ChatGPT

Android 앱 SQLite DB 삭제 데이터 복구 분석

SQLite 데이터베이스 내부 구조

SQLite 스키마 저장 구조 (sqlite_master 테이블)

SQLite에서는 데이터베이스의 스키마(테이블 및 컬럼 정보 등)를 `sqlite_master` 또는 `sqlite_schema` 라는 내장 테이블에 **평문 텍스트(SQL DDL 형태)**로 보관합니다 [1]. 예를 들어 테이블 생성 SQL문을 그대로 저장하여 컬럼 이름과 타입 등을 식별하며, 별도의 메타데이터 테이블에 컬럼 정보가 행 단위로 저장되지는 않습니다. `sqlite_master` 테이블에는 각 객체의 종류(type), 이름(name), 테이블명(tbl_name), 루트 페이지 번호(rootpage), 생성 SQL문(sql)이 저장됩니다. 따라서 **테이블 구조 변경(예: 컬럼 추가/삭제)** 시에는 이 테이블에 담긴 CREATE TABLE 문을 수정하는 방식으로 스키마가 변경됩니다 [2]. (SQLite 3.35 버전부터 **ALTER TABLE DROP COLUMN**을 지원하며, 이 역시 `sqlite_master`의 해당 테이블 정의문에서 컬럼을 제거하는 방식으로 처리됩니다 [3].)

페이지 기반 저장소와 B-트리 구조

SQLite 데이터는 **페이지(page)** 단위로 저장되며, 각 페이지는 일반적으로 4096바이트(디폴트) 크기의 고정 길이 블록입니다. 데이터베이스 파일의 **첫 페이지**(페이지 1)는 100바이트 크기의 파일 헤더를 포함하고, 이후에는 **B-트리(B-tree)** 구조로 조직된 데이터가 담깁니다 [4]. 각 테이블은 하나의 **"테이블 B-트리"**로 저장되고(인덱스는 별도의 "인덱스 B-트리"), **해당 B-트리의 루트 페이지 번호**가 `sqlite_master`에 기록됩니다 [5] [6]. B-트리는 내부 노드와 리프 노드로 구성되며, 테이블의 경우 **리프 노드**(leaf page)에 모든 행 데이터가 저장됩니다 (내부 노드는 탐색을 위한 키만 저장) [7]. 특히 **테이블 B-트리의 리프 페이지**는 첫 바이트가 0x0D (13의 문자)로 표시되며, **테이블 행(record)**들이 이 리프 페이지에 저장됩니다 [8] [9]. 각 페이지에는 **페이지 헤더**와 **셀(cell) 포인터 배열**, 그리고 실제 레코드들이 들어있는 **셀 영역**으로 구성됩니다.

페이지 헤더는 리프 페이지(0x0D)의 경우 8바이트 길이이며, 다음과 같은 구조를 가집니다 [10] [11]:

오프셋 (바이트)	크기 (바이트)	내용
0	1	페이지 유형 (0x0D = 테이블 리프 페이지)
1	2	첫 번째 freeblock의 오프셋 (없으면 0)
3	2	셀(레코드) 개수
5	2	첫 번째 셀 영역의 시작 오프셑
7	1	파편화된 프리바이트 수 (free bytes)

페이지 헤더 바로 뒤에는 **셀 포인터 배열**이 옵니다. 각 셀 포인터는 2바이트로, 해당 레코드가 저장된 **셀의 시작 위치 오프셋**을 가리킵니다 [12]. 리프 페이지의 각 **셀(Cell)**은 하나의 행 레코드를 담고 있으며, **RowID**(64비트 정수 키)와 **페이로드(payload)**로 구성됩니다 [13]. **행 페이로드**는 해당 행의 모든 컬럼 값을 **하나의 바이너리 시퀀스**로 결합한 것입니다 [6]. 이때 컬럼값들은 **테이블 정의 순서**대로 배치되고, **레코드 헤더(record header)**에 각 컬럼의 **타입 및 길이를 나타내는 시리얼 타입(serial type)** 코드들이 포함됩니다 [14] [15]. 레코드 헤더는 가변길이 정수(varint)로 인코딩된 **헤더 전체 길이**로 시작하고, 이어서 각 컬럼의 시리얼 타입들이 varint로 나열된 후, 본문에 실제 컬럼 데이터가 저장됩니다 [16] [15]. 이와 같은 구조 덕분에 SQLite는 각 행을 **하나의 연속된 바이너리 데이터**로 저장하며, 행 단위로 B-트리에 관리합니다.

데이터 저장 예시

예를 들어, (ID, 이름, 나이) 세 개의 컬럼으로 이루어진 테이블에 한 행(`ID=1, 이름='Alice', 나이=30`)이 저장될 때, 해당 행의 RowID는 1이 되고, 레코드 페이로드는 'Alice'와 30이라는 두 값이 **바이트 배열**로 결합되어 저장됩니다. 레코드 헤더에는 첫 번째 컬럼(이름)은 텍스트(String)로, 두 번째 컬럼(나이)은 정수(Integer)로 기록되었음을 나타내는 시리얼 타입 코드들이 들어가며, 헤더와 본문을 합쳐 하나의 셀이 됩니다. 이 셀은 해당 테이블 B-트리의 리프 페이지 내에 저장되고, 셀 포인터 배열에 그 위치가 기록됩니다.

(이와 같이 SQLite는 행의 각 컬럼값을 개별 저장하지 않고, **한 행 전체를 하나의 바이너리 레코드로 처리**하므로, **컬럼 삭제**와 같은 스키마 변경 시엔 모든 행 레코드를 새로 써야 할 수도 있습니다. 이를 뒤에서 자세히 다룹니다.)

SQLite에서의 데이터 삭제 동작

레코드 삭제(DML) 시의 동작

DELETE 문 등으로 행을 삭제(DML)하면, SQLite는 해당 행 레코드를 저장하고 있던 B-트리 셀을 제거하거나 표시하여 더 이상 유효하지 않게 합니다. 그러나 그 **데이터 내용 자체를 즉시 파일에서 지우지는 않으며**, 대신 해당 공간을 **재사용을 위해 표시**해 둡니다 [17] . 두 가지 경우가 있습니다:

- **페이지 내 일부 레코드 삭제:** 해당 레코드 셀이 차지하던 공간은 그 페이지 내에서 **freeblock(자유 영역 블록)**으로 표시됩니다. 즉, 삭제된 레코드의 바이트들은 그대로 남겨두고, 그 **앞부분 4바이트를 freeblock 헤더**로 덮어써 연결 리스트에 추가합니다 [18] [19] . Freeblock 헤더의 **첫 2바이트는 다음 freeblock의 오프셋(없으면 0)**, **다음 2바이트는 현재 freeblock의 크기**를 나타냅니다 [18] . 그리고 해당 페이지의 페이지 헤더의 offset 1-2 위치(첫 freeblock 오프셋 필드)가 이 새 freeblock을 가리키도록 업데이트됩니다. 이로써 이 공간이 빈 것으로 표시되지만, 실제 데이터 바이트들은 그대로 남아 있게 됩니다. 만약 그 페이지에서 일부만 삭제되고 다른 레코드들은 여전히 남아 있다면, 페이지는 **부분적으로만 채워진 상태**가 되며, 삭제된 레코드 부분은 freeblock으로 **남은 데이터의 일부를 포함한 채** 존재하게 됩니다.

- **페이지 전체가 비게 될 경우:** 만약 어떤 테이블에서 여러 행을 삭제하여 특정 페이지의 모든 레코드가 없어지면, 그 **페이지 자체가 "비어있는 페이지"**가 됩니다. SQLite는 이러한 사용되지 않는 페이지들을 추적하기 위해 **프리리스트(freelist)**를 사용합니다 [20] . 빈 페이지는 **프리리스트 트렁크(trunk) 페이지 또는 리프(leaf) 페이지**로 편성되어, 데이터베이스 파일 내에서 **활성 B-트리에는 속하지 않지만 파일 크기 내에 남아있는 페이지들의 목록**에 등록됩니다 [21] [22] . 즉, 한 페이지가 완전히 비면 해당 페이지 번호가 프리리스트에 추가되고, 그 페이지는 차후 새로운 데이터를 저장할 때 재사용될 수 있도록 남겨둡니다 [20] . 이때도 **페이지의 이전 내용(삭제된 모든 레코드 데이터)**이 페이지 내부에는 남아 있지만, 더 이상 그 테이블이나 인덱스에서 접근되지 않을 뿐입니다.

요약하면, SQLite는 **기본 설정**에서 데이터 삭제 시 "지우는(mark) 것처럼 표시만 하고 실제 바이트는 치우지 않는" 전략을 사용합니다 [17] . 삭제된 레코드의 내용은 그 페이지 내에 **freeblock**이나 **미할당(unallocated) 영역**으로 남거나, 페이지 전체가 비었다면 **프리리스트**로 남아서, **나중에 새로운 데이터로 덮어써질 때까지** 파일 내에 흔적으로 존재하게 됩니다.

이때 **페이지 내 미할당 공간(unallocated space)**도 고려할 필요가 있습니다. 예를 들어, 어떤 페이지에 레코드를 삽입했다가 곧바로 삭제하면, 그 레코드가 차지했던 셀이 freeblock으로 표시됩니다. 그러나 페이지의 가장 끝에 위치한 레코드가 삭제된 경우에는 셀 포인터 배열의 끝부분과 페이지 데이터 끝 사이에 **연결되지 않은 남는 공간**이 생길 수 있습니다 [23] [24] . SQLite는 필요에 따라 페이지를 **재편성(defragmentation)**하여 freeblock들을 통합하거나 말단의 빈 공간을 확보하기도 하는데, 이 과정에서 일부 삭제 레코드 조각이 페이지 내 **unallocated 영역**으로 남겨지기도 합니다 [25] [26] . 즉, **페이지 내에서 참조되지 않는 모든 공간**(freeblock + unallocated 영역)이 삭제된 데이터의 흔적을 담고 있을 가능성이 있습니다 [27] . 이러한 영역은 **어떤 테이블에도 속하지 않는 파편 데이터**이므로 분석이 어렵지만, 포렌식 관점에서는 **특정 패턴이나 시그니처**를 통해 이들 속에서 유의미한 데이터를 찾아낼 수 있습니다 [28] [29] .

컬럼 삭제(DDL) 시의 동작

DDL을 통해 테이블의 컬럼을 삭제하는 경우(예: `ALTER TABLE 테이블명 DROP COLUMN 컬럼명`), SQLite는 내부적으로 **해당 테이블의 모든 행 레코드를 새로 써서 해당 컬럼의 데이터를 제거**합니다 [3] . 이는 간단히 스키마 텍스트만 바꾸는 것이 아니라, 각 행의 바이너리 레코드에서 그 컬럼에 해당하는 부분을 없애야 하기 때문에, **테이블을 새로 만드는 수준의 작업**이 수행됩니다 [3] . SQLite 문서에 따르면, DROP COLUMN 실행 시 **"해당 컬럼과 연관된 데이터를 정리(purge)하기 위해 테이블 컨텐츠를 모두 다시 쓴다"**고 합니다 [3] . 구체적으로, 다음과 같은 절차가 일어납니다 (버전 3.35 기준):

1. 우선 `sqlite_master`의 CREATE TABLE SQL에서 해당 컬럼 정의를 제거하고 스키마 유효성을 검사합니다 [2] .
2. 그런 다음 기존 테이블의 데이터를 새로운 형식으로 옮기는데, 이는 **새로운 임시 테이블을 생성하고 (컬럼이 제거된 구조) 기존 데이터에서 해당 컬럼을 제외한 나머지 컬럼 값들을 SELECT하여 새 테이블에 INSERT** 하는 방식으로 구현됩니다 [30] [31] .
3. 데이터 이동이 완료되면 기존 테이블를 드롭(drop)하고, 임시 새 테이블의 이름을 원래 테이블 이름으로 변경합니다 [32] [33] .
4. 인덱스나 트리거 등 부가 요소들도 모두 수정/재생성됩니다.

이 일련의 과정은 SQLite 엔진 내부에서 자동으로 트랜잭션 내에 수행되며, **결과적으로 해당 컬럼이 제외된 새로운 테이블 구조로 데이터베이스가 갱신**됩니다 [34] [32] .

포렌식적으로 볼 때, **컬럼 삭제 전 원래 테이블에 저장되어 있던 컬럼 데이터의 잔존 여부**가 중요합니다. 위 과정에서 **기존 테이블을 DROP**하게 되는데, DROP TABLE은 해당 테이블의 모든 페이지를 **프리리스트에 추가**하는 방식으로 처리됩니다. 즉, **원래 테이블의 데이터가 담겼던 페이지들**이 한꺼번에 자유 공간으로 표시되고 [20] [21] , 이 페이지들의 내용(삭제된 컬럼을 포함한 기존 레코드 데이터)은 파일 내에 그대로 남게 됩니다. 새로 생성된 테이블은 보통 **다른 페이지들**(새로 할당하거나 혹은 해제된 페이지를 재사용)로 채워지므로, 운이 좋다면 **옛 테이블의 페이지들은 아직 덮어써지지 않은 채 파일에 존재하게** 됩니다. 이는 마치 대량의 행 데이터를 한꺼번에 삭제(drop)한 것과 같아서, **삭제된 컬럼의 데이터가 해당 레코드 조각들과 함께** 이전 테이블의 페이지들에 남아있을 수 있다는 의미입니다. 특히, SQLite는 DROP COLUMN 시 **모든 데이터를 다시 쓰기 때문에** 삭제된 컬럼이 **행 레코드의 마지막 컬럼이었다면** 원래 레코드의 끝부분이 잘려나간 형태로 남을 수 있습니다. 하지만 **삭제된 컬럼이 중간 컬럼인 경우**에도 SQLite는 임시 테이블로 옮기는 방식을 사용하므로, 결과적으로 기존 전체 레코드가 통째로 삭제된 것으로 간주할 수 있습니다 (전체 페이지들이 자유화됨).

정리하면, **컬럼 삭제(DDL)**를 수행하면 **해당 컬럼의 논리적인 데이터는 모두 제거**되지만, **물리적으로는** 그 컬럼 값을 포함했던 기존 레코드들이 **프리리스트 페이지 형태로 파일 내에 잔존**하게 될 가능성이 높습니다. 이러한 잔존 데이터는 일반적인 DB 접근으로는 보이지 않지만, **파일 레벨 분석으로 복구가 가능**합니다. (단, 컬럼 삭제 직후 **VACUUM**을 수행했다면 이러한 잔존 데이터 페이지들이 완전히 제거될 수 있습니다. 다음 섹션에서 설명합니다.)

삭제된 데이터 복구 가능성 및 영향 요인

프리리스트와 Freeblock 잔존 데이터

앞서 설명한 대로, **기본 설정의 SQLite는 데이터를 삭제해도 즉시 파일에서 지우지 않기 때문에**, 삭제된 레코드나 컬럼의 데이터가 **파일 내부에 흔적으로 남아 있을 가능성**이 큽니다. 이 잔존 데이터는 크게 두 가지 형태로 존재합니다:

- **프리리스트 페이지**: 삭제 또는 컬럼드롭으로 **완전히 비게 된 페이지들**로, 데이터베이스 헤더에 기록된 프리리스트에 속합니다 [20] . 이러한 페이지들은 **더 이상 테이블이나 인덱스에 속하지 않으므로** 일반 쿼리로는 접근 불가능하지만, 파일 상에는 그대로 남아 있습니다. **프리리스트 트렁크 페이지**는 다음 트렁크에 대한 포인터와 여러 개의 리프 페이지 번호 목록만 담고 있고, **프리리스트 리프 페이지**는 내용은 없지만 이전 사용 시의 데이터가 그대로 남아 있습니다 [21] [35] . 즉, 프리리스트에 속한 페이지들은 **과거에 사용된 데이터의 원본 바이너리를 그대로 포함**하고 있으며, 단지 헤더에 의해 "비어있음"으로 표시되어 있을 뿐입니다. 이러한 페이지들에 들어있는

삭제 전 레코드 전체를 파싱하면 유용한 정보를 얻을 수 있습니다. 다행히 SQLite는 기본적으로 **auto_vacuum 옵션이 꺼져(off)** 있어서, 삭제된 페이지들이 즉시 파일에서 제거(truncate)되지 않고 프리리스트로 남아 있게 됩니다 [36] [37] . 이는 포렌식 전문가에게 유리한 부분으로, Belkasoft 등에 따르면 "기본 설정에서는 프리리스트가 유지되므로 삭제 데이터 발견이 가능하다"고 합니다 [36] [38] .

- **페이지 내 freeblock 및 미할당 영역**: 완전히 페이지가 비워지지 않고 **부분적으로 삭제된 경우**, 그 페이지의 내부에 삭제 레코드 조각이 남습니다. 삭제된 각 레코드는 freeblock으로 표시되며, 앞서 말한 **4바이트 헤더만 덮어씌운 후 나머지 바이트는 원래 데이터 그대로** 남겨둔 형태입니다 [18] . 따라서 freeblock의 **헤더 이후 부분**을 해독하면 삭제된 행의 실제 컬럼 값들을 얻을 수 있습니다. 또한 페이지 내의 **unallocated** (할당되지 않은) 공간에도 레코드 조각이 남을 수 있는데, 이는 freeblock으로 관리되지 않는 자투리 공간입니다 [25] [26] . Belkasoft 설명에 따르면, "SQLite의 미할당 공간은 페이지 단편 조각들로, 이전에 사용된 페이지의 일부가 들어있을 수 있다"고 합니다 [28] . 이러한 조각들은 **어느 테이블의 것인지 추적할 수 없을 정도로 산발적**일 수 있지만, 특정 **문자열 패턴 또는 시그니처**로 검색하면 의미 있는 삭제 데이터를 찾을 수 있습니다 [39] [29] . (예를 들어, 삭제된 SMS의 전화번호나 텍스트 조각 등으로 검색.)

SQLite가 **freeblock을 관리하는 방식**은 비교적 간단하지만, 실제 **삭제 레코드 복구 시에는 몇 가지 한계**도 있습니다. Pawlaszczyk 등의 연구에 따르면, 레코드 삭제 시 **행 레코드의 처음 몇 바이트(최대 4바이트)**가 freeblock 헤더로 덮여쓰이기 때문에, **RowID나 레코드 전체 길이 등의 정보가 일부 손실**될 수 있습니다 [19] [40] . 하지만 일반적으로 컬럼 값 데이터 대부분은 그대로 남아서, 헤더 일부를 추론하거나 이웃 레코드와 비교함으로써 복구할 수 있다고 합니다 [41] [42] . 요약하면, **삭제된 레코드의 나머지 내용이 덮어써지지 않았다면 충분히 복구 가능**하며, 일부 헤더 정보가 사라졌더라도 **테이블 스키마로부터 예상되는 시리얼 타입 패턴을 이용**해 해당 바이트열을 레코드로 재조립할 수 있습니다 [43] [44] .

VACUUM 및 Auto-vacuum의 영향

VACUUM 명령은 SQLite 데이터베이스 파일을 **완전히 재구성하여 불필요한 공간을 제거**하는 기능입니다. VACUUM을 실행하면 새로운 파일에 현존하는 모든 데이터를 연속적으로 복사한 후 원본을 대체하는 방식으로 동작하므로, **프리리스트 페이지나 페이지 내 freeblock 등에 남아있던 삭제 데이터가 모두 사라지게 됩니다** [45] . 즉, VACUUM을 한 번이라도 수행하면 해당 시점까지의 **삭제된 데이터 잔존 흔적은 대부분 사라지고**, 데이터베이스 파일 크기도 줄어들게 됩니다 [45] . 따라서 **VACUUM이 실행된 DB에서는 포렌식 복구가 매우 어려워지며**, 디스크 레벨에서 과거 파일 조각을 찾는 방법밖에 없을 수 있습니다.

SQLite 설정 중 `auto_vacuum` 옵션도 삭제 데이터 잔존에 큰 영향을 줍니다. 기본값(`OFF`)에서는 위에 설명한 대로 프리리스트에 남기지만, `FULL` **모드(auto_vacuum=FULL)**일 경우 **트랜잭션 커밋마다 프리리스트 페이지들을 파일 끝으로 옮긴 뒤 파일을 잘라내기(truncate)** 때문에 **아예 프리리스트를 남기지 않습니다** [37] [46] . 다만, 이 과정에서도 실제 페이지의 데이터를 즉각 덮어쓰는 것은 아니므로, 잘려나간 파일 뒷부분이 물리 디스크 상에 남아있을 가능성은 있습니다 [46] . (이는 파일 시스템 관점의 잔존에 해당하며, 아래에서 다룹니다.) 한편 `INCREMENTAL` **모드**는 부분적으로 자동 Vacuum을 수행하는 절충안인데, 이 경우도 시간이 지나면 결국 삭제 페이지들이 제거되므로 잔존 흔적이 줄어듭니다.

결론적으로, **삭제된 데이터 복구 가능성은 VACUUM/auto_vacuum 여부에 크게 좌우**됩니다. **VACUUM이나 auto_vacuum(FULL)**이 **한 번도 수행되지 않은 DB**에서는 상당수 삭제 레코드가 파일 내에 남아 있을 것이며, **반대로 VACUUM이 최근에 수행되었다면** 논리적으로도 물리적으로도 복구가 힘들 수 있습니다. 따라서 포렌식 시에는 **DB 파일의 헤더 정보를 확인**하여 마지막 Vacuum 시점이나 auto_vacuum 설정 등을 고려해야 합니다. (예: `PRAGMA auto_vacuum;` 값 확인, 또는 `sqlite_sequence` 등 테이블 변화 이력으로 유추.)

Write-Ahead Log (WAL)과 Rollback Journal의 역할

SQLite는 트랜잭션 무결성을 위해 **저널(journal)**을 사용합니다. 고전적인 **롤백 저널(rollback journal)** 모드에서는 트랜잭션 중 변경된 페이지의 이전 상태를 .journal 임시 파일에 기록하고, 트랜잭션이 완료되면 해당 파일을 삭제합니

다. 한편 **WAL(Write-Ahead Log)** 모드에서는 .wal 파일에 변경 내용(페이지 단위 프레임)을 계속 Append하고, 일정 조건(기본 1000 페이지 이상 등)에서 **체크포인트(checkpoint)**를 실행하여 WAL의 내용을 메인 DB에 병합합니다 [47]. **스마트폰 환경**에서 성능과 동시접근을 위해 WAL 모드가 사용되는 경우가 많습니다.

WAL 모드의 특징은 **메인 데이터베이스 파일을 바로 수정하지 않고**, 변경된 페이지들을 WAL 파일에 기록한 채로 둘 수 있다는 점입니다 [48]. 이로 인해 **삭제나 업데이트된 데이터의 "이전 버전"이 메인 DB에 그대로 남아 있는 상황**이 흔히 발생합니다. 예를 들어, 어떤 메시지를 편집하거나 삭제한 경우:

- **편집의 경우** WAL에는 수정된 새 레코드가 기록되지만, **체크포인트가 되기 전까지** 메인 DB 파일에는 **옛 내용(편집 전 텍스트)**이 남아 있습니다. 따라서 WAL과 메인 파일을 함께 분석하면 **동일 레코드의 두 버전**(예: 편집 전후 메시지)을 볼 수도 있습니다 [49]. Belkasoft에 따르면 WAL을 면밀히 분석함으로써 "보내기 후 수정된 채팅 메시지의 최초 버전을 발견"하는 식의 인사이트를 얻을 수 있다고 합니다 [49].

- **삭제의 경우** DELETE 실행 시, WAL에는 해당 레코드가 제거된 페이지 이미지가 기록됩니다. 하지만 **체크포인트 전까지는 메인 DB에 그 레코드가 여전히 존재**하므로, 겉으로는 삭제된 것처럼 보여도 파일 내에는 남아 있는 셈입니다. 일반 SQLite API로 DB를 열면 WAL의 변경사항을 우선 적용하여 보여주므로 삭제 후의 상태를 나타내지만, **포렌식 도구를 사용해 WAL과 메인 파일을 따로 살펴보면 삭제 전 데이터도 확인**할 수 있습니다. (만약 DB를 복사할 때 .wal 파일을 누락한다면, 메인 DB에 과거 내용이 고스란히 남아 있을 수도 있습니다.)

- 또한 WAL에는 **최근 커밋된 신규 레코드나 갱신된 데이터**가 메인에 반영되지 않은 채 저장되어 있을 수 있습니다 [48]. 이는 **"아직 커밋되지 않은(not yet checkpointed) 데이터"**를 담고 있는 것으로 볼 수 있는데, 포렌식 관점에서는 이 역시 중요합니다. 예를 들어, 앱이 강제 종료되어 WAL이 그대로 남았다면 WAL 속에만 있고 메인 DB에는 없는 **미반영 데이터**가 존재할 수 있습니다.

WAL/저널 분석의 중요한 점은, **일반 데이터베이스 뷰어나 SQLite 엔진을 통해 접근하면 증거를 잃을 위험**이 있다는 것입니다. 예를 들어, DB Browser for SQLite와 같은 툴은 DB를 열 때 자동으로 **WAL 체크포인트를 수행**하거나, 종료 시 WAL을 적용(commit)해버리는 경우가 있습니다 [50]. 만약 원본 매체에서 직접 이런 도구를 실행하면 **증거가 되는 WAL 내용이 메인 DB에 병합**되면서 .wal 파일이 삭제되어(Hash 변조) 버릴 수 있습니다 [50]. 그러므로 포렌식에선 **절대로 원본 DB를 일반 툴로 쓰기 모드로 열지 말고**, 가능하면 복제본을 만들거나 전용 **읽기 전용 파서**를 사용해야 합니다 [51]. Belkasoft X 등 전문 포렌식 도구들은 **표준 SQLite API를 사용하지 않고**, WAL과 저널 파일을 **파일 레벨로 파싱**하여 내용을 보여주므로 안전하게 두 파일을 모두 분석할 수 있습니다 [52] [53].

Rollback Journal 파일(.journal)도 유사하게 포렌식 가치를 가집니다. 롤백 저널 모드에서는 커밋 시 .journal 파일을 지워버리지만, 예기치 못하게 남겨진 저널 파일이 있다면 거기에 **트랜잭션 시작 직전의 페이지 데이터**가 들어있어서, **삭제/변경 전의 데이터 복구**에 활용될 수 있습니다. 예컨대, 어떤 DB가 롤백 저널 모드이고 대량 삭제 트랜잭션 도중 앱이 Crash됐다면, .journal 파일에 삭제 전 데이터 페이지들이 들어있을 수 있습니다. 포렌식 툴은 이런 .journal 파일을 찾아 자동으로 분석하여, 거기서 **삭제되기 전 레코드**들을 추출해 주기도 합니다 [54] [55].

요약하면, **WAL/저널 파일**은 메인 DB에 반영되지 않았거나 과거 버전의 데이터를 담고 있어 **중요한 복구 단서**가 됩니다. 특히 WAL은 설정과 사용 패턴에 따라 꽤 오래(수천 페이지, 또는 앱이 명시적으로 체크포인트하지 않는 한 계속) 유지되므로, 모바일 앱의 경우 **수일~수주 전에 삭제된 데이터도 WAL에 남아 있을 수 있습니다** [47]. 따라서 모바일 기기에서 DB를 수집할 때에는 .db 파일과 함께 .wal, .journal 파일을 모두 확보*하고, 안전하게 복사하여 분석하는 것이 필수적입니다.

물리 디스크 상의 잔존 흔적

SQLite 파일 내부에 남은 흔적 외에도, **파일 시스템/디스크 관점**에서 삭제 데이터를 찾을 여지도 있습니다. 두 가지 경우를 고려해볼 수 있습니다:

- **파일 크기 감소에 따른 디스크 단편(slack)**: VACUUM이나 auto_vacuum FULL에 의해 파일 크기가 줄어들거나, DB 파일을 아예 삭제/교체한 경우, 원래 파일이 차지하고 있던 디스크 영역에 데이터가 남아 있을 수 있습니다. 예를 들어 VACUUM 전 DB 파일이 10MB였는데 후에 8MB로 줄었다면, 마지막 2MB 영역은 파일 시스템에서는 **해제된 클러스터들**로 남지만 물리적으로 그 안에 있던 SQLite 페이지 데이터는 그대로일 수 있습니다. PC의 HDD/SSD라면 이 클러스터들을 포렌식 이미징으로 덤프하여 **삭제된 파일 조각을 카빙(carving)** 할 수 있습니다. 모바일 기기의 플래시 스토리지의 경우도, 파일 삭제/축소 시 **즉각적인 데이터 소거(TRIM)**가 이뤄지지 않았다면, 내부 플래시 메모리에 데이터가 잠시 남아 있을 가능성이 있습니다. 다만 현대 스마트폰의 **Ext4** 파일시스템 등은 일정 주기나 조건에 따라 TRIM을 수행하므로, 시간이 많이 경과한 경우에는 해당 섹터들이 이미 초기화되었을 수도 있습니다.

- **DB 파일 내 미사용 공간의 파일 시스템 Slack**: 만약 DB 파일이 파일 끝에 일부 **안 쓰는 공간**(예: 마지막 할당 클러스터의 일부만 사용 중)이 있다면, 그 **파일 slack 공간**에도 이전 쓰레기 데이터가 남아있을 수 있습니다. 그러나 이는 SQLite 고유의 삭제 흔적이라기보다는 파일 시스템 차원의 흔적입니다.

물리적 디스크 분석은 주로 **전체 디스크 이미징(physical extraction)**을 통해 수행됩니다. 논리적으로 추출한 DB 파일에는 이미 잘려나간 부분이 없지만, **물리 이미징**을 했다면 DB 파일의 과거 버전이나 삭제된 WAL 파일 등의 **파일 단위 잔재**도 찾아낼 수 있습니다. 예를 들어, 한 앱이 DB를 삭제하고 새로 만들었다면, 이전 DB 파일의 내용이 디스크에 남아 있을 수 있는데, 파일명으로 찾기 어렵더라도 **"SQLite format 3"** 헤더 서명을 검색하면 발견될 수 있습니다. 또한, **삭제된 WAL이나 journal 파일** 이름(예: `msg.db-wal`가 삭제되었다면)의 흔적을 검색해볼 수도 있습니다.

하지만 스마트폰에서는 **파일 단위의 삭제**가 일반적으로 **플래시 메모리 GC/웨어레벨링**과 결합되어 있어 PC보다 복구가 어렵습니다. 예를 들어, 안드로이드에서 사용자 데이터 파티션은 암호화되어 있을 수도 있고, TRIM이 수행되면 실제 데이터 영역이 지워져 복구 불가능해집니다. 따라서 **가장 신뢰도 높은 방법은 DB 파일 자체 내의 잔존 데이터(WAL, freelist 등)**를 활용하는 것입니다. 물리 디스크 레벨 복구는 **마지막 수단**으로 간주되며, 주로 **치명적인 손상**(DB 파일 자체가 손상/삭제된 경우)이나 **VACUUM 등으로 내부 흔적이 모두 사라진 경우**에 시도됩니다.

SQLite 삭제 데이터 복구: 도구 및 절차

모바일 기기 SQLite 데이터 복구 사례와 도구

현대의 스마트폰은 대부분 앱 데이터를 SQLite DB로 저장하기 때문에, **삭제된 메시지나 로그를 복구**해야 하는 상황이 많이 발생합니다 [56]. 예컨대, **카카오톡이나 WhatsApp** 채팅 내역(DB 형식), **통화 기록**(DB), **SMS 메시지**(DB) 등이 대표적입니다. 실제로 **텔레그램(메신저) 대화**의 삭제 여부를 포렌식 툴로 분석한 사례를 보면, Belkasoft X 툴이 **삭제된 채팅을 다른 정상 채팅과 함께 보여주고, "Is deleted" 속성으로 표시**해주어 해당 항목이 삭제되었음을 알 수 있었습니다 [55]. 이처럼 전문 포렌식 도구들은 삭제된 레코드도 별도 표시하여 사용자에게 제공하므로, 수사에 활용되고 있습니다.

SQLite 삭제 데이터 복구를 위해 활용되는 주요 도구들을 몇 가지 들면 다음과 같습니다:

- **Belkasoft X**: 대표적인 상용 포렌식 소프트웨어로, **SQLite Viewer** 기능이 내장되어 있습니다. 이 도구는 **표준 SQLite API를 사용하지 않고** 파일을 저수준에서 파싱하여, **프리리스트에 남은 삭제 데이터, WAL 및 저널의 미반영 데이터, 페이지 단편(unallocated)까지 모두 검색**합니다 [57] [58]. Belkasoft는 이러한 삭제 레코드를 일반 레코드와 함께 표시하면서, 원본 테이블과 동일한 형식으로 보여주거나 "(deleted)"와 같은 마크를 붙여줍니다 [55]. 또한 아티팩트(carving) 기능으로, **미할당 공간을 대상**으로 채팅 메시지나 URL 등 알려진 패

턴을 찾아내기도 합니다 [59]. **NIST의 테스트 보고서**에서도 Belkasoft가 SQLite 데이터 복구를 정확히 수행함을 확인한 바 있습니다 [60].

- **Cellebrite UFED / Physical Analyzer**: 모바일 포렌식 분야의 표준적인 도구로, Physical Analyzer 소프트웨어가 **SQLite 딥 카빙(deep carving)** 기능을 제공합니다 [61]. 이는 기기에서 추출한 전체 물리 이미지를 대상으로 **SQLite 파일 조각을 탐색**하고, **삭제된 레코드까지 복원**하는 기능입니다. 예를 들어 UFED PA는 메신저 DB에서 삭제된 메시지를 복원하여 보고서에 "deleted"로 표시해줄 수 있습니다. Cellebrite는 문서에서 "Physical Analyzer는 SQLite DB 내부의 삭제 아티팩트를 깊이 있게 찾아낼 수 있다"고 소개하고 있으며 [61], 2022년 NIST CFTT 보고서에서도 해당 기능을 검증하고 있습니다 [62].

- **Magnet AXIOM / IEF**: Magnet Forensics의 도구로, 역시 모바일 앱 데이터베이스를 폭넓게 지원하며 **삭제된 SQLite 레코드 복원** 기능이 있습니다. 예컨대 WhatsApp DB의 삭제 메시지를 복원해주는 것이 알려져 있습니다. (Magnet 공식 문서에도 SQLite 잔존 데이터 처리에 대한 언급이 있습니다.)

- **오픈 소스 도구 및 스크립트**: 무료 도구 중에는 **FQLite (SQLite Forensic Toolkit)** [63], **Undark** 등이 있습니다. FQLite는 2021년 발표된 오픈소스 프로젝트로, **프리리스트와 페이지 내 삭제 레코드를 분석하여 복구**하는 기능을 구현했습니다 [64] [65]. Undark는 비교적 오래된 툴이지만, SQLite DB 파일을 입력하면 삭제된 레코드를 **별도 추출(export)**해주는 스크립트로서 "레코드가 덮어써지지 않은 경우" 효과적이라는 평가가 있습니다 [66]. 이밖에 **PySQLiteParser** 같은 파이썬 스크립트나, Paul Sanderson이 제안한 방법론 등이 존재하며 [67], 필요에 따라 수사관들이 직접 SQLite 파일을 Hex 에디터로 보면서 수동 복구하기도 합니다.

- **DB Browser for SQLite (DB Browser)**: 이 도구는 SQLite 파일을 GUI로 열람/편집하게 해주는 일반 유틸리티입니다. **삭제된 데이터나 WAL 내용을 보여주지 않기 때문에** 포렌식 복구용으로는 부적합하지만, 테이블 구조 확인이나 현재 남아있는 데이터 확인에는 유용합니다 [68] [69]. 다만 앞서 언급했듯, DB Browser를 부주의하게 사용하면 WAL이 적용되어 증거를 변형할 위험이 있으므로, **WAL 파일이 있는 경우 주의**해야 합니다.

- **Autopsy (및 SleuthKit)**: Autopsy는 오픈소스 포렌식 플랫폼으로, 주로 파일 단위 삭제 복구에 강점이 있습니다. SQLite 삭제 레코드 **전문 복구 기능은 Autopsy 자체엔 없지만**, 전체 디스크 이미지를 스캔하여 **삭제된 SQLite 파일 자체를 복구**하거나, 문자열 검색으로 중요한 키워드를 찾아낼 수 있습니다. 예컨대 Android 폰의 physical 이미지에서 Autopsy로 "msgstore.db" 파일을 찾거나, "SQLite format 3"을 키워드 검색하여 과거 DB 조각을 찾는 식입니다. 또한 **최근 활동(웹 브라우징 내역 등)** 모듈이 SQLite에 저장된 기록을 추출해주기도 하지만, 삭제 내역까지는 자동으로 추출하지 못하므로, 이러한 부분은 앞서 언급한 전용 툴이나 수작업으로 보완해야 합니다 [70].

요약하면, **전문 포렌식 도구**들은 이미 SQLite 삭제 데이터 복구 기능을 통합하고 있어서, 수사자는 이를 활용해 **삭제된 메시지, 연락처, 기록 등을 비교적 손쉽게 복원**할 수 있습니다. 반면, 오픈소스 도구나 스크립트를 활용하면 비용 없이도 복구 가능하지만 **수동 작업량**이 많을 수 있습니다. 다음에는 일반적인 **SQLite 삭제 데이터 복구 절차**를 단계별로 정리합니다.

SQLite 삭제 데이터 복구 절차 (매뉴얼 가이드)

다음은 Android 스마트폰에서 추출한 앱의 SQLite 데이터베이스에 대해, 삭제된 컬럼이나 레코드를 복구하는 일반적인 절차를 단계별로 정리한 것입니다:

1. **증거 확보 및 사본 생성**: 대상 장치로부터 **SQLite DB 파일과 관련 파일(WAL, journal)**을 획득합니다. 논리 추출로 파일을 덤프했다면 해당 경로의 `.db`, `.sqlite3`, `.sqlite` 등의 확장자 파일과 함께 같은 폴더의 `-wal`, `-journal` 파일을 모두 복사합니다. 물리 추출 이미지를 입수한 경우, 우선 SleuthKit이나 Autopsy 등을 사용해 관심 있는 DB 파일을 찾아내고 추출합니다. 원본 증거는 **쓰기 방지** 장치를 통해 취급하며, 분석은 반드시 사본에서 진행합니다.

2. **DB 무결성 및 상태 확인**: 사본 DB를 읽기 전용 모드로 열어 **무결성 검사 (**`PRAGMA integrity_check;`**)** 등을 수행합니다. 이때 **WAL 모드 여부**를 확인하기 위해 `PRAGMA journal_mode;`를 조회하거나, WAL 파일 존재 여부를 봅니다. 만약 WAL 파일이 존재하면, SQLite 엔진으로 DB를 열 때 해당 WAL이 자동 적용되므로 주의가 필요합니다. **가장 안전한 방법**은 SQLite를 **읽기 전용**으로 열거나, WAL을 적용하지 않고 별도로 보존하는 것입니다. (일부 포렌식 툴은 WAL을 자동으로 병합하지 않고 따로 보여줌)

3. **현재 데이터 내용 확인 및 스키마 파악**: DB Browser for SQLite와 같은 도구를 사용하여 **현재 남아있는 데이터**를 열람합니다. 이 단계에서는 삭제된 데이터는 보이지 않지만, **테이블 구조(컬럼명, 타입)**와 **남아있는 레코드들**을 살펴봄으로써 삭제된 내용의 종류를 유추할 수 있습니다. 예를 들어 테이블에 `[id, message, sender, deleted_flag]`와 같은 컬럼이 있고 일부 메시지가 삭제되었다면, 남은 레코드들에서 `deleted_flag`가 1인 것과 0인 것을 비교하거나, 연속된 id값의 누락 등을 통해 어떤 레코드들이 삭제되었을지 가늠합니다. **컬럼 삭제의 경우**에는, 해당 테이블의 현재 스키마(SQL 문)를 확인하여 어떤 컬럼이 제거되었는지 파악합니다 (예: 스키마에 존재하지 않지만 다른 버전 문서에 언급된 컬럼 등).

4. **프리리스트 페이지 검사**: 포렌식 도구나 스크립트를 이용해 **프리리스트에 남은 페이지**들을 분석합니다. 예를 들어 Belkasoft X의 SQLite Viewer에서 **"Freelist"** 항목을 열거나, FQLite 툴을 사용하여 프리리스트 페이지를 스캔합니다. 이 과정에서 **삭제된 테이블/레코드의 전체 내용**이 그대로 남아 있는 페이지를 발견할 수 있습니다 [71] [72]. 발견된 페이지들은 해당 테이블의 레코드 구조에 맞게 디코딩하여 표시되며, 이를 통해 (1) 삭제된 **컬럼의 값**이나 (2) **삭제된 행 전체**를 복원할 수 있습니다. 예를 들어, Drop된 컬럼이 `phone_number`였는데 프리리스트 페이지의 레코드를 파싱해보니 여전히 전화번호 문자열이 들어있다면, 그 값을 추출합니다.

5. **페이지 내 삭제 레코드 복원**: 프리리스트 외에도, **할당된 페이지 내의 freeblock/unallocated 영역을 검사**해야 합니다. 이는 자동화 도구의 도움을 받거나, 수동으로 수행할 수 있습니다. 자동화의 경우 Belkasoft X에서 **"Unallocated space"** 탭을 확인하거나 [73], UFED Physical Analyzer의 **SQLite 카빙 기능**을 활용합니다 [61]. 수동으로 한다면, Python 등의 스크립트를 이용해 각 테이블 리프 페이지(헤더 0x0D)에서 **freeblock 연결 리스트**를 따라가며 삭제된 레코드 바이트를 수집합니다 [74]. 앞서 Linux Sleuthing 블로그의 예제 코드처럼, 페이지 헤더의 freeblock 오프셋을 읽고 해당 위치에서 4바이트 헤더를 파싱한 뒤 나머지 바이트를 추출하는 과정을 반복합니다 [18] [75]. 추출한 바이트 배열을 해당 테이블의 레코드 포맷(시리얼 타입 구조)에 맞춰 해석하면 컬럼별 값을 복원할 수 있습니다. 예컨대, 삭제된 SMS 메시지 레코드의 freeblock을 읽어들여 UTF-8 텍스트 시리얼 타입에 해당하는 부분을 문자열로 변환하면 메시지 본문을 얻을 수 있습니다.

6. **WAL 및 저널 분석**: WAL 파일이 있는 경우, 거기서 **메인 DB에 없는 기록**을 찾아냅니다. 이를 위해 WAL을 sqlite3의 `.wal` 모드로 열어 diff를 보거나, 포렌식 툴에서 **WAL 레코드 보기 기능**을 사용합니다 [53]. WAL 분석을 통해 "삭제된 줄의 기존 내용"이나 "수정되기 전 값"을 확인할 수 있습니다. 예를 들어 WAL에만 존재하는 채팅 메시지가 있다면 이는 최근에 추가됐으나 아직 체크포인트되지 않은 데이터일 수 있고, WAL에 기존 본문과 메인 DB에 최종 본문이 다르면 이는 편집되었음을 의미합니다. Rollback Journal(.journal) 파일이 있다면, 이를 바이너리 뷰어로 열어 **"SQLite format 3"` 헤더 이후의 페이지들을 확인**합니다. 저널 내 페이지들이 어떤 테이블의 것인지 파악하려면, 페이지 내용을 해당 테이블 스키마에 맞춰 봐야 하는데, 예를 들어 텍스트 "Alice"가 포함된 페이지가 있다면 사용자 이름 컬럼이었던 것으로 식별하는 식입니다. 일부 포렌식 툴은 저널 파일도 자동 파싱하여 보여주므로 활용합니다 [53].

7. **결과 검증 및 재구성**: 추출된 삭제 데이터 조각들을 **교차 검증**합니다. 예를 들어, 프리리스트에서 복원한 레코드의 RowID가 50이고 그 이전/다음 레코드가 여전히 DB에 있다면, 해당 위치에 실제로 기록이 있었음을 뒷받침합니다. 또한 복원된 컬럼 값들이 논리적으로 일관되는지 (예: timestamp 컬럼 값이 정상 범위, 문자 인코딩이 깨지지 않았는지 등) 확인합니다. 여러 조각을 합쳐야 할 경우 (예: overflow 페이지에 걸친 매우 큰 레코드의 복구) 해당 부분도 처리합니다. 필요한 경우 수작업으로 SQL을 작성해, 복구된 레코드들을 임시 테이블에 INSERT하고 조회하여 **정상 레코드와 비교**하기도 합니다.

8. **보고서 정리**: 최종적으로, 복구된 데이터들을 **식별 가능한 형태로 정리**합니다. 삭제된 레코드는 원래의 테이블 구조에 맞춰 컬럼별로 값을 나열하고, **삭제되었음을 명시**합니다. 만약 컬럼 삭제를 복구한 경우, 해당 컬럼명을

명시하고 복구된 값을 행별로 정리합니다. 가능한 한 표 형태로 정리하여 (예: 메시지ID, 보낸사람, 보낸시각, **<복구된 삭제 메시지 본문>** 같이 컬럼 표시) 제공하면 이해하기 쉽습니다. 또한 **근거자료로서 Hex 오프셋**이나 **페이지 번호**를 제시하여, 특정 복구 데이터가 파일의 어느 위치에서 추출되었는지도 기록해 둡니다. 예를 들어 "메시지ID 123의 삭제된 본문 'Hello'는 DB 파일 페이지 37 (0x25) 오프셋 0x1F4부터 0x1FA에 걸쳐 존재"와 같이 표시하면 증거의 신뢰성을 높일 수 있습니다.

9. **추가 탐색 (선택)**: 필요에 따라, **디스크 이미지의 비할당 영역**을 추가로 검색합니다. 이는 주로 DB 파일이 아예 삭제된 경우나, 내부 잔존이 없을 때 시행합니다. 키워드(예: 전화번호 뒷자리, 특정 단어)나 **SQLite 시그니처**를 기반으로 검색하여 SQLite 조각을 찾고, 이를 수동으로 재조합하는 고난이도의 작업이 될 수 있습니다. 이런 단계는 최후의 수단이며, 일반적인 상황에서는 1~8단계까지만으로 충분한 결과를 얻을 수 있습니다.

이상의 절차를 통해, 안드로이드 앱 SQLite 데이터베이스에서 **DDL로 삭제된 컬럼**이든 **DML로 삭제된 행 데이터**이든, 잔존한 데이터를 최대한 복구할 수 있습니다. 핵심은 **SQLite 내부 구조에 대한 이해**와 **전문 도구의 활용**이며, 무엇보다도 **증거를 안전하게 다루는 것**입니다. 실제 디지털 포렌식에서는 한 DB에서 삭제된 정보가 다른 곳(로그 DB나 캐시 파일 등)에 남아 있는 경우도 많으므로, SQLite 복구 결과를 종합적으로 해석해야 합니다. 하지만 기본 원칙은 본 보고서에서 기술한 바와 같이 **"삭제되어도 어디엔가 흔적은 남는다"**는 것이며, 이를 체계적으로 찾아나가는 것이 중요합니다.

참고 자료: SQLite 공식 문서 [20] [6], Belkasoft 자료 [17] [48], 연구 논문 [19], Linux Sleuthing 블로그 [18] 등. 이들을 통해 SQLite의 삭제 동작 메커니즘과 복구 기법을 상세히 확인할 수 있습니다.

---

1 2 3 30 31 32 33 34 ALTER TABLE
https://www.sqlite.org/lang_altertable.html

4 8 9 10 11 12 13 18 67 74 75 Linux Sleuthing: Recovering Data from Deleted SQLite Records: Redux
https://linuxsleuthing.blogspot.com/2013/09/recovering-data-from-deleted-sqlite.html

5 6 14 15 16 20 21 22 35 Database File Format
https://www.sqlite.org/fileformat.html

7 Structure of a SQLite3 database. | Download Scientific Diagram
https://www.researchgate.net/figure/Structure-of-a-SQLite3-database_fig1_348834288

17 28 29 36 37 38 39 45 46 47 48 49 50 51 52 53 54 55 57 58 59 60 68 69 71 72 73 Forensic data recovery with SQLite analysis in Belkasoft X
https://belkasoft.com/sqlite

19 23 24 25 26 27 40 41 42 43 44 56 64 65 (PDF) Making the Invisible Visible – Techniques for Recovering Deleted SQLite Data Records
https://www.researchgate.net/publication/348834288_Making_the_Invisible_Visible_-_Techniques_for_Recovering_Deleted_SQLite_Data_Records

61 Deep Carving Inside SQLite to Find Deleted Data in the ... - Cellebrite
https://cellebrite.com/en/deep-carving-inside-sqlite-to-find-deleted-data-in-the-database-in-cellebrite-physical-analyzer/

62 [PDF] Test Results for SQLite Data Recovery Tool: Cellebrite Physical ...
https://www.dhs.gov/sites/default/files/2022-03/22_0316_st_TestResults_SQLiteDataRecoveryTool_CellebritePhysicalAnalyzerv747049.pdf

63 FQLite - SQLite Forensic Toolkit
https://www.staff.hs-mittweida.de/~pawlaszc/fqlite/

66 sqlite - How do I undelete accidentally deleted records?
https://stackoverflow.com/questions/454942/how-do-i-undelete-accidentally-deleted-records

70 Problems with Recent Activity (Browsing History) - Autopsy Help
https://sleuthkit.discourse.group/t/problems-with-recent-activity-browsing-history/1603