CHAPTER

06

트랜잭션과 무결성 · 무정지성

이 장에서는 「트랜잭션」에 대해 설명하겠다. 평소 별로 의식하지 않은 사람도 많을지 모르겠지만, 트랜잭션은 데이터 무결성과 무정지성을 요구하는 애플리케이션에서는 빠뜨릴 수 없는 기능이다. 그 본질에 대해 설명하도록 하겠다.

6.1 트랜잭션의 중요성 이해하기

데이터베이스가 갖는 「트랜잭션」이라는 기능은 애플리케이션 개발에 익숙한 사람이라도 「그런 기능 필요 없잖아?」라고 생각하는 경우가 많은 것 같다. 그에 비하여 필자는 일관되게 「트랜잭션은 애플리케이션 개발 및 운영에 필수적인 기능」이라고 생각하고 있다. 이 장에서는 트랜잭션 기능이 없으면 구체적으로 어떻게 곤란한지, 또 어떤 작업들이 늘어나는지에 대한 설명을 하겠다.

어중간한 상태 방지하기

데이터베이스 서버에 있어 한 번의 처리 단위는 UPDATE 문 또는 INSERT 문과 같은 SQL 문의 단위다. 한편, 사용자 측면에서의 처리 단위는 「상품을 산다」 「던전(dungeon)을 클리어한다」라는 「업무 처리」다. 단일 업무 처리가 하나의 SQL 문에서 완결되는 일은 적으며, 대체로 여러 SQL 문을 함께 조합해야만 완결할 수 있다.

「상품을 구입한다」라는 프로세스를 예로 들어 보면,

① 대상 상품 재고 및 가격 등을 조사하기(item 테이블, inventory 테이블)
② 자기 자신이 갖고 있는 제품의 수를 한 개 늘리기(user_item 테이블)
③ 소지하고 있는 돈을 감소시키기(user 테이블)
④ 구매 내역을 갱신하기(purchase_history 테이블)

이와 같은 여러 작업이 필요하다(테이블 이름은 임시로 예를 들었다). 또한 갱신

되는 테이블이 다르기 때문에 각각의 테이블에 대해 SELECT 문, UPDATE 문, INSERT 문 등을 수행하게 된다.

일단 동작하기만 하면 아무런 어려움이 없다. 트랜잭션이라는 잘 모르는 말을 꺼낼 필요도 없을 것이다. 그러나 애플리케이션 개발에서는 「**오류 처리**」를 반드시 고려하지 않으면 안 된다. 즉, 갱신하는 동안 오류가 발생했을 때 어떤 상황이 발생하며 거기에서 어떻게 복구할 수 있는지를 생각할 필요가 있다.

그리고 그 로직이 제대로 동작하는지 테스트해야 한다. 처리 도중에 실패하는 시나리오에는 여러 가지가 있다. 예를 들어 소지하고 있는 돈을 줄인 직후 데이터베이스가 다운될지도 모른다. 또는 소지하고 있는 돈을 줄일 UPDATE 문을 실행 후 애플리케이션이 다운되는 경우도 있을 것이다. 두 경우 모두 그 후의 구입 기록을 갱신하는 쿼리가 실행되는 일은 없다.

이러한 실패 시나리오에서 데이터베이스는 어떤 상태로 있는 것이 바람직할까? 많은 경우는 「업무를 처리하기 이전 상태로 되돌려 달라」고 생각할 것이다. 처리하기 전 상태로 되돌릴 수 있다면 정규 처리를 다시 반복하는 「일반 처리」를 다시 실시하면 되기 때문이다.

갱신 작업 도중에 데이터가 확정된 경우는 여러 가지 곤란한 경우를 겪게 된다. 이력의 갱신이 이뤄지지 않은 정도라면 아직 소생할 방법이 있지만, 「상품을 구매했는데도 소지하고 있던 돈이 줄지 않았다」라고 하면 사용자만이 이득을 얻게 되고 운영 측으로서는 손해를 입게 된다. 이러한 경우는 사용자 측에 이득이 있으므로 큰 문제가 되지 않을지도 모른다[주1]. 그러나 당연히 반대의 경우도 있다. 예를 들어 롤플레잉 게임에서

① 던전을 클리어했다는 플래그를 On시킨다.

② 다음 공략 가능한 던전을 설정한다

이라는 처리가 있었다고 하고, 전반의 갱신이 끝난 시점에서 애플리케이션이 다운되

주1 업무 처리를 단일 트랜잭션에 정리하는 것이 곤란한 경우에는 사용자에게 유리한 처리부터 순서대로 해나가도록 하는 설계 방법이 있다. 도중에 오류가 발생한 경우의 중간 상태가 사용자에게 유리한 것이라면 나중에 문제점으로 되기 어렵기 때문이다.

어 전반부만이 확정되면 다음 던전은 시간이 아무리 지나도 공략할 수 없다.

이러한 어중간한 상태를 특정하여 그것을 복구하는 작업을 애플리케이션에서 작성하는 것은 불가능하지 않다. 그러나 성가신 일이고 코드량도 증가한다. 또한 중간 상태로 끝나게 되는 경우를 재현하기 위한 테스트도 만약을 대비해 정성스럽게 작성해야 한다. 실제로 시간 제약도 있기에 이러한 테스트를 제대로 작성하는 것은 곤란하므로 종종 「사전에 충분히 테스트를 할 수 없기 때문에 운영 후 문제가 발생했을 때마다 대처하겠다」라는 방침을 세우게 된다. 이보다 좀 더 간편하게 접근하는 방법은 없을까?

데이터베이스가 일관성 있는 상태로 자동 복구해 주면 그러한 편이 좋은 것은 당연하다. 이것을 실현해 주는 구조가 「**트랜잭션**」이다. 어디선가 실패하면 어중간한 상태로 갱신이 확정되는 것이 아니라 전부 없었던 것으로 해준다[주2]. 마지막까지 처리를 마치고 결과를 확정시키는 것을 **커밋**(COMMIT), 커밋하지 않고 모든 작업을 원래대로 되돌리는 것을 **롤백**(ROLLBACK)이라고 한다.

SQL 문은 다음과 같이 기술한다.

```
BEGIN; (트랜잭션 시작)
UPDATE …;
UPDATE …;
COMMIT; (커밋)
```

COMMIT 대신에 ROLLBACK을 하거나 COMMIT하기 전에 데이터베이스와 접속을 끊거나 하면 BEGIN 이후의 갱신 처리가 모두 없던 일처럼 된다. 웹 애플리케이션에서는 기본적으로 한 번의 HTTP 요청의 처리에 대해 하나의 트랜잭션을 수행하게 된다.

현대적인 프로그래밍 언어는 예외 처리 메커니즘을 가지고 있기 때문에 「BEGIN 후부터 COMMIT할 때까지의 사이에서 예외가 발생하면 그것을 캐치하여 ROLLBACK한다」라고 기술한다. 데이터로서는 처리 이전 상태이므로 사용자가 작업을 다

주2 트랜잭션이 가지고 있는 「ACID 특성」이라는 네 가지 주요 기능 중 「Atomicity(원시성)」에 해당하는 기능이다.

시 실행하여 그것이 COMMIT되기만 한다면 무사히 갱신이 확정된다.

「도중의 UPDATE 문에서 실패하면 지금까지의 UPDATE 문의 처리를 취소하기 위한 UPDATE 문을 조립하여 ……」라는 처리를 작성할 필요가 없다. 애플리케이션에서 보면 데이터 무결성 주변에서의 에러 핸들링을 크게 없앨 수 있어 간단하게 된다. 이 간결함은 중요하며, 애플리케이션 테스트의 수고를 줄여주는 데 매우 효과적이다.

SQL 문 레벨에서의 롤백

「자기 자신의 애플리케이션에서는 UPDATE 문을 한 번밖에 실행하지 않기 때문에 트랜잭션은 불필요하다」라고 생각하는 사람도 있을지 모르겠다. 그러나 이야기가 그렇게 단순하지는 않다. 한 번의 업데이트성의 SQL 문이라 해도 그 안에는 다양한 작업을 하고 있기 때문이다.

우선 한 번의 업데이트성 SQL 문에서는 여러 레코드를 업데이트할 수 있다. 1만 레코드를 갱신하는 UPDATE 문을 실행하고 있다고 가정하자. 8,000 레코드에서 데이터베이스가 다운되거나 클라이언트가 그 UPDATE 문의 실행을 강제 종료하면 어떻게 될까?

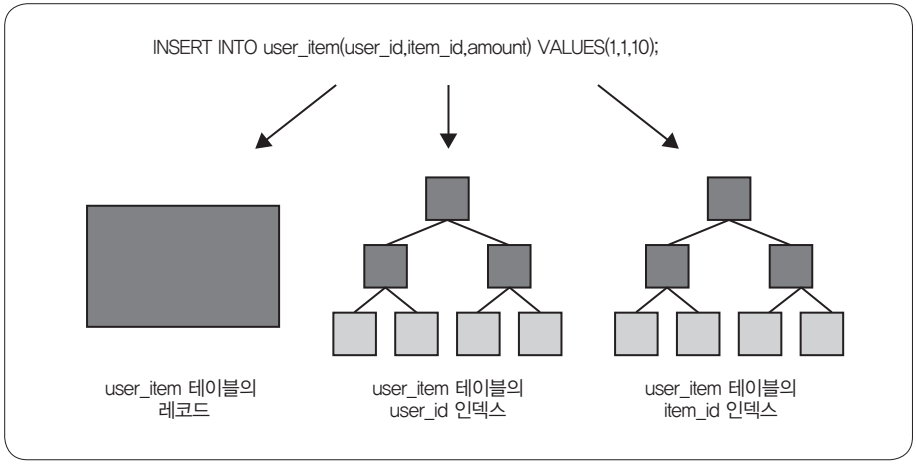
트랜잭션 기능을 갖고 있지 않은 데이터베이스라면 이러한 상황에서 「중간까지만 갱신되는」 결과를 낳게 된다. 예를 들어 8,000 레코드 번째까지 갱신되고 8,001~10,000 레코드 번째까지는 변하지 않는 결과가 된다.

이러한 어중간한 상태로 데이터 처리가 확정되게 되면 이에 대한 복구에는 엄청난 수고가 들어간다. 섣불리 갱신 쿼리를 다시 실행하거나 하면 8,000 레코드 번째까지 두 번의 갱신이 발생하게 되므로 경우에 따라서는 기대와 다른 결과를 초래한다. 이런 상태가 되는 것을 원하는 사람은 거의 없을 것이다.

트랜잭션 기능을 갖고 있는 데이터베이스는 업데이트 자체를 없던 일로 해준다. 어중간하게 8,000 레코드까지 업데이트된 상태로 끝나는 일은 없게 된다. 업데이트 쿼리를 다시 실행하여 10,000번째 레코드까지가 무사히 갱신되면 당초의 예상한 결과가 된다.

단일 SQL 문에서 여러 레코드를 한꺼번에 업데이트하는 경우 일부 레코드만 업데이트된 상태로 멈춰지는 것을 방지하기 위해 트랜잭션 기능이 중요하다는 것을 알 수 있었다. 그렇다면 한 개의 레코드밖에 갱신하지 않으면 문제없는 것일까? 실제로는 그런 경우에도 문제가 있다라는 것이 어려운 부분이다. 인덱스가 있는 테이블에서는 갱신 대상의 열 값만 아니라 갱신 대상의 인덱스도 갱신한다. 이 시점에서 갱신 대상은 데이터와 인덱스 모두가 된다(그림 6-1).

● **그림 6-1** 하나의 레코드밖에 조작하지 않는 SQL 문이라고 해도 내부적으로 다양한 작업을 하고 있다



인덱스가 없는 경우에도 갱신 대상의 열이 하나라고 단정할 수 없다. 여러 열로 되어 있고 각각의 물리적 저장 위치가 다르면 역시 여러 차례의 갱신이 행해지게 되는 것이다. 이러한 내부적인 갱신 처리의 도중에 데이터베이스가 다운되면 어떻게 될까? 이런 경우의 구현은 데이터베이스에서도 크게 의존하는 부분이지만, 트랜잭션 기능을 갖고 있지 않고 구현을 값싸게 한 데이터베이스라면 이러한 문제 상황에서 데이터베이스 자체가 손상될 수도 있다.

한 개의 SQL 문이라고 해도 내부에서는 복수의 작업을 수행한다. 처리 도중에 멈춘 경우에는 데이터가 어중간한 상태가 될지도 모른다. 그러한 상황을 「아무것도 하지 않은 상태」로 복원할 수 있는 것도 트랜잭션 기능 때문이다.

Column

데이터베이스는 적재 적소에서 사용한다

갱신 도중에 다운되면 데이터베이스 자체에 손상을 입을 수 있는 데이터베이스라 해도 실제 시스템에서는 널리 사용되기도 한다. 「간단히 망가지는 데이터베이스라니 현장에서 사용할 수 없는 것이 아닌가」라고 생각하는 사람도 많을 것이다.

물론 단일 DB 서버로 사용한다면 그런 상황이 무서워서 실전 투입은 어려울 수 있다. 그래서 이러한 데이터베이스를 사용하는 경우 복제 구성과 함께 세트로 사용한다. 만약 하나의 서버가 망가져도 남은 서버에서 운용할 수 있도록 하는 것이다. 연달아 모든 서버가 동시에 망가지게 되면 당연히 아무 소용이 없지만, 그러한 가능성은 거의 없다고 생각되므로 그냥 이렇게 운용하는 경우도 있다.

무정지성 확보하기

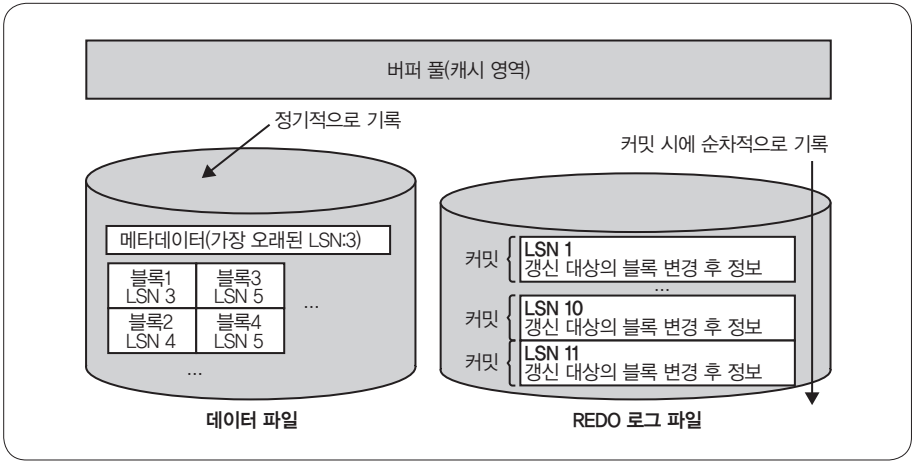
트랜잭션 기능의 대표적인 두 번째 이점은 **무정지성의 향상**이다. 즉, OS 장애 등의 서버 장애가 발생하여 그로부터 데이터베이스를 재기동한 때에 「장애 직전까지의 커밋 결과를 손실하지 않고 마치는 것」이 가능하다.

트랜잭션을 지원하지 않는 데이터베이스의 경우는 OS 장애뿐만 아니라 데이터베이스 프로세스가 비정상적으로 종료하기만 해도 데이터베이스가 손상될 수 있다. 깨진 데이터베이스를 복구하려고 해도 일부 데이터가 없어지거나 완전히 복구되지 않는 등 완전한 회복이 불가능한 경우도 많다.

Oracle과 InnoDB 같은 트랜잭션 대응의 데이터베이스에서는 이러한 문제가 발생하지 않는다. 기술적으로는 그림 6-2와 같은 「REDO 로그[주3]」를 이용한 아키텍처로 무정지성을 보장하고 있다. 여기에서는 InnoDB를 예를 들어 그 구조를 간단하게 설명하고자 한다.

주3 데이터베이스에서 수행한 작업을 다시 실행하는 로그다.

● **그림 6-2** 무정지성을 담보하는 메커니즘



REDO 로그의 역할

InnoDB에서는 트랜잭션을 커밋하면 그때마다 LSN(Log Sequence Number)이라는 시퀀스 번호가 증가하고, 그 번호와 갱신 대상의 블록의 정보(갱신할 곳의 블록 ID 및 갱신할 곳의 위치 및 값)를 REDO 로그 파일에 쓴다.

한편, 열 값과 인덱스를 갖는 데이터 파일에는 커밋을 할 때마다 기록을 하는 것이 아니라 캐시 영역에 보관해 두고 정기적으로 디스크에 기록하는 동작을 한다[주4]. 따라서 REDO 로그 파일이 최신 커밋 정보를 가지고 있는 반면, 본체의 데이터 파일은 오래된 데이터를 가지고 있게 된다. 그래도 캐시 영역에 최신 데이터가 있기 때문에 데이터 파일에 이전 데이터밖에 없더라도 애플리케이션에서 보면 최신 데이터를 읽고 쓸 수 있어 모순된 상태가 되지는 않는다.

서버 장애 등의 이유로 데이터베이스가 멈춰서 재기동을 하게 되는 경우가 있다. 이 경우 캐시 영역의 데이터는 없어졌기 때문에 데이터 파일과 REDO 로그에 기록되어 있는 데이터가 올바르다. 그러나 데이터 파일은 이전 데이터밖에 남지 않았기

주4 이렇게 정기적으로 기록하는 처리를 체크 포인트라고 부른다.

때문에 그 상태 그대로 데이터의 파일 읽기 및 쓰기를 하면 최신 데이터를 읽지 못하고 모순된 결과가 되어 버린다.

그래서 이러한 상황에서는 REDO 로그의 내용을 데이터 파일에 적용시켜 나감으로써 데이터 파일과 REDO 로그의 LSN과 일치시키는 작업을 수행한다. 이 과정을 「충돌 복구(Clash Recovery)」라고 한다. 복구 시에는 적용을 시작할 가장 오래된 LSN을 특정한 뒤 그 위치에서부터 REDO 로그의 내용을 순서대로 대응해 나가면 된다. 이로 인하여 장애 이전 상태로 복구할 수 있다.

이중 기록의 비용

데이터 파일과 REDO 로그 파일의 두 종류의 파일을 갖는다는 방식은 트랜잭션을 지원하는 데이터베이스의 표준적인 아키텍처다. 이 방식에서는 1회 갱신에 대한 그 자체의 갱신 정보를 REDO 로그에 동기적으로 기록하면서 나중에 데이터 파일에도 기록하는 식의 두 번의 기록이 발생하게 된다. 기록량이 두 배가 되면 두 배 늦어 버리는 것은 아닐까라고 생각할지도 모르겠다.

그러나 실제로는 그렇게까지 느려지지는 않는다. 그것은 REDO 로그 파일이 순차적 기록을 하는 특징이 있기 때문이다. 레코드나 인덱스로의 기록은 랜덤 기록(Random Write)이다. 구체적으로는 우선 기록 대상의 영역(블록)을 메모리로 읽어들이고 그것을 갱신한 후 디스크에 기록하는 흐름이다. 대상 블록이 메모리에 없으면 디스크에서 읽어 온다라는 식의 처리(랜덤 읽기)도 발생한다.

이것은 HDD에서는 매우 느리다는 문제가 있다. 한편, 시퀀셜 기록(Sequential Write)은 HDD에서 매우 빠르다. 이 때문에 REDO 로그 파일로의 기록량이 증가하더라도 그 추가 비용은 문제가 되지 않는다.

단, SSD(Solid State Drive, 플래시 드라이브)에 의한 고속화가 진행되어 가게 되면서 점차 무시할 수 없게 되었다. 이에 대해서는 다음 장에서 설명하겠다.

Column

NoSQL 및 트랜잭션

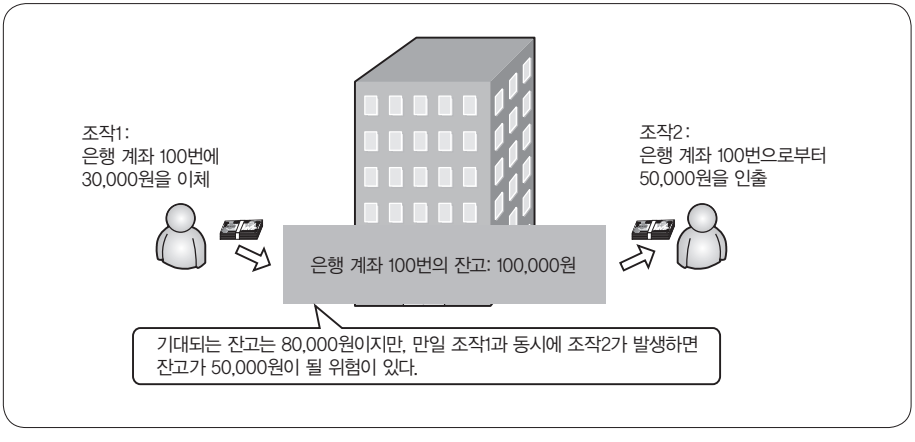
본문에서 언급했듯이 트랜잭션은 데이터베이스가 갖고 있는 기능으로, 이를 이용함으로써 데이터가 어중간한 상태가 되거나 장애가 발생하면 데이터가 손상되거나 하는 비정상계의 약점을 극복할 수 있다. 주요 RDBMS의 경우 모두 트랜잭션 기능이 있다.

최근 들어 「NoSQL」이라는 데이터 저장소가 주목을 끌게 되었다. RDBMS와의 사이에서 우열을 언급하는 경우도 많지만, NoSQL의 대부분은 트랜잭션 기능을 갖고 있지 않은 것도 중요한 포인트가 된다. NoSQL을 사용하면서 비정상계의 처리에 대응하기 위해서는 그에 상응하는 애플리케이션 로직을 만들어 내지 않으면 안 된다.

6.2 잠금 메커니즘에 의한 배타 제어

다수의 사용자가 이용하는 데이터베이스에서는 동일한 레코드에 대해서 동시에 액세스가 발생할 가능성이 있다. 자주 소개되는 예로 「은행 계좌」가 있다. 예를 들어 어떤 가족이 잔액 10만원의 계좌를 갖고 있었다고 하자. 남편이 3만원을 입금하고 아내가 5만원을 인출하면 그날 계좌의 잔고는 8만원이 될 것이다. 데이터베이스에서는 이러한 동시 액세스가 발생했을 경우에 결과가 모순된 상태가 되지 않도록 배타 제어를 해야 한다. 만약 배타 제어가 대충 이루어진다면 잔고가 5만원이 되거나 13만원이 될 가능성이 있다(그림 6-3).

● **그림 6-3** 배타 제어를 하지 않으면 일관성이 망가진다



이것을 막기 위해 정평이 나 있는 메커니즘이 바로 「잠금(Lock)」이다. 먼저 갱신할 것이 잠금을 확보하고, 다른 트랜잭션에서의 동일한 레코드에 대한 갱신을 방지하는 동작을 한다. 잠금 메커니즘은 MySQL의 MyISAM과 같이 트랜잭션을 지원하지 않는 것에서도 구현되어 있지만, "잠금의 범위", "잠금의 기간"에 대한 측면에서 제약을 받는 경우가 많으며, 트랜잭션을 지원하는 Oracle과 InnoDB 쪽이 사용하기에 편리하다.

잠금의 범위

MyISAM의 경우는 대상 테이블을 갱신하기 전에 해당 테이블에 대해 배타적 잠금을 건다. 그리고 갱신을 마치게 되면 잠금을 해제한다. 잠금의 범위는 테이블이 된다.

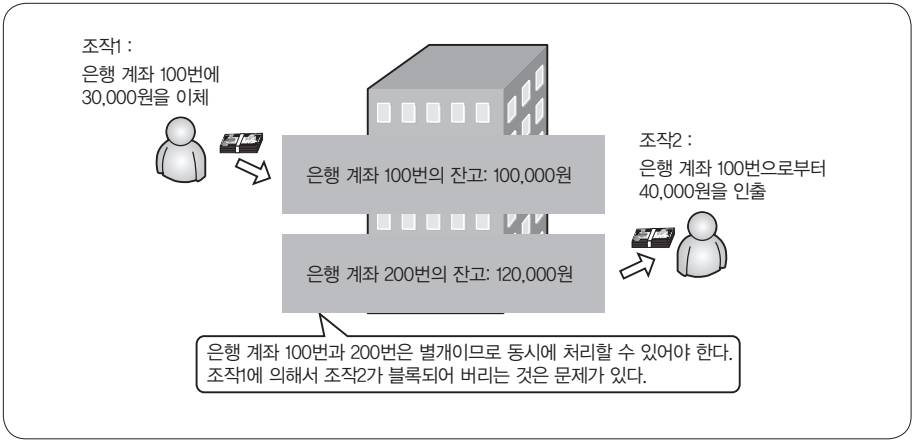
한편, InnoDB 등 현대적인 RDBMS에서 확보하고 있는 잠금의 범위는 레코드다. 테이블 전체 등 넓은 범위에서 잠금을 걸어 버리면 관계없는 레코드에 대해서도 갱신으로 인해 블록이 되어 버려 병렬성이 크게 저하되기 때문이다(그림 6-4). MyISAM의 경우는 잠금이 테이블 단위다. 잠금을 하는 동안 같은 테이블에 대한 읽기 및 쓰기 처리가 그와 관계없는 레코드라 해도 블록되어 버린다. 따라서 병렬성이 저하된다. 또한 Oracle이나 InnoDB의 경우 기록을 하고 있는 동안에도 커밋 완료된 데이터를 읽을 수 있다는 특징이 있어 병렬성 향상에 크게 공헌하고 있다.

잠금 기간

잠금은 트랜잭션의 종료(커밋 또는 롤백) 시까지 유지한다. 이것은 커밋 전에 잠금을 해제하면 나중에 롤백할 수 없는 위험이 있기 때문이다.

예를 들어 「상품을 구입하기」라는 처리를 생각해 보자. 이것은 상품의 재고를 줄이고 자신이 소지하고 있는 돈을 줄이는 처리로 각각 별도의 UPDATE 문이 되지만, 「상품의 재고가 제로인 경우와 소지하고 있는 돈이 부족한 경우에는 오류로 처리하도록」 하자.

● **그림 6-4** 배타 제어의 범위가 넓으면 병렬성이 떨어진다



다음과 같은 처리를 생각해 보자.

T1: 상품 A의 주식 수를 1 감소(2 → 1)

T2: 상품 A의 주식 수를 1 감소(1 → 0)

T1: 소지하고 있는 돈을 감소시킨다(부족: 오류)

만약 T2가 블록되지 않는다면 재고 수는 0이 된다. 그 후 T1은 소지하고 있는 돈이 부족하기 때문에 오류가 발생하는데, 이때 롤백을 하려면 재고 수량을 얼마로 되돌리면 좋은가? 2로 되돌리면 T2의 갱신 결과(1 줄어듦)가 반영되지 않는다. 상품 A의 재고가 잠겨 있지 않으면 원래대로 되돌리는 것은 어렵다.

잠금 메커니즘의 단점은 동일한 레코드에 대한 갱신이 동시에 한 개의 클라이언트밖에 할 수 없다는 점에 있다. 여러 클라이언트에서 갱신할 수 있도록 잠금을 걸지 않고 배타 제어를 실시하는 「락 프리」라는 알고리즘도 주목을 끌고 있다. 이 경우 위에서 언급한 바와 같이 충돌이 발생하더라도 일관성 있는 상태로 복원할 수 있도록 구현해야 할 필요가 있어 구현의 복잡성이 크게 향상된다.

또한 MyISAM은 트랜잭션을 지원하지 않기 때문에 일반적으로 SQL 문의 실행이 끝난 시점에서 잠금이 해제된다. 이 경우 여러 SQL 문에 걸쳐 잠금을 확보하려 하면 곤란을 겪게 된다.

MySQL에서는 이러한 용도를 위해 「LOCK(UNLOCK)TABLES」이라는 전용의 SQL 문을 준비하고 있다. 일련의 처리를 시작하기 전에 LOCK TABLES을 하여 모든 것이 끝난 뒤에 UNLOCK TABLES을 하면 된다는 것이다. 그러나 추가로 이러한 SQL 문들을 실행할 필요가 있기 때문에 수고가 늘어난다. InnoDB처럼 트랜잭션을 지원하는 것은 이러한 수고가 불필요하고 데이터베이스가 전부 대신해 주기 때문에 간단하다.

데이터베이스를 사용하지 않을 경우는 이러한 배타 제어를 프로그래밍으로 해야 한다. 여러 프로세스/스레드에서 파일을 읽고 쓸 수 있도록 프로그램을 작성하는 경우, 파일을 갱신하기 전에 잠금 파일이나 mutex 등의 글로벌 잠금을 얻어 다른 작업을 블록하곤 한다. 데이터베이스의 잠금 메커니즘을 사용하면 이런 귀찮은 기술이 필요하지 않으므로 구현의 수고를 크게 줄일 수 있다.

6.3 복제 및 트랜잭션

제5장에서 설명한 대로 복제 구성(그림 5-1)을 만들 경우 트랜잭션 기능은 더욱 중요성을 갖게 된다. 트랜잭션 설명의 마지막 주제로서 조금 어려운 이야기가 되겠지만 이 주제에 대해서 설명해 나가고자 한다.

「원자성을 갖는 복제」의 중요성

복제 구성에서 있어서 슬레이브에서는 「마스터에서 전송되어 온 업데이트성 쿼리를 실행하는」 역할을 한다. 이 처리는 사실 업데이트성 쿼리를 실행하는 것만으로는 충분하지 않고 「슬레이브는 마스터에서 보낸 업데이트성 쿼리 중에서 어디까지를 실행했는지」라는 정보도 관리할 필요가 있다. 그렇게 처리하지 않으면 도중에 슬레이브가 멈추었을 경우 어디서부터 다시 시작하면 좋은지를 알 수 없기 때문이다. 엄밀하게는 이 정보들은 원자성 있게[주5] (동일 트랜잭션에서) 갱신해야 한다.

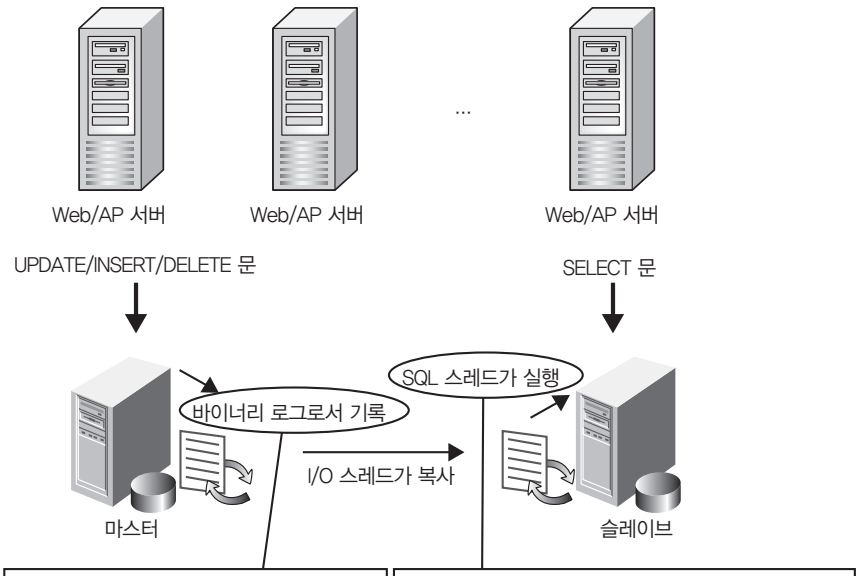
MySQL에서는 MySQL 5.6과 InnoDB의 경우 처음으로 원자성 있는 갱신이 가능하게 되었다. 그 이미지를 그림 6-5에 나타내고 있다.

그림 6-5에서 마스터는 트랜잭션이 커밋될 때마다 그 갱신 SQL 문을 바이너리 로그에 기록한다. 슬레이브에서는 바이너리 로그의 내용을 실행하여 나가지만, 「현재 어디까지 실행했는지」를 관리하기 위해 「실행을 마친 바이너리 로그의 위치 정보」를

주5 더는 나눌 수 없다는 뜻이다.

관리하는 InnoDB 테이블을 제공한다. 그리고 갱신 SQL 문의 실행과 위치 정보 갱신을 동일한 트랜잭션에서 실시한다.

● **그림 6-5** MySQL의 복제의 진화형



복제를 어디까지 진행했는지를 전용 테이블(이 경우는 slave_relay_log_info)에 보관한다.

만약 갱신 작업 도중에 슬레이브가 크래쉬해도 재기동하면 마지막 커밋 시점의 상태를 복원할 수 있다. 그때 테이블에 기록되어 있는 위치로부터 복제를 다시 시작하면 그 이후의 갱신을 계속할 수 있게 되는 것이다.

애플리케이션용의 테이블의 갱신 처리와 복제 상태 관리 테이블로의 갱신 처리가 원자성을 갖고 있지 않은 경우는 크래쉬의 상황에 따라 쿼리 10까지 갱신했지만 복제 상태로서는 9라는 어긋난 상태가 발생될 수도 있다. 그대로 복제를 재개하면 다시 한 번 업데이트성 쿼리 10을 실행하려고 할 것이다. 하지만 10은 이미 커밋되어 있는 상태이므로 10을 두 번 실행하게 된다. 돈을 100원 감소하는 처리라면 200원 줄어드는 형국이다. 이것은 곤란하다.

따라서 애플리케이션의 관점에서 모든 업데이트성 쿼리가 UPDATE 문 한 번밖에 없는 자동 커밋 쿼리라 할지라도 복제 구성과 조합한 시점에서 업데이트성 쿼리의 실행에 더하여 복제 상태 업데이트 처리도 모두 한 개의 트랜잭션으로 하지 않으면 안 되는 것을 알 수 있다. 이것들이 하나의 트랜잭션으로 되어 있으면 어느 타이밍에 크래쉬해도 갱신 확정된 쿼리를 또다시 복제하거나, 반대로 복제를 날려 버리는 일은 없게 될 것이다.

사용자는 원자성이 있는 복제에 어떻게 대처하고 있는가?

MySQL 5.5까지는 이 같은 기능이 없었는데 웹 서비스를 운용하고 있는 사용자들은 이러한 문제에 어떻게 대처하고 있을까? 사실 많은 사용자들이 「이 문제를 알지 못한다」. 의아하게 생각될지도 모르겠지만 데이터베이스, 그것도 트랜잭션 및 복제를 조합한 것 같은 고난이도의 내용이 되면 머릿속이 하얗게 되는 사람이 많은 것 같다.

물론 문제를 잘 인식하는 사용자도 있다. 현실적으로는 MySQL과 같은 검증된 데이터베이스는 쉽게 다운되지 않기 때문에 다운되었을 때는 슬레이브를 재구축하면 된다는 식으로 결론 짓고 있는 곳도 많다.

현재의 데이터 규모라면 그걸로도 좋겠지만, 결국 하드웨어의 고성능화에 따라 데이터 양이 방대해진다면 순순히 제로부터의 재구축을 할 수 없게 되므로, 여기서 언급한 것같이 트랜잭션으로서의 일관성을 갖게 하여 장애 직전부터의 복구를 하는 솔루션이 필요하게 된다.

여기서 언급한 MySQL 5.6(InnoDB) 기능은 슬레이브가 다운되었을 때 데이터베이스의 전체 복구를 해야 하는 제품과 비교하면 큰 장점이 있다.

동일한 것이 MySQL Cluster(NDB 스토리지 엔진)에서는 사실 오래 전에 구현되어 있다. NDB의 경우에는 사용자층이 주로 엔터프라이즈 고객이므로 이 부분의 요구가 엄격한 측면도 있어서 예전부터 지원하고 있었다.

슬레이브가 다운된 경우에는 슬레이브를 재구성하면 된다고 말했다. 물론 복제의 위치 정보의 갱신이 트랜잭션에 대응하면 슬레이브 재구축이 아닌 차등 복구를 할 수 있게 된다.

그러나 많은 사용자들이 이러한 문제점을 제대로 인식하고 있지 않아 트랜잭션을 지원하지 않는 복제를 사용하는 환경임에도 불구하고 다운된 슬레이브를 그대로 재기동하는 경우가 상당히 많이 있는 듯하다. 이는 갱신된 정보와의 동기화가 이루어졌다는 보장이 없기 때문에 한 번 실행한 쿼리를 재실행하거나 할 위험성이 높다. 이는 결과적으로 불일치를 초래할 위험이 있다.

그런데 이 불일치가 일어나고 있는지 어떤지는 도대체 어떻게 하면 알아낼 수 있을까? 원래부터 불일치를 인지하지 못했다면 "재기동하는 것만으로 자동 복구했다! 굉장하다!"라는 엉뚱한 말을 할지도 모르겠다.

불일치의 검출은 전통적으로 테이블의 전부 또는 일부를 스캔하여 체크섬(checksum)을 계산하거나 하는 등의 형태로 이루어진다. MySQL에서는 이러한 툴이 존재한다. 이때 참조 처리가 무거워지므로 갱신을 블록하지 않도록 하는 것이 중요하지만, InnoDB라면 잠금을 하지 않고 참조 가능하므로 현실적으로도 실행 가능하다.

CHAPTER 6 요약

이 장에서는 트랜잭션 메커니즘에 대해 설명하였다. 원자성이 있는 처리를 할 수 있다는 것은 애플리케이션의 오류 체크의 수고를 방지하는 데 매우 유용하다. 또한 무정지성 메커니즘은 복구 노력을 줄이는 데도 도움이 된다. 트랜잭션은 현장에서 많이 사용되고 있는 RDBMS와 빈약한 파일 메커니즘 또는 경량 데이터베이스와의 차이를 말하는 데 있어 중요한 기술이다. 최근에는 복제 메커니즘을 많이 사용하는데, 그러한 경우에도 도움이 되는 기술이라는 것을 알게 되었을 것이다.

다음의 제7장에서는 데이터베이스의 성능의 지배 요인이 되는 「스토리지 기술」의 변천에 대해 설명하겠다.