CUBRID File Create/Destroy with File Tracker and Temp Cache

Posted by : Jaeeun Kim at Aug 12, 2019

파일의 생성과 제거

이번 글에서는 파일이 생성되는 과정과 제거되는 과정에 대해 다룬다. 큐브리드 파일은 볼륨내의 섹터들로 이루어진 논리적인 집합이며, 섹터들을 페이지단위로 나눠서 필요한 만큼 이용하고 부족할 경우 디스크매니저를 통해 추가적인 섹터를 할당받아 사용하는 것을 이전 글들에서 살펴보았다. 파일의 생성과정은 이를 위한 준비과정으로 기본적인 섹터들을 미리 예약하고, 파일 헤더 및 파일테이블의 초기화를 수행한다. 또한 파일 관리를 위해 File Tracker를 사용하고, 임시파일을 위해 Temp Cache를 사용한다.

이 글에서 다루는 내용은 다음과 같다.

- 파일 생성
- File Tracker
- 파일 제거
- Temp Cache

파일 생성

이미 파일의 구조를 이전 글에서 살펴보았기 때문에 파일 생성과정(*file_create()*)을 이해하는 것은 어렵지 않다. 파일생성의 주요 과정은 다음과 같다. 통계정보추가 및 로깅등은 제외하였다.

1. 기본적으로 필요한 섹터들(from *FILE_TABLESPACE*)을 예약(*disk_reserve_sectors()*)한다.
2. 파일헤더를 초기화한다.
3. 파일테이블들을 초기화한다.
4. 3이후 정보를 바탕으로 파일헤더의 몇몇 값을 업데이트한다.
5. 영구파일의 경우, 파일 트래커(File Tracker)에 추가한다.

파일 생성함수를 보면 코드가 꽤 긴데 대부분은 파일테이블의 초기화(3), 즉 Extendible Data format를 위한 공간배정과 각 파일테이블의 아이템들 채우기, 그 과정에서 발생하는 예외를 처리하는 코드이다.

파일 트래커는 볼륨 내의 파일들의 정보를 추적하기 위한 또다른 파일로 아래에서 좀 더 자세히 살펴보겠다.

File Tracker

볼륨내에는 여러 파일들이 존재하지만 볼륨헤더에는 볼륨내 파일에 대한 정보는 존재하지 않는다. 볼륨 내 여러파일의 정보를 추적하고 저장하기 위해서 큐브리드는 볼륨 마다 파일 트래커(File Tracker)라는 영구파일을 기본적으로 생성한다. 이 파일트래커는 영구파일들의 정보를 담고 있는 영구파일이다. 즉, 파일이면서 내부적으로 볼륨내의 다른 파일들의 정보를 트래킹하고 있다.

큐브리드의 파일들은 각 파일의 목적에 맞게 각자의 헤더를 가지고 있는 경우가 많은데 (예를들어 HEAP File은 힙헤더를 가진다.) 파일트래커는 별다른 헤더가 없이 앞서 이야기한 File Extendible Data 형태로 이루어져 있다. 즉, 파일 전체가 하나의 Extendible Data이다. 새로운 영구파일이 추가될 때마다 하나의 아이템을 추가(*file_tracker_register()*)하고, 영구파일이 제거될때마다 이를 제거한다.

각 아이템은 다음과 같다.

```
typedef struct file_track_item FILE_TRACK_ITEM;
struct file_track_item
{
    INT32 fileid;           /* 4 bytes */
    INT16 volid;            /* 2 bytes */
    INT16 type;         /* 2 bytes */
    FILE_TRACK_METADATA metadata;   /* 8 bytes */
};
```

즉, 파일 트래커를 이터레이션해보면 각 파일의 위치(fileid, volid)와 타입, 그리고 추가적인 임의의 메타정보(*FILE_TRACK_METADATA*)를 알 수 있다.

파일트래커의 위치는 어떻게 찾을까?

다른 파일들의 위치는 파일트래커를 통해 금방 알 수 있다. 그렇다면 파일트래커에 접근하기 위해서는 어떻게 해야할까? 파일트래커를 찾기위해 볼륨을 다 탐색한다면 파일트래커의 존재의미가 무색해질 것이다. 이를 위하여 시스템 힙파일에 파일트래커의 위치를 저장(*boot_Db_parm->trk_vfid*)해두고, 데이터베이스 서버를 재시작(*boot_restart_server()*)할 때 *file_Tracker_vpid*라는 전역변수에 파일트래커의 위치를 저장해 두고 접근한다.

파일 트래커의 사용

*file_tracker_map()*이 파일트래커내의 파일 아이템들을 순회하며 특정작업을 하는 함수인데, 이 함수의 호출자들을 보면 파일트래커가 어떤 용도로 사용되는지 알 수 있다. 주로 볼륨내의 파일정보들을 뽑아내거나 체크하는 용도로 사용되며, 힙파일의 재사용을 위해서도 사용되는 것을 확인할 수 있다.

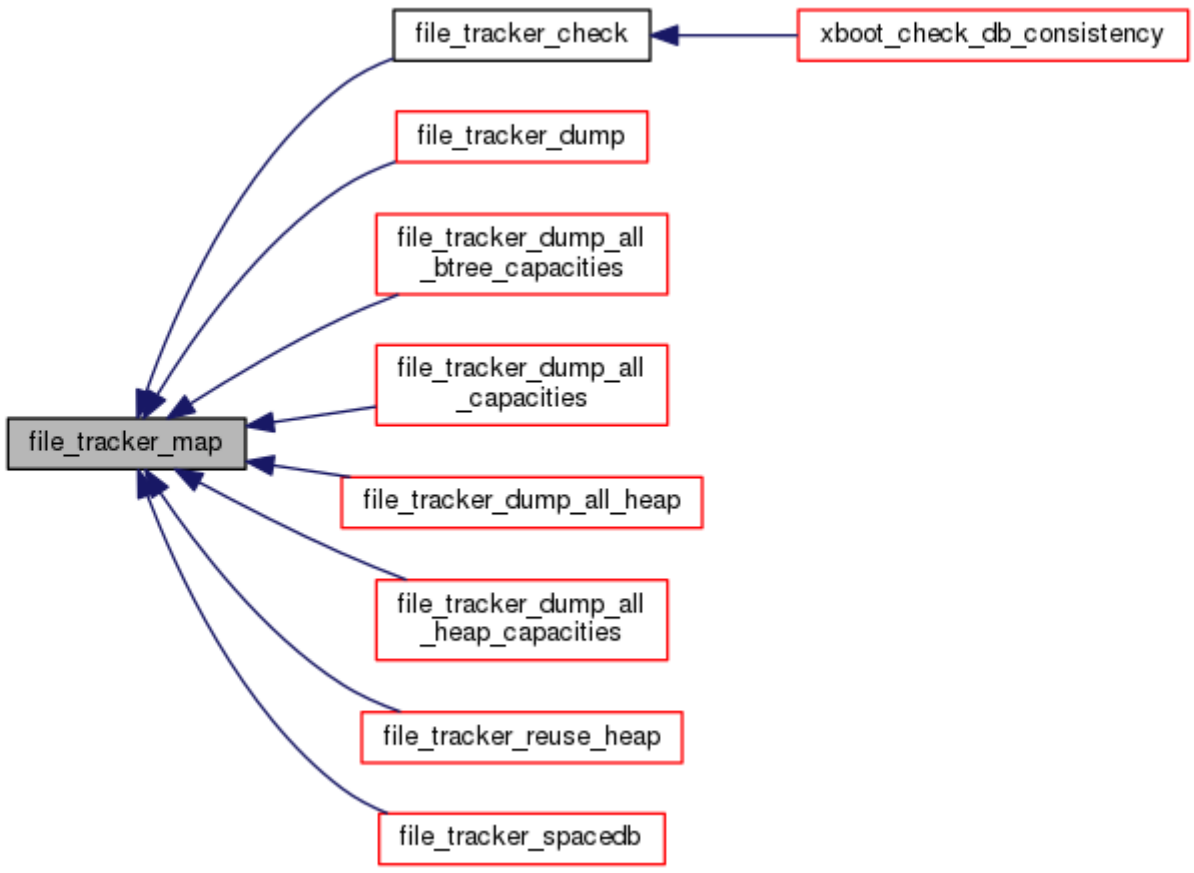

Figure 1: file_tracker_map() 호출자

파일 제거

파일 제거(*file_destroy()*)도 페이지 할당해제와 같이 트랜잭션이 커밋될 때 수행되는 postpone 명령(*file_postpone_destroy()*)으로 수행된다. 파일 제거과정은 다음과 같다.

1. 영구파일이라면 파일트래커에서 제거한다.
2. Partial/Full Sector Table에서 모든 섹터들의 VSID를 수집한다.
3. 파일테이블을 순회하며 파일테이블정보를 수집하는 동시에 모든 유저 페이지를 버퍼페이지 할당해제(*pgbuf_dealloc_page()*)한다.
4. 수집한 모든 페이지테이블과 페이지헤더를 버퍼헤이지 할당해제한다.
5. 수집한 모든 섹터들을 예약해제(*disk_unreserve_ordered_sectors()*)한다.

여기서 주의할 점은 각 페이지에 대해 수행하는 할당해제가 디스크페이지 할당해제의 file_dealloc 류의 함수가 아닌 버퍼페이지 할당해제인 *pgbuf_dealloc_page()*라는 것이다. 이는 파일의 모든 페이지를 할당해제 하므로 파일테이블을 변경하는 연산이 무의미하기 때문이다. 디스크페이지의 할당해제(*file_perm_dealloc()*)의 마지막 과정인 해당 디스크페이지의 버퍼페이지 할당해제만을 수행한다.

VSID 및 파일테이블 수집과 file_extdata_apply_funcs()

VSID와 파일테이블 수집은 각각 *FILE_VSID_COLLECTOR, FILE_FTAB_COLLECTOR* 구조체의 내용을 파일테이블을 순회하면서 채워가는 것을 말한다. 이는 모두 Extendible Data Foramt의 연산 중 하나인 *file_extdata_apply_funcs*를 통해서이다. 이는 이전에 살펴보았던 섹터테이블을 순회하며 유닛마다 특정 연산을 수행하는 *disk_stab_iterate_units()*와 유사하다.

이 함수의 선언부는 다음과 같다.

```
int
file_extdata_apply_funcs (THREAD_ENTRY * thread_p, FILE_EXTENSIBLE_DATA * extdata_in,
  FILE_EXTDATA_FUNC f_extdata, void *f_extdata_args,
  FILE_EXTDATA_ITEM_FUNC f_item, void *f_item_args,
  bool for_write, FILE_EXTENSIBLE_DATA ** extdata_out, PAGE_PTR * page_out)
```

굉장히 인자가 많고 복잡한데, 동작은 단순히 Extendible Data를 이터레이션하면서 각 컴포넌트(링크로 연결된 아이템들의 묶음. 일반적으로 하나의 페이지)마다 인자로 받은 *f_extdata()*를 수행하고, 컴포넌트내의 각 아이템마다 인자로 받은 *f_item()*을 수행한다.

예를 들어 앞서 설명한 파일 해제의 경우는 Extendible Data로 이루어진 Partial Sectors Table을 순회하며 파일테이블정보는 수집하고, 파일테이블내에 있는 각 아이템에 대해서는 모든 페이지를 할당해제하는데 이는 다음과 같다.

```
...
FILE_HEADER_GET_PART_FTAB (fhead, extdata_ftab);
is_partial = true;
error_code = file_extdata_apply_funcs (thread_p, extdata_ftab,
                file_extdata_collect_ftab_pages, &ftab_collector,
                file_sector_map_dealloc, &is_partial, true, NULL, NULL);
```

임시파일 생성과 제거

처음 디스크/파일매니저 Overview에서 살펴보았듯이 임시파일은 트랜잭션이 실행되면서 쿼리나 정렬의 중간결과들을 일시적으로 저장하는 파일로 기본적으로 트랜잭션 수행동안 생성되었다가 종료(commit/abort)될 때 함께 제거된다. 혹은, 필요에 따라 트랜잭션종속에서 벗어나 쿼리매니저에서 관리되기도 한다. 이를 위해서 앞서 살펴본 파일 생성(*file_destroy()*)과정만을 거치는 영구파일과는 다르게 임시파일은 파일의 생성과 제거시 다음과 같은 추가작업을 수행한다.

1. 트랜잭션과의 종속여부를 관리한다.
2. 매 트랜잭션 수행마다 임시파일의 생성/제거가 반복되므로 파일을 바로 제거하지 않고 캐싱(Caching)한다.

큐브리드는 File Temp Cache를 사용하여 이 두가지를 모두 관리하고, 관련 추가작업이위해 일반적인 파일생성을 래핑(wrapping)한 *file_create_temp[_internal]()*을 통해 임시 파일을 생성한다. 마찬가지로 파일을 제거할때도 바로 *file_destroy()*를 사용하지 않고 *file_temp_retire[_internal]()을* 통해 수행한다. 그럼 File Temp Cache와 함께 어떻게 임시파일이 생성제거 되는지 알아보자.

File Temp Cache

File Temp Cache에는 이전 트랜잭션에서 사용하고 반납한 임시파일들을 리스트형태로 들고 있다. 만약 새로운 임시파일 요청이 일어나면 캐싱해둔 임시파일을 먼저 분배해주고, 없을 경우에만 새로운 임시파일을만들게 된다. 또한, 각 트랜잭션별로 사용중인 임시파일들을 추적하여 트랜잭션이 종료(*log_commit_local(), log_abort_local()*)될 때 이들을 캐시로 반납하거나 파괴한다. 만약 트랜잭션의 종료와 무관하게 임시파일을 유지하고 싶다면 임시파일을 preserve시켜 쿼리매니저에게 귀속시킨다. 이를 위한 File Temp Cache의 구조와 연산은 다음과 같다. FILE_TEMPCACHE_ENTRY는 관리를 위해 임시파일을 담는 그릇이다.

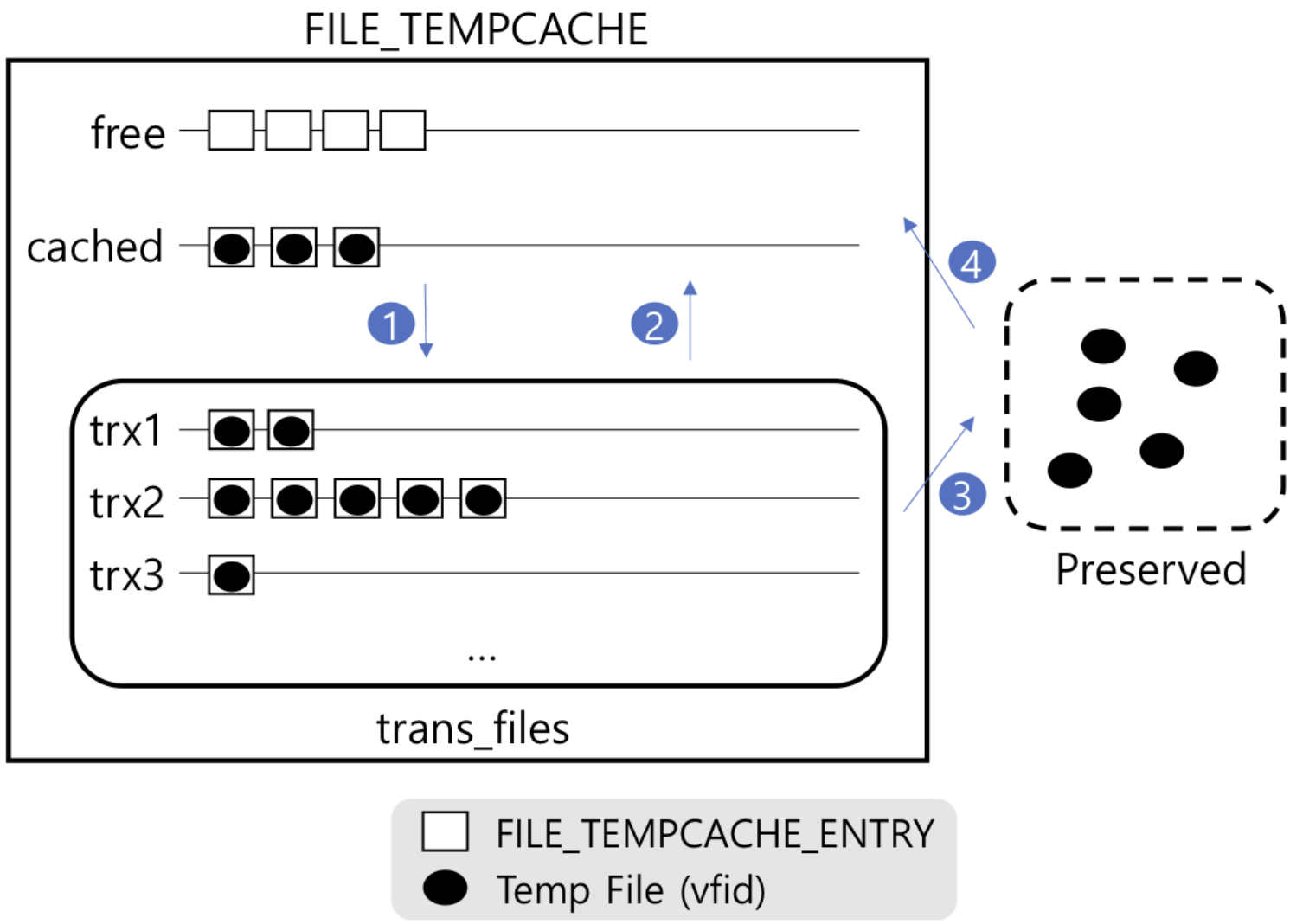

Figure 1: File Temp Cache

1. **Create**: 임시파일을 생성 (create_temp_file())할 때, 먼저 cached list를 살펴본다. 캐싱된 임시파일이 있다면 새로운 파일을 생성할 필요없이 그 파일을 사용한다. 만약 캐싱된 파일이 없다면 새로운 파일을 생성(create_file())하고 새로운 엔트리에 이를 넣어 수행중인 트랜잭션의 캐시엔트리 리스트에 삽입한다. 이 때, 엔트리는 free리스트에서 재활용할 수 있다.
2. **Retire (from trans_files)**: 임시파일의 사용이 끝났을 때, 모든 유저페이지를 초기화시키고 cached 리스트로 반납한다 (*file_temp_retire_internal()*). 이 때, cached 리스트가 가득 차있다면 임시파일은 파괴하고 엔트리는 재활용을 위해 free리스트에 삽입한다. 기본적으로 트랜잭션이 종료될 때(*log_commit_local(), log_abort_local()*) 각 트랜잭션의 캐시 엔트리리스트를 모두 retire시킨다(*file_tempcache_drop_tran_temp_files()*).
3. **Preserve**: 만약 트랜잭션이 종료될 때 임시파일을 같이 제거하고 싶지 않다면 preserve (*file_temp_preserve()*)시킨다. Preserve된 임시파일은 트랜잭션별로 관리되는 캐시엔트리리스트에서 빠지게 되어, 트랜잭션이 종료되도 제거되지 않는다.
4. **Retire (from Preserved)**: Preserve된 임시파일들은 쿼리매니저에 관리되어 이 후에 별도로 retire된다. 2와 마찬가지로 캐싱을 시도하고 가득찼을 경우 파일을 파괴한다.

Cached 리스트

cached는 numerable과 non-numerable 두가지의 리스트로 별도로 관리된다. 이는 임시파일을 캐싱할 때 유저페이지만을 초기화시키고 파일의 형태나 크기는 그대로인 상태로 저자하는데 numerable 여부에 따라서 파일의 헤더의 포맷이 다르기 때문인 것으로 보인다.

쿼리 매니저로 관리되는 preseve 임시파일은 언제 제거 될까?

쿼리매니저에 의해 관리된다는 것은 preserve된 파일을 retire시키는 함수인 *file_temp_retire_preserved()*이 쿼리매니저의 *qmgr_free_temp_file_list()*안에서 불린다는 것을 바탕으로 추측한 것으로, 정확한 시점과 관리방식은 추가적인 분석이 필요하다.

Cached, Free 리스트의 최대 크기

*FILE_TEMPCACHE*를 보면 nfree_entries_max와 ncached_max라는 각 리스트의 최대크기를 지니고 있다. 그런데 파일 retire시 cached 리스트에 캐시엔트리 삽입을 시도하는 *file_tempcache_put()*를 보면 ncached_max 가 아닌 nfree_entries_max값으로 캐시 리스트의 최대크기를 제한한다. 이는 개발자의 실수로 보인다.