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.

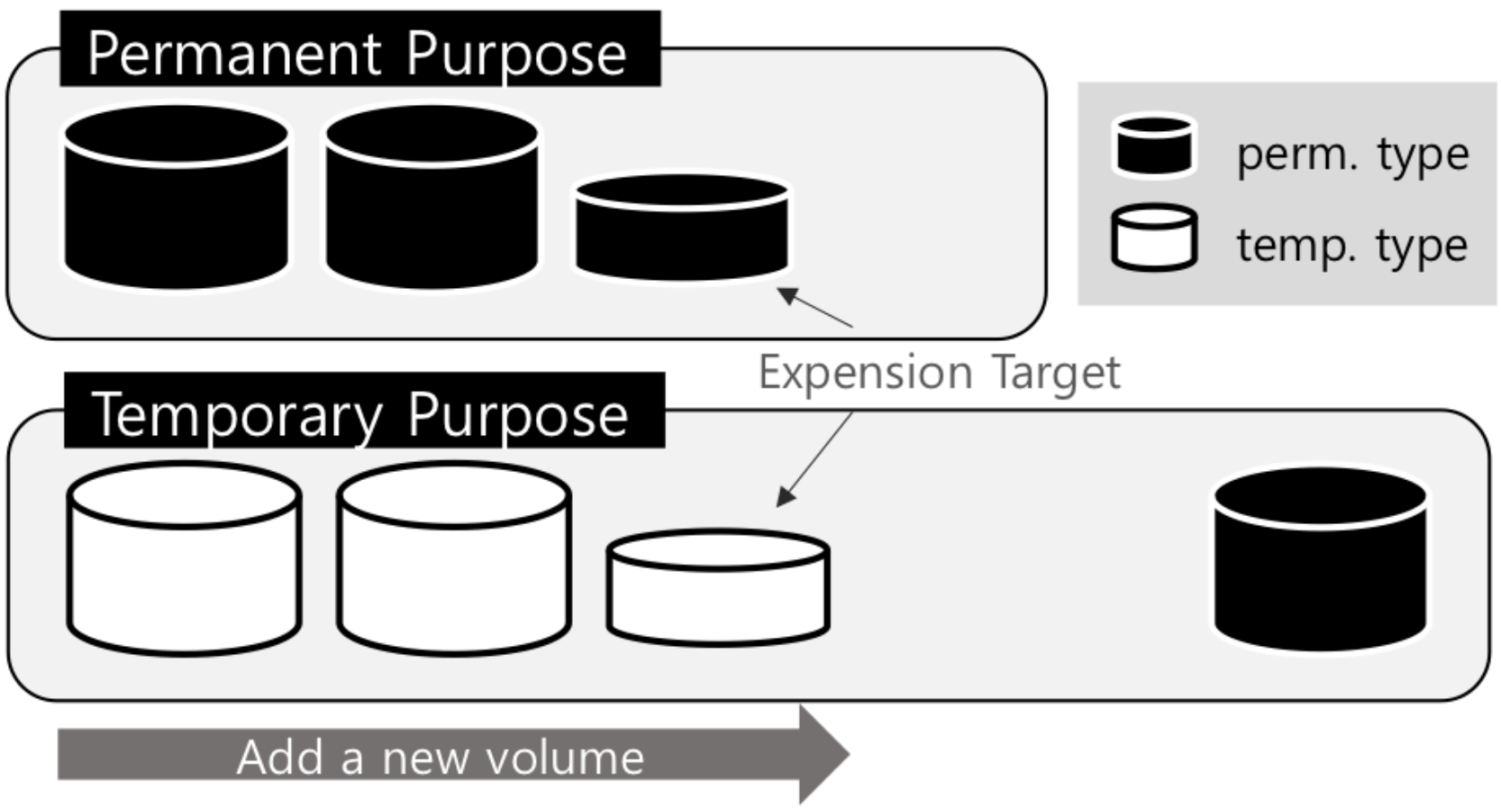

Figure 1: 목적에 따른 볼륨 확장 및 추가

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.

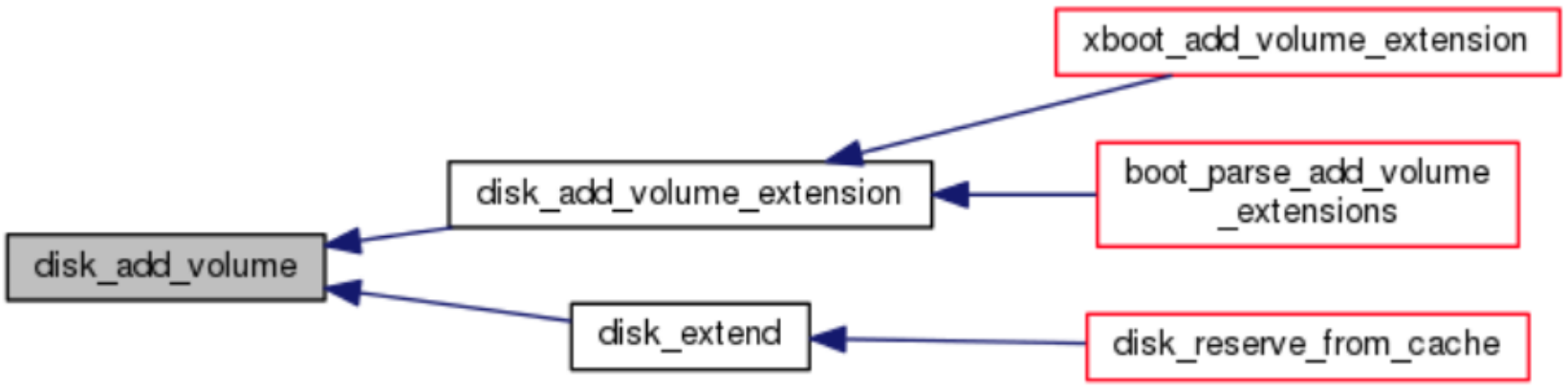

Figure 1: Callers of disk_add_volume()

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.

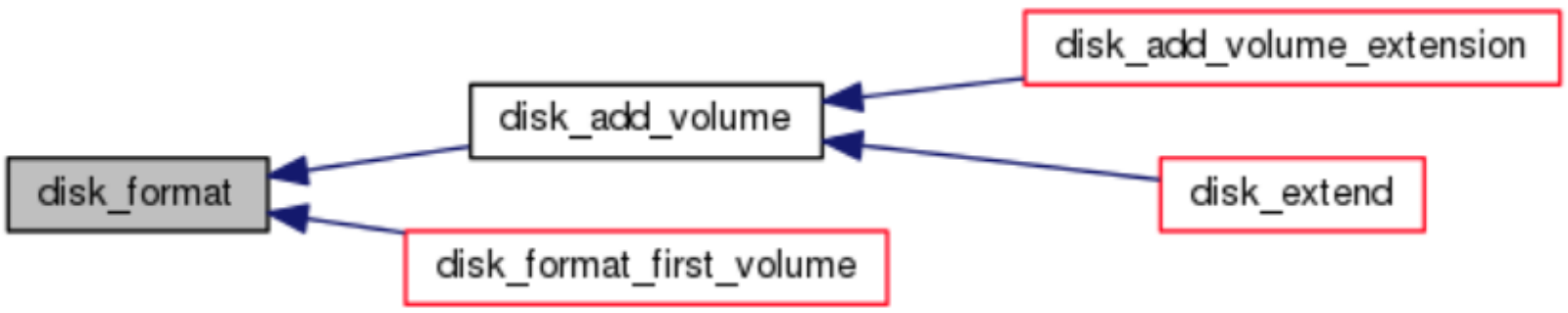

Figure 1: Callers of disk_format()

*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.