리키는 유닛에 대한 연산을 위해 커서가 가리키고 있는 유닛의 포인터(page, unit)를 들고 있다.

```
typedef struct disk_stab_cursor DISK_STAB_CURSOR;
struct disk_stab_cursor
{
    const DISK_VOLUME_HEADER *volheader;    /* Volume header */

    PAGEID pageid;      /* Current page ID */
    int offset_to_unit;     /* Offset to current unit in page. */
    int offset_to_bit;      /* Offset to current bit in unit. */

    SECTID sectid;      /* Sector ID */

                // 위의 변수들은 모두 현재 커서가 가리키는 섹터에 대한 정보와 STAB내에서 섹터의 위치
                // 아래의 변수들은 위의 변수들이 가리키는 STAB내의 유닛을 참조하기 위한 포인터

    PAGE_PTR page;      /* Fixed table page. */
    DISK_STAB_UNIT *unit;       /* Unit pointer in current page. */
};
```

이터레이션 함수인 *disk_stab_iterate_units()* 의 선언부는 다음과 같다. (설명에 필요하지 않은 인자들은 제외하였다.)

```
static int disk_stab_iterate_units (..., DISK_STAB_CURSOR * start, DISK_STAB_CURSOR * end,
DISK_STAB_UNIT_FUNC f_unit, void *f_unit_args)
```

앞서 이야기한 커서 자료형의 *start, end*와 이터레이션하면서 유닛에 적용할 함수(*DISK_STAB_UNIT_FUNC*)와 함수의 인자를 매개변수로 받는 것을 볼 수있다. 이 함수는 [*start, end*) 범위의 유닛을 순회하면서 각 유닛마다 *DISK_STAB_UNIT_FUNC*함수를 적용시킨다. 여타 함수형 프로그래밍언어에 있는 map()함수를 생각하면 이해가 쉽다. *start, end* 커서는 *disk_stab_cursor_set_at_*()* 류의 함수를 통해 STAB의 시작이나 끝, 특정 sector ID로 설정된다. *DISK_STAB_UNIT_FUNC* 는 함수포인터로 다음과 같다.

```
typedef int (*DISK_STAB_UNIT_FUNC) (..., DISK_STAB_CURSOR * cursor, bool * stop, void *args);
```

*disk_stab_iterate_units()*에서 이터레이션되어 만나는 각 유닛에 대한 커서를 인자로 받아 사용자가 정의한 작업을 진행한다. 이 때 *stop*에 true를 넣고 함수를 종료하면, *disk_stab_iterate_units()* 의 이터레이션이 종료된다. 예를 들어 30개의 섹터를 예약하려 할 때, 이번 유닛에서 30개의 섹터 예약을 모두 완료했다면 더 이상의 작업을 중지하는 종료조건으로 활용할 수 있다. 이러한 유닛이터레이션을 통한 연산에는 섹터들 예약, 섹터들 예약 해제, 가용섹터들의 갯수확인등이 있다. 좀 더 확실한 이해를 위해 가용섹터들의 갯수확인에 사용되는 *DISK_STAB_UNIT_FUNC* 는 *disk_stab_count_free()* 와 이에 대한 호출부를 살펴보자.

```
// free sector의 갯수를 구하는 함수 정의
static int disk_stab_count_free (THREAD_ENTRY * thread_p, DISK_STAB_CURSOR * cursor, bool * stop, void
*args)
{
    DKNSECTS *nfreep = (DKNSECTS *) args;

    /* add zero bit count to free sectors total count */
    *nfreep += bit64_count_zeros (*cursor->unit);
    return NO_ERROR;
}

// 함수 호출부
int disk_rv_volhead_extend_redo (THREAD_ENTRY * thread_p, LOG_RCV * rcv)
{
        ...
        disk_stab_cursor_set_at_sectid (volheader, volheader->nsect_total - nsect_extend,
&start_cursor);
        disk_stab_cursor_set_at_end (volheader, &end_cursor);
                error_code = disk_stab_iterate_units (thread_p, volheader, PGBUF_LATCH_READ,
&start_cursor, &end_cursor, disk_stab_count_free, &nfree);
        ...
    disk_cache_update_vol_free (volheader->volid, nfree);
        ...
}
```

호출부의 예는 recovery의 redo phase에 사용되는 함수중 하나인 *disk_rv_volhead_extend_redo()* 로, 실제로 확장된 볼륨내의 free setor의 갯수를 디스크 캐시에 업데이트하기 위한 코드이다. 확장하기 전의 위치(*volheader->nsect_total - nsect_extend*)에 *start*커서를 두고, stab의 끝에 *end*커서를를 두고 *disk_stab_iterate_units()*함수를 호출하여 [*start, end*)를 순회하며 모든 유닛들에서 0인 비트들의 갯수를 구하는 것을 볼 수 있다.

> 이러한 이터레이션 방식은 파일매니저와 디스크매니저의 여러 곳에서 사용된다. 대표적으로 나중에 살펴볼 파일 매니저의 파일테이블과 유저테이블등에서도 이러한 패턴으로 데이터를 접근, 조작한다.

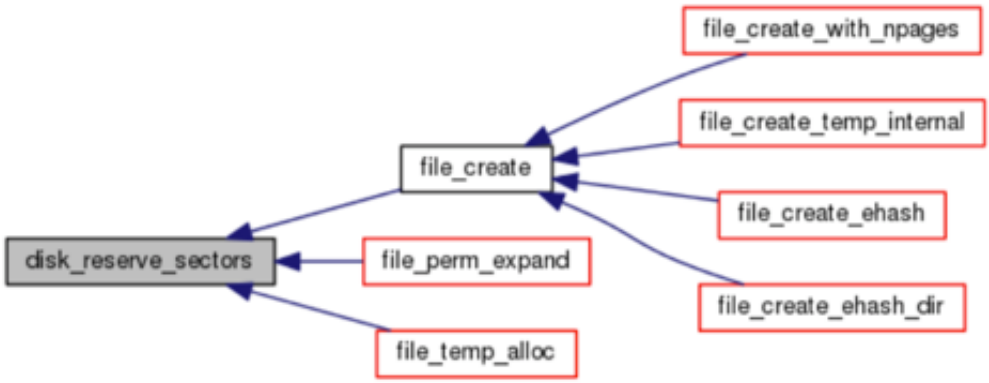
이어서 다룰 디스크매니저 내용은 다음과 같다.

1. 섹터 예약 및 예약해제
2. 볼륨 확장

섹터 예약은 어떻게 이루어질까?

- 2 단계 섹터 예약 (2-Step Sector Reservation) -

볼륨 내의 파일이 이전에 예약한 섹터를 모두 사용하여 추가적인 페이지를 할당할 수 없다면 디스크 매니저에게 추가적인 섹터를 요청한다. 디스크 매니저는 이러한 섹터 예약요청을 어떻게 처리할까? 이전 글 의 내용으로 예상할 수 있듯이, *disk_stab_iterate_units()*함수를 통해 볼륨 파일의 섹터테이블을 순회하며 가용한 섹터가 있는지 확인하고 해당 비트를 set 해줌으로써 섹터를 예약했음을 표시할 수 있을 것이다. 기본적으로 섹터 예약이란 이처럼 디스크에 있는 섹터테이블을 변경시키는 것은 맞지만, 큐브리드는 이에 앞서 한 단계 과정을 더 수행한다. 이번 글에서는 섹터 예약이 어떻게 이루어지는 자세히 알아본다.

> 언제 섹터 예약 요청이 발생할까?
>
> 섹터 예약함수(*disk_reserve_sectors()*)를 호출하는 경우를 살펴보면 아래의 그래프에서 볼수 있듯이 새로운 파일이 생성될 때, 혹은 파일에서의 페이지할당과정에서 이전에 예약해둔 섹터를 모두 사용하여 파일을 확장해야 할 때이다.
>
> 
>

Sector Reservation Overview

큐브리드의 섹터 예약은 다음과 같은 **두 단계**로 이루어진다.

1. **디스크 캐시 (Disk Cache)를 통해 섹터를 사전 예약**
    1. 현재 볼륨들로 섹터예약이 가능한지 확인, 불가능할 경우 공간 확보 (볼륨 확장, 추가)
    2. 섹터를 예약할 볼륨들 선택, 각 볼륨별로 몇개의 섹터를 예약할지 결정
2. **사전예약에서 결정된 사항대로 섹터테이블을 순회하면서 예약비트 세팅**

디스크캐시는 영구/임시볼륨의 개수, 볼륨별 전체 섹터 수, 가용 섹터 수등 전체 볼륨에 대한 섹터들의 정보를 가지고 있다. 즉, 섹터테이블과 볼륨헤더의 정보를 바탕으로 예약 시 사용되는 정보들을 미리 계산하여 메인메모리에 저장해 둔다. 실제로 예약을 하기 전에 디스크캐시를 이용한 사전 예약을 함으로써 다음과 같은 문제들을 해결한다.

1. 현재 볼륨들에 요청된 섹터 예약을 처리할 만큼의 충분한 공간이 있는지 확인하기 위해 각 볼륨의 섹터테이블을 순회해야 한다.

2. 어떤 볼륨이 가용 섹터를 가지고 있어 예약을 처리해줄 수 있는지 확인하기 위해 각 볼륨의 섹터테이블을 순회해야 한다.
3. 섹터테이블 순회를 통해 예약공간을 찾았으나, 트랜잭션 A가 섹터를 예약하는 중에 트랜잭션 B가 예약을 해버려서 공간이 부족해지고, 실패할 수 있다(Concurrency).

큐브리드는 섹터의 예약 가능 여부를 판단할 때 직접 섹터테이블을 탐색하지 않고 미리 계산된 디스크 캐시를 통하여 이 여부를 판단한다. 덕분에 1, 2의 비효율적인 문제를 해결할 수 있다. 3번의 동시성문제 또한 디스크캐시의 정보를 바탕으로 사전예약을 함으로써 단순 값 비교의 짧은 뮤텍스(mutex)로 해결할 수 있다. 사전예약은 공간이 부족한 것을 확인하면 OS에게 추가적인 공간을 요청하는 등의 실제 예약시 필요한 모든 작업을 수행한다. 덕분에 실제 예약에서는 섹터테이블을 순회하여 정해진만큼의 비트를 set 하는 것만으로 안전하게 예약을 수행할 수 있다.

2단계 섹터 예약과정을 함수레벨로 좀 더 자세히 살펴보면 다음과 같다. 아래에서 이 과정을 자세히 설명하겠지만, 많은 함수가 사용되고 이름도 유사하여 먼저 전체적인 흐름과 함께 살펴보고 가면 이해가 더 쉬울 것이다. 들여쓰기는 함수의 호출을 뜻하며 핵심적인 함수 외에는 제외하였다.

```
disk_reserve_sectors ()
  disk_reserve_from_cache () // step 1: pre-reservation with Disk Cache
    disk_reserve_from_cache_vols () // conditional, 기존 볼륨의 가용 섹터만으로 예약이 가능할 경우
      disk_reserve_from_cache_volume()
    disk_extend () // conditional, 기존 볼륨의 가용 섹터만으로 예약이 불가능할 경우
      disk_volume_expand ()
      disk_reserve_from_cache_volume ()
      disk_add_volume () // conditional, 확장한 볼륨만으로는 예약이 불가능할 경우
      disk_reserve_from_cache_volume ()
  disk_reserve_sectors_in_volume () // step 2: actual reservation with Sector Table
```

- ***disk_reserve_sectors()***: 임의의 볼륨들에서 n개의 섹터 예약을 요청
- ***disk_reserve_from_cache()***: step 1 - 디스크 캐시를 조회하고 변경하여 사전예약을 수행한다. 이때, 사전예약을 위한 뮤텍스를 잡는다.
- ***disk_reserve_from_cache_vols()***: 요청된 n개의 섹터를 예약하기 위해 디스크 캐시의 볼륨들을 순회하며 각 볼륨에 사전예약을 요청한다(*disk_reserve_from_cache_volume()*).
- ***disk_reserve_from_cache_volume()***: 특정 볼륨에 요청받은 섹터수만큼 사전예약을 시도한다. 이때 요청량이 해당 볼륨의 가용 섹터보다 많다면 가용 섹터만큼만 사전예약을 수행한다. 사전예약 수행이란 디스크캐시에 있는 정보를 변경하는 것을 말한다.
- ***disk_extend()***: 디스크캐시를 참고하여 가용 섹터가 부족하다면 사전예약 전에 먼저 추가적인 공간을 확보한다. 이때 공간을 확보(확장, 추가)하면 확장한 볼륨에서 사전예약(*disk_reserve_from_cache_volume()*)을 시도한다.
- ***disk_volume_expand()***: 추가적인 공간확보방법 중 하나로 먼저 볼륨을 확장한다. 이때 확장하는 볼륨은 최근 추가한 마지막 볼륨이다. 한 볼륨을 최대크기로 확장하고 나서야 새로운 볼륨을 추가하므로, 확장가능한 볼륨은 항상 하나이다.
- ***disk_add_volume()***: 볼륨을 최대 크기로 확장했음에도 가용 섹터가 부족하다면 새로운 볼륨을 추가한다.
- ***disk_reserve_sectors_in_volume()***: step 2 - 실제적인 예약표시과정으로 step 1에서 약속된 내용으로 단순히 섹터테이블을 수정한다(이전 글).

Sector Reservation in details

섹터 예약과정을 자세히 살펴보기에 앞서 과정 전체에서 사용되는 몇 가지 구조체를 살펴보자.

DISK_RESERVE_CONTEXT

큐브리드는 예약과정 동안 요청진행 상황을 저장/참고하기 위하여 *DISK_RESERVE_CONTEXT* 구조체를 사용한다. 이 구조체는 예약과정 시작 시에 요청정보를 바탕으로 만들어져 예약이 끝날 때까지의 진행과정을 저장한다. 어떤 변수들을 저장하고 있는지 확인해보면 예약과정을 어느 정도 가늠해볼 수 있다. 구조체가 지닌 변수를 살펴보면 다음과 같다.

변수	설명
int nsect_total	예약 요청된 섹터수
VSID *vsid	섹터(sectid, volid)의 배열. 예약과정의 최종산출물로 예약한 섹터들의 위치이다. 섹터 요청시 배열을 할당해 넣어주면, 예약된 섹터들을 채워준다.
DISK_CACHE_VOL_RESERVE cache_vol_reserve[VOLID_MAX]	볼륨별 사전예약 섹터 수(VOLID, DKNSECTS)의 배열. Step 1에서 사전예약결과로 어떤 볼륨에서 얼마나 예약할지를 저장한다. Step 2에서 이정보를 바탕으로 볼륨을 찾아 섹터테이블을 변경한다.
int n_cache_vol_reserve	사전예약한 섹터들이 속한 볼륨의 수. cache_vol_reserve 배열의 크기.
int n_cache_reserve_remaining	아직 사전예약처리되지 못한 섹터 수. 예약 시작시 nsect_total로 초기화되고 사전예약과정에서 하나씩 감소한다. 0이되면 사전예약이 끝난다.
DKNSECTS nsects_lastvol_remaining	Step 2를 수행할 때 현재 예약중인 볼륨에서의 남은 섹터 예약량을 저장. cache_vol_reserve[volid]의 사전예약량을 초기값으로하여 섹터테이블을 변경해가며 감소시킨다.
DB_VOLPURPOSE purpose	예약 목적 (temporary or permanent).

DISK_CACHE

디스크 캐시는 앞서 말했던 것처럼 전체 볼륨들의 섹터 예약정보들을 바탕으로 예약 시 필요한 값들을 미리 계산하여 가지고 있다. 이 값들을 조회/변경함으로써 섹터테이블 변경 전에 사전예약을 수행한다. 디스크 캐시는 *DISK_CACHE* 구조체로 표현되며 *disk_Cache*라는 전역변수로 접근된다.

DISK_CACHE

변수	설명
int nvols_perm	영구타입 볼륨의 개수
int nvols_temp	임시타입 볼륨의 개수
DISK_CACHE_VOLINFO vols[LOG_MAX_DBVOLID + 1]	(DB_VOLPURPOSE purpose, DKNSECTS nsect_free) 볼륨별 목적과 가용 섹터 수, 이 볼륨별 가용 섹터를 기준으로 볼륨별 사전예약을 수행한다.
DISK_PERM_PURPOSE_INFO perm_purpose_info	영구목적 볼륨들 전체의 합산 섹터 정보와 확장 관련 정보를 지니고 있음. DISK_EXTEND_INFO(아래에서 설명)를 유일 변수로 지닌다.
DISK_TEMP_PURPOSE_INFO temp_purpose_info	임시목적 볼륨들 전체의 합산 섹터 정보와 학장 관련 정보를 지니고 있음. 영구목적과는 다르게 nsect_perm_total/free변수를 추가로 지니고 있다.
pthread_mutex_t mutex_extend	볼륨 확장을 위한 뮤텍스
int owner_extend	mutex_extend를 잡은 스레드의 ID

이 중 사전예약 시 예약 가능 여부, 확장의 필요성을 판단하기 위하여 참고하는 변수는 *perm/temp_purpose_info*의 내용이다. 확장 가능한 볼륨들의 확장정보와 함께 전체 볼륨의 가용/전체/최대 섹터 수 등을 저장하고 있다.

DISK_EXTEND_INFO

변수	설명
DKNSECTS nsect_free	볼륨들의 합산 가용 섹터 수
DKNSECTS nsect_total	볼륨들의 합산 전체 섹터 수
DKNSECTS nsect_max	볼륨들의 합산 최대 섹터 수
DKNSECTS nsect_intention	사전예약 시 공간부족으로 추가적인 공간을 확보하려 할 때, 확보하려 하는 섹터 수. 이 값보다는 크게 볼륨을 확장, 추가한다.
pthread_mutex_t mutex_reserve	사전예약을 위한 뮤텍스.
int owner_reserve	mutex_reserve 뮤텍스를 잡은 쓰레드 ID
DKNSECTS nsect_vol_max	볼륨 확장 시 최댓값. 볼륨 생성시 볼륨 헤더의 nsect_max 로 설정된다.
VOLID volid_extend	볼륨 확장 시 auto extend 대상이 되는 볼륨. 하나의 볼륨의 최댓값까지 확장되어야 새로운 볼륨이 생성되므로, 항상 마지막에 생성된 볼륨을 가리킨다.
DB_VOLTYPE voltype	볼륨 타입

이 중 *extend_info*는 개별 볼륨이 아닌 전체 볼륨들의 섹터들에 대한 정보를 합산해서 지니고 있고 temp/perm 두 종류를 나눠서 관리된다. 이 값을 바탕으로 예약할 수 있는지, 확장을 해야 하는지 등을 판단하는 데 임시목적의 데이터와 영구목적의 데이터는 저장되는 볼륨의 종류가 다르므로 이처럼 나누어서 관리한다. 또, 영구목적 볼륨은 영구타입만을 가질 수 있지만, 임시목적 볼륨은 영구타입과 임시타입 모두로 가능하다(디스크매니저 Overview). 이 때문에 임시목적의 purpose_info의 경우는 *DISK_EXTEND_INFO* 외에도 영구목적 볼륨을 위한 정보(*nsect_perm_total/free*)를 지닌다.

> nsect_max는 모든 볼륨의 최대 확장 가능크기의 합으로, 새로운 볼륨이 추가되면 증가한다. 다른 볼륨들은 섹터예약시 공간부족으로 자동으로 확장/추가될 수 있지만 임시목적/영구목적은 이러한 자동확장이 불가능하고 사용자가 임의로 추가(addvoldb)해줘야 한다. 때문에 extend_info없이 nsect_perm_total/free로 예약가능 여부만을 확인한다.

섹터 예약과정

앞서 이야기한 구조체 둘과 함수들을 이용하여 예약이 어떻게 이루어지는지를 살펴보자.

1. *DISK_RESERVE_CONTEXT* 초기화

섹터 예약은 예약요청(*disk_reserve_sectors()*)을 바탕으로 이 context변수를 초기화시켜주고 사전예약(*disk_reserve_from_cache()*)에 들어가는 것으로 시작한다.

```
/* init context */
context.nsect_total = n_sectors;  // n_sectors: 요청 섹터 수
context.n_cache_reserve_remaining = n_sectors;
context.vsidp = reserved_sectors; // 외부에서 메모리할당된 n_sectors만큼의 빈 배열, 예약된 섹터들의 정보를 입력해서 돌려준다. out parameter
context.n_cache_vol_reserve = 0;
context.purpose = purpose;
```

2. 예약목적에따른 *extend_info* 선택

디스크캐시에서 첫 번째로 확인해야 할 것은 무엇일까? 먼저 현재 볼륨들로 요청된 섹터의 예약을 처리할 수 있는지 확인해야 한다. 섹터의 예약을 처리할 수 있다는 것은 현재 볼륨들의 가용 섹터(*extend_info->nsect_free*)가 충분하냐는 것이다. 충분하다면 사전예약을 실행하면 될 것이고, 부족하다면 추가적인 섹터를 확보해야 한다. 임시목적과 영구목적의 데이터는 저장되는 볼륨이 다르므로 각 목적에 맞게 *extend_info*를 선택하여 이를 확인한다.

> 이때, 임시목적의 데이터는 임시볼륨뿐만 아니라 사용자가 addvoldb로 생성한 영구타입/임시목적의 볼륨에도 저장될 수 있다. 때문에 큐브리드는 임시목적의 요청일 경우에 영구타입/임시목적 볼륨에 먼저 섹터 예약을 수행해보고 남은 양에 대하여 임시타입의 볼륨으로 예약요청을 처리한다. 모두 처리된다면 사전예약을 마치고, 아직 모든 요청을 처리하지 못했다면 임시타입의 볼륨으로 다음의 과정으로 넘어간다. 이
>
> > 영구타입/임시목적 볼륨은 사용자가 의도적으로 생성했고, 이후에 과정인 확장과정이 없기 때문에 먼저 확인하고 가능한 만큼 사전예약을 수행한다.

3. 현재 볼륨들의 섹터들로 예약처리 시도

사전예약 요청 수가 현재 볼륨들의 가용 섹터들의 수보다 적어 모든 요청을 처리할 수 있을 것으로 예상되면 현재 볼륨들로 사전예약을 수행한다. 만약 현재 상태로 불가능하게 확실하다면 예약을 시도조차 하지 않고 다음 과정으로 넘어간다.

현재 상태에서의 사전예약처리(*disk_reserve_from_cache_vols()*)란, 요청 타입의 모든 볼륨들을 순회하며 적절한 볼륨에 사전예약 (*disk_reserve_from_cache_volume()*)을 하는 것이다. 이때, *disk_Cache->vols[]* 에 각 볼륨의 목적과 가용 섹터 수가 저장되어 있어, 이를 바탕으로 목적에 맞지 않거나 가용 섹터 수가 너무 적은 경우는 건너뛴다. 볼륨의 가용 섹터 수의 크기가 너무 적을 경우에는 파편화(fragmentation)가 발새할 수 있다. 섹터 추가는 파일로부터 요청되는데, 한 파일 내의 페이지들은 공간 지역성 (space locality)을 가지기 때문에 파편화가 심해질 경우 성능이 저하될 수 있다.

볼륨별 사전예약은 *disk_reserve_from_cache_volume()*를 통해 수행된다. 디스크 캐시의 정보들을 수정(*disk_Cache->vols[volid].nsect_free, extend_info.nsect_free* 감소 등)하고 섹터테이블을 변경하는 작업(Step 2)을 수행하기 위한 정보들(*context->cache_vol_reserve[]* 등)을 생성한다. 마지막으로 예약을 완료한 섹터수 만큼 *disk_reserve_from_cache_volume* 값을 감소시켜 외부에서 추가적인 예약이 필요한지 확인할 수있도록 한다. 이 함수는 사전예약을 수행하는 primitive 함수로 공간 확보 후의 사전예약에서도 사용된다.

4. 현재 상태로는 예약을 처리할수 없다면 공간 확보 후 사전 예약

현재 볼륨들의 상태로는 섹터 요청을 처리할 수 없다면 OS로부터 공간을 추가적으로 할당받는다. 공간 확보방법은 볼륨확장과 새로운 볼륨 추가가 있다. 볼륨 확장 후 섹터를 예약해보고, 여전히 공간이 부족할 경우 추가적인 볼륨을 생성한다.

볼륨확장: 예약요청을 처리할 수 있을 만큼 볼륨의 크기를 확장시킨다. 이 때 요청량이 볼륨헤더에 있는 해당 볼륨의 확장가능한 최댓값보다 크다면 그 값까지만 확장한다. 확장 후에는 확장한 볼륨에 사전예약을 요청한다. 이때, 이전엔 가용 섹터수가 부족하면 사전예약을 시도도 하지 않았던 것과는 다르게 볼륨을 확장한 후에는 가능한 만큼 사전예약을 수행한다. 만약 모든 요청을 처리했다면 사전예약이 종료된다.

볼륨 추가: 남은 모든 섹터의 요청이 처리될 수 있을 때까지 새로운 볼륨을 추가(*disk_add_volume()*)하고 추가한 볼륨에 사전예약(*disk_reserve_from_cache_volume()*)하는 것을 모든 요청을 처리할 수 있을때까지 반복한다.

> 한 볼륨을 최대크기까지 확장하고 나서야 새로운 볼륨을 추가하므로, 확장가능한 볼륨은 항상 하나이다. 이 볼륨을 가리키는 것이 extend_info->volid_extend 변수이다.
>
> 볼륨의 확장과 추가 과정은 별도의 포스트에서 자세히 다루도록 하겠다.

5. 사전예약 정보를 바탕으로 실제 섹터테이블을 변경한다.

3, 4에서 사전예약한 정보를 바탕으로 실제 예약 (섹터테이블 수정)을 수행한다. 사전예약 과정(*disk_reserve_from_cache_volume()*)에서 어떤 볼륨에서 몇개의 섹터를 예약할지를 결정한 상태 (*context->cache_vol_reserve[]*)이고, 이 과정에서 race condition 문제는 모두 해결해 두었으므로 이 정보를 바탕으로 안전하게 실제 예약을 수행한다 (*disk_reserve_sectors_in_volume()*). 볼륨내의 어떤 섹터를 예약할지는 사전예약 시 결정해두지 않았으므로, 각 볼륨의 섹터테이블을 순회하며 사전예약 수만큼 가용 섹터 비트를 찾아 set 해준다. 해당 과정은 STAB Cursor, *disk_stab_iterate_units()*등을 사용하는 것으로 이전 글을 참고하길 바란다.

> volheader->hint_allocset
>
> disk_reserve_sectors_in_volume()을 통해서 섹터테이블을 순회할 때, 처음부터 순회하지 않고 volheader->hint_allocset부터 순회한다. 이는 예약 시 섹터테이블을 차례로 순회하며 처리하고, 끝까지 순회 되지 않은 초기 상태의 섹터테이블이나 파일이 반납한 섹터들이나 모두 연속적으로 있을 확률이 높기 때문이다. 이를 고려하여 예약이 끝난 위치의 뒷부분을 volheader->hint_allocset로 설정하여 다음 예약요청 시 그 위치부터 예약을 시도한다.
>
> ```
> // disk_reserve_sectors_in_volume()
> /* ...(섹터테이블을 순회하며 섹터 예약)... */
> /* update hint */
> volheader->hint_allocsect = (context->vsidp - 1)->sectid + 1;
> volheader->hint_allocsect = DISK_SECTS_ROUND_DOWN (volheader->hint_allocsect); // STAB의 UNIT단위로 내림
> ```

섹터 예약해제

섹터의 예약이 파일의 생성과 추가적인 페이지할당과정에서 발생했던 것 과는 다르게, 예약해제는 파일이 파괴될 때(*file_destroy()*)만 발생한다. 볼륨의 제거나 축소가 없어 예약과정보다 비교적 단순하고, 함수의 길이들도 짧다. 과정을 간단히 요약하자면 다음과 같다.

1. *file_destroy()*에서 파일의 섹터정보를 수집하여 예약해제 요청
2. 수집한 섹터들이 속한 볼륨들을 순회하며 볼륨별 예약해제 요청
3. 각 볼륨마다 커서(cursor)를 사용하여 섹터테이블을 순회하며 유닛(unit)단위로 예약해제

3에서 유닛단위로 이루어지는 예약해제란 예약할 때의 역순으로 섹터테이블의 비트들을 0으로 바꾸고 디스크캐시를 업데이트하여 공간을 확보하는 것을 말한다. 유닛단위로 예약을 해제할 때의 루틴은 영구/임시 목적에 따라 다른데, 임시목적의 경우는 즉시 예약해제를 하지만 영구목적의 경우는 *log_append_postpone()*을 통해 트랜잭션이 commit(*log_commit()*)될 때까지 해당 작업을 미룬다. 이는 데이터베이스 예약 해제되는 섹터들이 트랜잭션이 커 밋되 기전까지는 다른 트랜잭션에 의해 사용되는 일이 없도록 하기 위함으로 이후에 트랜잭션 혹은 로그를 다루는 포스트에서 자세히 다루도록 한다.

> 관련 함수:
>
> file_destroy()
>
> disk_unreserve_ordered_sectors()
>
> disk_unreserve_ordered_sectors_without_csect()
>
> disk_unreserve_sectors_from_volume()
>
> disk_stab_unit_unreserve()

그 밖에

임시목적 볼륨과 로깅 (Logging)

코드를 분석하다 보면 임시/영구 목적에 따라 분기되는 코드들을 자주 만날 수 있다. 관련 정보가 따로 관리되는 부분에서도 분기되지만, 디스크에 쓰는 작업이 발생할 때 항상 분기되어 로딩(logging) 여부를 결정한다. 임시목적의 경우 로깅도 하지 않고 commit 될 때까지 연산을 미루는 일도 없다. 살펴본 내용 중에는 예약이나 예약해제 과정의 마지막 부분(*disk_stab_unit_[un]reserve()*)에서 섹터테이블을 변경하는 때에 분기되는 것을 확인할 수 있다.

Disk Cache Sync

두 단계로 나누어진 예약과정은 atomic 하지 않고 이 둘을 동기화시켜주는 락(Lock) 또는 래치(Latch)도 존재하지 않는다. 즉, 사전예약을 끝내고 섹터테이블을 순회하며 변경하는 동안에 캐시와 실제 디스크 내용이 다른 상태를 지닐 수 있다. 하지만 예약할 때는 항상 캐시->섹터테이블 순으로 수정하고, 예약해제 시에는 항상 섹터테이블->캐시 순으로 수정하기 때문에 섹터 예약의 기준이 되는 디스크 캐시는 항상 섹터테이블보다 적은 양의 가용 섹터를 지닌다. 따라서 이러한 불일치는 섹터 예약 시 문제를 발생시키지 않는다.

Disk Check

예약과정에서는 이러한 불일치가 문제가 되지 않지만, 디스크 캐시와 볼륨 내의 정보는 언젠가 일치되어야 할 정보이다. 이를 확인하기 위해 *disk_check()*란 함수가 존재하는데, 이때는 두 단계의 예약과정을 atomic 하게 보고 consistency를 확인해야 할 것이다. 예약과정에서 확인할 수 있는 *csect_enter_as_reader()*등이 이를 위한 것이다.

Volume Extension

Posted by : Jaeeun Kim at Jul 9, 2019

섹터 예약시 볼륨에 섹터가 부족하면?

- Volume Extension -

이전 글에서 두단계로 이루어진 예약과정, 그 중에서도 특히 디스크캐시를 기반으로 사전예약이 어떻게 처리되는지를 중점적으로 알아보았다. 섹터를 예약할 볼륨을 찾는 과정에서 요청된 양을 모두 처리할 수 없을 경우 디스크매니저는 OS로부터 추가적인 공간을 할당받는다. 이번 글에서는 이러한 추가적인 공간 확보방법에 대하여 알아본다. 이전글에서도 간단히 언급했듯이, 공간 확보방법에는 **볼륨의 확장(Expansion)**과 **새로운 볼륨의 추가**가 있다.

들어가기 전에

목적에 따른 볼륨 분류

Overview에서 볼륨을 타입과 목적에 따라 분류해서 설명했지만 다시 한번 이를 살펴보며 각 분류에 따라 어떻게 예약 및 볼륨공간 확보가 분기되는지 정리해보도록 하자. 섹터관리와 볼륨확장등에서 볼륨을 분류하는 주요한 기준은 볼륨의 목적이다. 이는 어떤 목적으로 섹터를 사용할 것이냐에 따라서 데이터의 관리 방법과 라이프싸이클이 달라지기 때문이다.

- **Permanent Purpose**: 영구목적의 데이터는 한 번 쓰여지면 영구적으로 보존되어야 하고 데이터베이스가 실행중에 Failure가 발생하더라도 데이터는 durable해야 한다. 이를 위해 WAL정책에 따라 영구목적볼륨에 데이터가 쓰여지기 전엔 로깅되며 commit된 데이터는 항상 보존된다. 영구타입 볼륨에만 영구목적 데이터가 담기며, 사용자가 임의로 생성하거나 공간이 부족해질 경우 추가적인 영구타입볼륨을 생성된다.
- **Temporary Purpose**: 임시목적의 데이터는 트랜잭션이 실행되는 동안 필요한 일시적인 데이터가 저장되는 곳이다. 리커버리시 커밋(commit)된 트랜잭션의 데이터는 필요없고, 커밋되지 않은 트랜잭션은 UNDO될테니 이러한 데이터는 로깅을 필요로 하지 않는다. 또한, 임시타입의 볼륨은 데이터베이스가 종료되거나 시작될 때 모두 제거된다. 임시목적 데이터를 저장하기 위하여 데이터베이스는 기본적으로 임시타입의 볼륨을 생성해서 사용하고, 만약 사용자가 영구타입볼륨으로 임시목적 데이터를 위한 공간을 미리 확보해두었다면 그 볼륨을 먼저 사용한다.

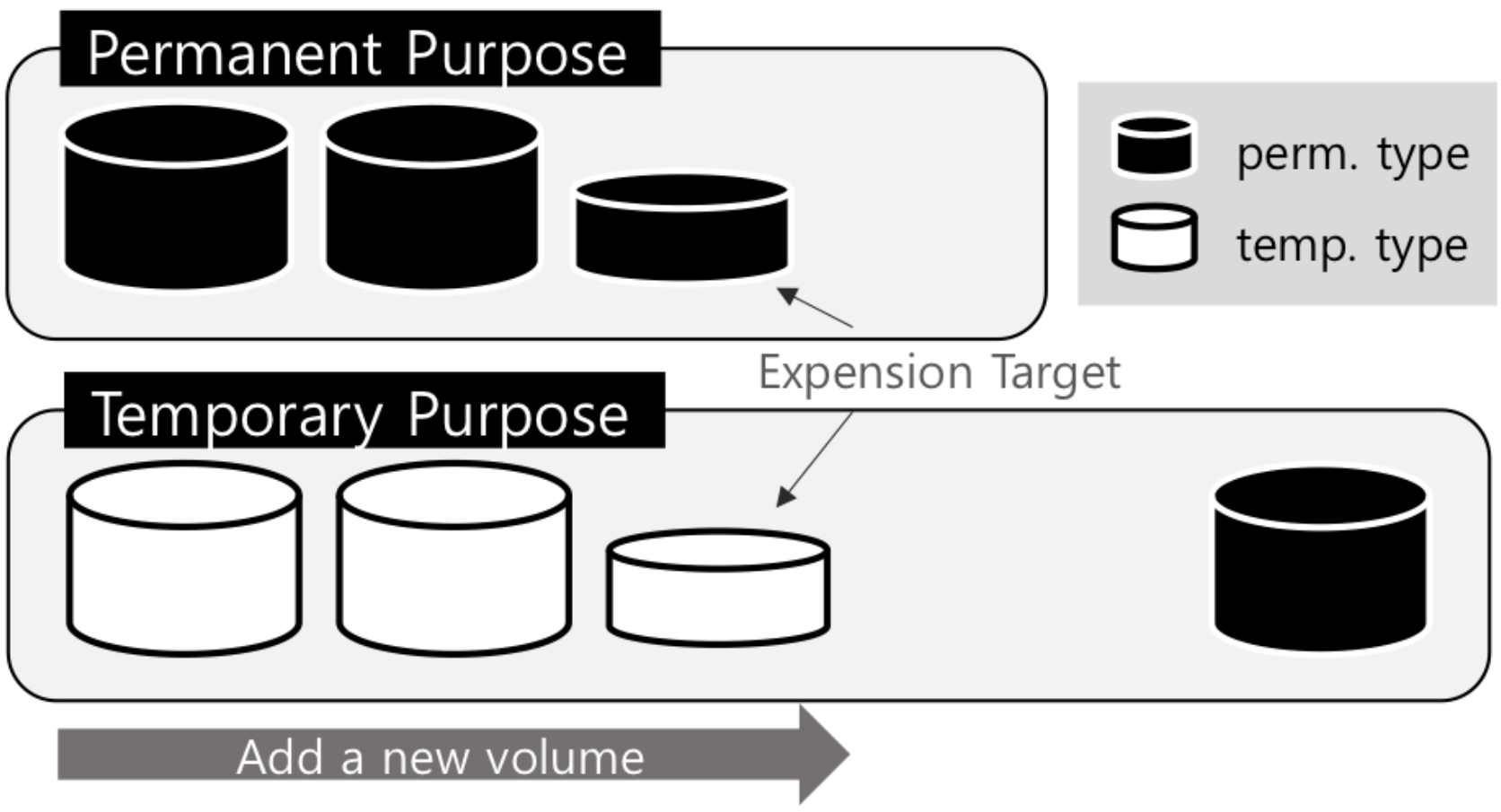


Figure 1: 목적에 따른 볼륨 확장 및 추가

석터예약시 이렇게 목적에 따라 볼륨을 선택하여 섹터를 추가하고, 필요에 따라 볼륨을 추가하기 때문에 디스크 캐시는 영구목적과 임시목적의 데이터를 별개의 *extend_info*로 관리한다(이전 글).

영구 목적의 섹터가 부족할 경우 영구타입의 볼륨을 확장 및 추가하고, 임시 목적의 섹터가 부족할 경우 임시타입의 볼륨을 확장 및 추가한다. 볼륨 공간확보는 임시/영구목적 공통적으로 마지막으로 추가한 볼륨을 최대크기까지 확장하고 새로운 볼륨을 추가하는 것을 반복한다. 임시목적/영구타입의 볼륨은 사용자가 임의로 생성한 것으로 추가/확장의 대상이 아니다.

DISK_EXTEND_INFO extend_info

이전 글에서 목적(영구, 임시)에 따른 각각의 extend_info의 섹터정보들을 바탕으로 사전예약가능 여부를 확인하였었다.
*extend_info*는 이 뿐만 아니라 볼륨의 공간확보를 위한(이름에서도 알 수 있듯이) 정보들을 담고 있다. 확장과정에서 계속 참조되므로 공간확보의 관점에서 구조체를 다시 살펴보고 가도록 하자.

DISK_EXTEND_INFO

변수	**설명**
DKNSECTS nsect_free	볼륨들의 합산 가용 섹터 수
DKNSECTS nsect_total	볼륨들의 합산 전체 섹터 수
DKNSECTS nsect_max	볼륨들의 합산 최대 섹터 수
DKNSECTS nsect_intention	사전예약 시 공간부족으로 추가적인 공간을 확보하려 할 때, 확보하려 하는 섹터 수. 이 값보다는 크게 볼륨을 확장, 추가한다.
pthread_mutex_t mutex_reserve	사전예약을 위한 뮤텍스.
int owner_reserve	mutex_reserve 뮤텍스를 잡은 쓰레드 ID
DKNSECTS nsect_vol_max	볼륨 확장 시 최댓값. 볼륨 생성시 볼륨 헤더의 nsect_max 로 설정된다.
VOLID volid_extend	볼륨 확장 시 auto expansion 대상이 되는 볼륨. 하나의 볼륨이 최대 크기까지 확장되어야 새로운 볼륨이 생성되므로, 항상 마지막에 생성된 볼륨을 가리킨다.
DB_VOLTYPE voltype	볼륨 타입

이 중 *nsect_intention*은 얼마나 공간을 확장해야 하는지를 담는다. 정확히는 *context->n_cache_reserve_remaining*의 값으로 초기화되는데, 앞서 글에서 살펴보았듯이 영구목적/임시타입 볼륨에 일부를 예약한 경우가 아니라면 요청한 섹터 수가 그대로 확장할 섹터의 수가 된다. nsect_vol_max는 볼륨을 추가할시 추가되는 볼륨의 max값으로 지정될 값이다. volid_extend는 해당 목적의 가장 최근에 생성된 볼륨의 정보를 가지고 있어, 볼륨 확장시 대상이 된다.

> *extend_info->nsect_intention*가 공간확보량을 저장하고 있지만, 실제 공간확보량은 이를 포함하여 어느정도 여유있게 수행된다.

볼륨 공간확보(*disk_extend()*)전의 가용섹터 더블 체크

볼륨 확장 및 추가는 *OS File I/O*를 포함하기 때문에 비용이 많이 드는 작업이다. 확장 중에도 기존 상태의 섹터들만으로도 처리할 수 있는 트랜잭션들이 작업을 진행할 수 있도록 확장 전에 캐시뮤텍스를 풀어 주어야 한다. 이 과정에서 공간부족을 확인 후 확장뮤텍스를 잡고 disk_extend()를 들어가기전에, 다시 한번 캐시뮤텍스를 잡고 가용섹터가 부족한지 체크하는 것을 볼 수 있다. 이는 확장이 필요하다는 사실을 인지한 후 확장에 들어가기 전에 다른 트랜잭션이 공간을 확보했을 수도 있기 때문이다. 그렇다면 캐시뮤텍스를 풀기전에 확장 뮤텍스를 잡고 캐시뮤텍스를 풀면 되지 않을까? 공간확보 과정 중 캐시의 값에 접근하는 경우가 있어 캐시 뮤텍스를 잡는데, 이것과 순서를 맞춰줘야 탐지불가능한 데드락을 피할 수 있다.

볼륨 확장

공간 확보(*disk_extend()*)과정에 들어오면 먼저 요청 타입의 가장 마지막에 추가된 볼륨을 *extend_info*의 정보를 바탕으로 최대 확장가능 크기까지 확장(*disk_volume_expand()*)한다. 볼륨 확장에 성공하면 해당 볼륨에서 가능한 만큼 섹터를 사전예약(*disk_reserve_from_cache_volume()*)한다. 볼륨 확장 과정은 단순하다.

1. 확장을 위한 시스템 오퍼레이션 로그를 적는다.

    - 볼륨헤더 변경(*volheader->nsect_total*)에 대한 로그 (*RVDK_VOLHEAD_EXPAND*)
    - 볼륨 확장에 대한 logical 로그 (*RVDK_VOLHEAD_EXPAND*)
2. 확장을 위한 IO연산 (*fileio_expand_to()*)

앞서의 글들에서 다루었던 영구목적의 데이터를 위한 볼륨 연산은 모두 로깅과정을 동반했다. 기존에는 각 연산에 대한 로깅을 언급하지 않다가 여기서 언급한 이유는 볼륨 확장의 경우에는 (nested) top action[1][2]으로 수행되어야 함을 이야기하기 위해서이다. Nested top action이란 간단히 이야기해서 해당 액션을 일으킨 부모 트랜잭션의 커밋/롤백과는 독립적으로 commit되는 트랜잭션의 연산을 말한다. 이는 nested top action을 포함한 트랜잭션이 롤백되어도 이미 commit된 nested top action은 롤백되지 않는다는 것을 의미한다. 큐브리드는 이러한 (nested) top action을 시스템 오퍼레이션 인터페이스(*log_sysop_start(), log_sysop_end()*)를 통해 제공한다.

> **볼륨 확장이 nested top action이 아니라면 어떻게 될까?**
>
> 볼륨확장 중이던 트랜잭션이 롤백될 수 있으므로 커밋되기전에 확장한 볼륨을 사용한 트랜잭션은 커밋될 수 없다. 또한, 볼륨확장 중이던 트랜잭션이 롤백되면, 확장한 볼륨을 사용한 다른 트랜잭션들도 모두 롤백되어야 한다(cascading rollback). 혹은 독립적인 진행을 위해 별개의 공간을 추가로 확보해야할 것이다.

시스템 오퍼레이션을 통한 로깅을 완료한 후에는 *fileio_expand_to()*를 통해 실제 IO연산을 수행한다. 기본 값의 페이지를 만들어 필요한 양 만큼의 페이지를 OS의 시스템콜(system call)들을 이용해 디스크에 적는다. 볼륨의 OS파일을 확장할 때 (혹은 아래에서처럼 새로운 볼륨을 생성할때) 영구타입볼륨의 경우에는 모든 페이지를 초기화하여 디스크에 적고(*fileio_initialize_pages()*), 임시타입볼륨의 경우에는 마지막 페이지만을 초기화하여 디스크에 적는다. 임시타입볼륨은 리커버리의 대상이 되지 않으므로 볼륨내의 페이지가 쓰레기값으로 초기화되어 있어도 상관 없으므로 크기확장을 위한 write만을 수행한다. 반면에 영구타입볼륨은 모든 페이지를 초기화 해줘야 한다.

> Logging, File IO 모두 기회가 될 때, 별도의 글에서 심도있게 다뤄보도록 하겠다.

볼륨 추가

볼륨확장을 통한 공간확보만으로는 섹터예약을 모두 처리 할수 없다면 새로운 볼륨을 추가(*disk_add_volume()*)한다. *extend_info*와 시스템 파라미터(*boot_Db_param*)를 바탕으로 *ext_info(DBDEF_VOL_EXT_INFO)*라는 볼륨 추가를 위한 구조체를 만들어 이 정보를 바탕으로 새로운 볼륨을 추가한다. 이 구조체에는 볼륨의 예정 크기, 볼륨의 full path등 볼륨 확장을 위한 정보와 확장된 볼륨헤더에 들어갈 볼륨 정보 등이 들어간다. 볼륨추가 과정을 정리하면 다음과 같다.

1. *extend_info*, *boot_Db_param*의 정보를 바탕으로 DBDEF_VOL_EXT_INFO 구조체를 할당, 초기화한다.
2. 새로운 볼륨을 위한 충분한 공간이 있는지 확인한다.
3. *disk_format()*을 통해 새로운 OS 파일을 만들고 볼륨헤더의 정보들과 섹터테이블등 볼륨정보를 초기화시킨다.
4. 영구타입볼륨이라면 볼륨 인포 파일(_vinf) 업데이트
5. 새로운 볼륨정보를 *boot_Db_param*에 업데이트한다.
6. 디스크 캐시를 업데이트한다.

볼륨추가는 물론 볼륨확장과 마찬가지로 시스템 오퍼레이션으로 수행된다. 볼륨 추가를 완료하면 추가된 볼륨에 사전예약을 수행(*disk_reserve_from_cache_volume()*)한다. 새로운 볼륨추가는 모든 요청을 만족시킬 수 있을 때까지 반복한다.

> boot_Db_param은 볼륨마다 있는 시스템 힙 파일에 저장된 볼륨에 대한 파라미터들을 지니고 있는 전역변수이다.

볼륨의 생성

볼륨이 생성되는 곳은 공간확보를 위해 볼륨을 추가하는 것 말고도 다음과 같은 path가 있고, 여기서 이야기한 함수들이 그대로 사용 된다.

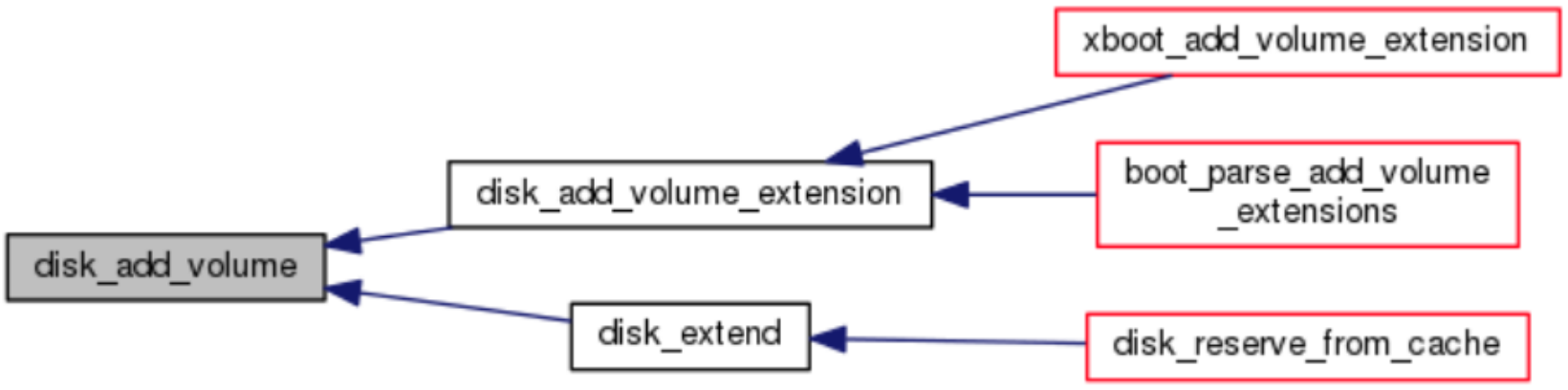


Figure 1: Callers of disk_add_volume()

볼륨이 추가되는 곳은 *disk_extend()*말고도 *disk_add_volume_extension()*이 있는데, 이는 사용자가 addvoldb 도구를 사용해 임의로 볼륨을 추가하거나, 처음 데이터베이스를 생성할 때에 파라미터로 생성할 볼륨들을 추가로 명시할 경우에 호출된다.

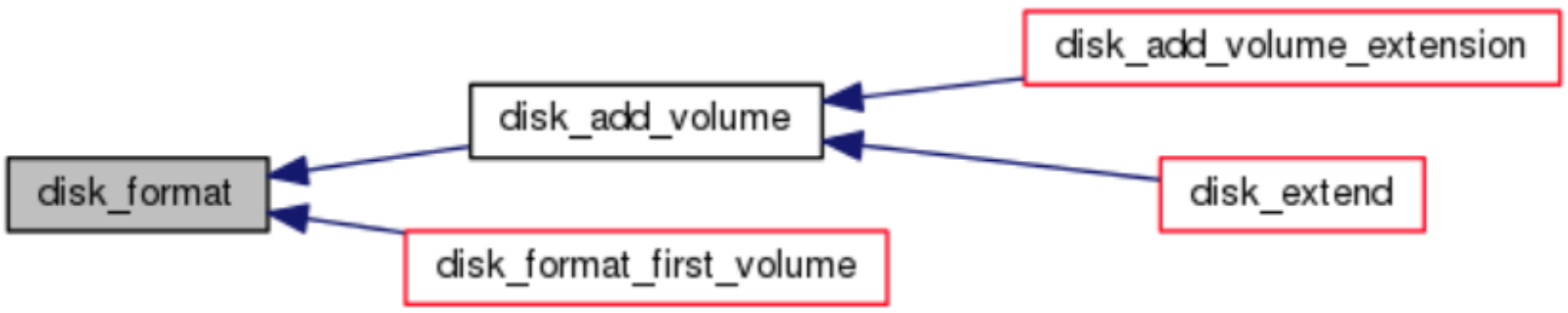


Figure 1: Callers of disk_format()

*disk_foramt()*은 볼륨 추가시 뿐만 아니라 데이터베이스 생성시 첫번째 볼륨이 생성될 때에도 호출된다.

Reference

[1] Liskov, Barbara, and Robert Scheifler. “Guardians and actions: Linguistic support for robust, distributed programs.” Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1982.

[2] Mohan, C., et al. “ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging.” *ACM Transactions on Database Systems (TODS)* 17.1 (1992): 94-162.