CHAPTER

01

데이터베이스가 없으면 무엇이 곤란한가?

「데이터베이스 기술」은 세상의 모든 서비스 가동에 필수라고도 말할 수 있는 중요한 기술이다. 그러나 접근하기 어려움과 난해함, 심오함 등으로 인해 제대로 이해해서 사용하고 있는 사람은 적지 않을까 싶다.

이 장에서는 데이터베이스의 중요성과 어떠한 사람이 이 책을 읽어야 하는지 등에 대해 설명해나갈 것이다.

1.1 기술자로서 요구되는 스킬

IT 업계에서 일하는 사람이나 그것을 목표로 하는 학생들에게 있어 반드시 몸에 익혀 두고 싶은 기술에는 어떤 것이 있을까? 많은 사람들이 제일 먼저 예를 드는 것은 「프로그래밍 언어」에 관한 기술이 아닐까 싶다. Perl/PHP/Python과 같은 「LL 언어(Lightweight Language, 경량 언어)」라 불리는 접근하기 쉬운 언어와 Java와 C#과 같은 고수준/고기능의 언어를 사용할 수 있다면 간단한 프로그램을 작성하는 것은 그다지 어렵지 않다.

컴퓨터와 OS가 어떻게 움직이고 있는가라는 본질적인 부분의 이해를 위해 C 언어와 어셈블러의 습득이 필요하다고 주장하는 사람도 많을 것이다. 고급 언어가 아니라면 한정된 시간 안에 빨리 고품질의 소프트웨어를 만드는 것은 어려울 것이다. 반면에 아래 계층인 메모리의 동작 원리 등 깊은 수준의 지식이 없으면 만일의 경우에 발생할지 모르는 트러블을 해결할 수 없다. 이들 모두 중요한 테마다. 어떤 프로그래밍 언어로 시작한다 해도 최근에는 설치 프로그램에서부터 단번에 개발환경을 설치할 수 있도록 되어 있고, Linux 배포판에 따라서는 처음부터 환경이 정비되어 있는 것마저 있으므로 학습 환경 자체는 풍족한 혜택 속에 있다고 말할 수 있다.

프로그래밍 언어의 지식이 있다 해도 좋은 소프트웨어를 만들 수 있는 것은 아니다. 예전부터 「뛰어난 소프트웨어는 무엇인가」라는 논쟁도 있지만, 상용 소프트웨어라면 「매출」이 중요한 지표가 될 것이다. 매출 그 자체는 「해당 소프트웨어로 무엇을 할 수 있느냐」나 「어떠한 판매 전략을 전개하는가」라는 기술적인 면 이외의 요소에 크게 의존한다.

상용 소프트웨어뿐만 아니라 게임이나 영화도 그렇지만, 기술적으로 최첨단인지

아닌지는 잘 팔릴 것인가 아닌가에 대한 요소와 반드시 결부되어 있는 것은 아니다. 좋은 상품 컨셉을 갖고 있는 것이 중요하다. 기술자적인 관점에서는 제품이 매력적으로 느낄 수 있도록 계획대로 신속하고 확실하게 구현해 나가는 기술이 중요하다.

소프트웨어 개발 현장에서는 일단 결정된 사양을 나중에 변경하거나 세부 사양을 나중에 추가하는 일이 빈번히 있다. 당초의 기획에서 전혀 변경되지 않는 것이 오히려 드물고, 더 좋은(혹은 더 잘 팔리는) 것을 만들어 가기 위해 지속적으로 개선을 거듭해 나가는 것이 오히려 당연하다. 기술자로서 그러한 사양 변경에 신속하게 대응할 수 있는 설계 기술 또한 중요하다.

또한 업무용 소프트웨어라면 릴리스 이후에 사양을 변경하여 새로운 버전으로 내놓는 경우가 많지만, 지금까지 사용하고 있던 사람들에게 악영향을 미치지 않도록 기능 및 성능 면에서 악영향이 없는지 확인하는 테스트[주1]를 정성스럽게 하는 것도 매우 중요하다. 이러한 설계 방법 및 개발 프로세스를 **소프트웨어 공학**이라 총칭한다. 설계 방법은 구조적 프로그래밍과 객체지향 기술이 유명하고, 개발 프로세스로서는 가장 기초적이라 할 수 있는 「폭포수 모델(waterfall model)」과 보다 기동성이 높은 「애자일/XP」 등 다양한 것들이 있다.

소프트웨어 공학에 대한 이해 또한 프로그래밍과 마찬가지로 매우 중요하며, 기술자로서 한번은 제대로 배워 두고 싶은 영역이다.

폭포수 모델 등 과거의 개발 프로세스는 최근에는 경시되는 풍조인데, 그 후에 나온 개발 프로세스가 앞서 언급한 사양 변경의 유연한 대처와 기능 및 성능 면에서의 신속한 검증이라는 과제를 해결하는 데에 주된 목적을 두고 있다는 점을 생각해보면, 왜 문제가 되었는지를 포함해서 제대로 이해하는 것이 중요하다고 생각한다.

데이터베이스 기술의 중요성

이렇듯 어떠한 제품을 만들기 위한 기술의 중요성은 자주 거론된다. 한편, 그 중에서 우리 IT 업계에서 일하는 사람들에게 빠뜨릴 수 없는 것이 데이터베이스에 관한

주1 일반적으로는 회기 테스트(regression test)라고 불리고 있다.

기술일 것이다. 동작하는 데 적지 않은 「데이터」를 필요로 하는 곳에서는 예외 없이 데이터베이스가 중요한 역할을 담당한다. 이 데이터베이스의 사용이 적절하지 않으면 나중에 성능 문제가 발생하거나, 데이터가 잘못된 (일관성 또는 정합성이 없는) 상태가 되거나, 데이터를 손실하는 등 심각한 문제를 야기시킬 가능성이 있다.

필자가 IT 업계에 들어선 2~3년차의 사람들과 이야기를 해본 결과, "이 업계에 들어서서 데이터베이스가 매우 중요한 것이라는 것을 처음 알았다"라는 사람이 많은 것에 다소 놀랐다. 이 업계에 들어오는 사람의 대부분은 대학생이나 전문대학생 시절의 교육 과정을 통해 다소나마 프로그래밍 언어에 대한 경험을 하게 된다. 그 중에는 초등학생 때부터 프로그래밍을 시작했었다는 대단한 사람도 있었지만, 왠지 모르게 데이터베이스 기술을 알고 있는 젊은 기술자는 거의 눈에 보이지 않았다.

프로그래밍 언어의 경우, 간단한 샘플을 만들 수 있어 무엇을 할 수 있고 무엇을 할 수 없는지 바로 알 수 있다. 따라서 접근하기 쉽고, 대학의 수업 등에서도 가르치기 쉬웠을 것이다. 한편, 데이터베이스의 경우에는 그것만으로는 제대로 동작하는 소프트웨어를 만들 수 없다. 간단한 예제라는 것은 존재하지도 않을 뿐더러 데이터베이스를 사용하지 않는 고기능의 소프트웨어들도 많이 있다(특히 게임 소프트웨어가 그렇다).

또한 그래픽과 같이 눈에 보이는 외견상의 UI 도 갖고 있지 않기 때문에 실제로 어떠한 일들이 벌어지고 있는지 알기 어렵다. 애플리케이션 개발 실무 경험이 없으면 데이터베이스의 중요성은 잘 모를 것이다. 대학의 컴퓨터 관련학과에서는 「데이터베이스 이론」이라는 과목이 분명 다루어지고 있지만, 그 필요성을 종합적으로 가르치는 곳은 적을 거라 생각한다. 필자도 대학에서 데이터베이스 이론을 공부했지만, 도대체 무엇 때문에 필요한지 알 수 없어서 대부분 그냥 넘겨버렸던 기억이 있다.

그러나 현실적으로 세상에서 사용되고 있는 웹 애플리케이션의 대부분은 데이터베이스 없이 서비스를 제공할 수 있는 단순한 것이 거의 없기 때문에 애플리케이션을 개발하는 과정에서 처음으로 데이터베이스 기술의 중요성을 절감하게 된다. 이 중요성을 모르는 상태에서 애플리케이션 개발을 하게 되면 적어도 한 번은 실패를 경험하게 되고, 그로 인해 데이터베이스의 중요성을 재고할 수 있을 것이다. 다행히도 주위에 실력이 좋은 데이터베이스 기술자가 있다면 그의 도움을 통해 자신의 실수를 외부에 알리지 않고 문제를 해결하여 자신의 지식으로 축적할 수 있을지도 모르겠다.

한편, 운이 나쁜 경우에는 지식의 부족으로 인해 시스템 장애를 일으킬지도 모른다. 발생한 문제에 대한 해결 방법은 알고 있어도 본질적인 부분을 이해하고 있지 않으면 다음에 새로운 문제가 발생했을 때 대처할 수 있을지 장담할 수 없다.

이 책의 대상 독자

이 책은 데이터베이스 기술 동향을 정리하고, 일상 업무에서 응용할 수 있는 필수적인 부분의 이해와 이를 토대로 향후 동향도 예측해 나가는 것을 목표로 한다. 주요 대상 독자로서 다음과 같은 사람들을 생각하고 있다.

데이터베이스의 필요성을 모르는 사람

"Excel로는 왜 충분하지 않지?"라고 질문을 받았을 때 대답하기 곤란한 사람, 선배 사원으로부터 "데이터베이스 기술은 중요하기 때문에 익혀 두어야 해!"라는 소릴 들었지만 왜 중요한 것인지 모르는 사람, 혹은 관심은 있지만 어디서부터 공부하면 좋은지를 모르는 사람 등이다. 실제 업무와 관련되지 않는 한 데이터베이스 기술의 중요성을 체감적으로 이해할 수 없는 것은 어쩔 수 없다. 필자 자신도 그랬다. 그런 사람들이야말로 이 책을 읽어 두길 바란다.

데이터베이스 관련 지식을 정리하고 싶은 사람이나 전체 모습을 파악하고 싶은 사람

데이터베이스를 둘러싼 테마는 많이 존재한다. 최근 화제가 되고 있는 「NoSQL vs SQL」이라는 주제도 있고, 정규화 이론 같은 테이블 설계 이론, 인덱스 디자인과 같은 고속화에 대한 이야기, 트랜잭션에 관한 이야기, 장애 대응에 대한 이야기, 중복에 관한 이야기, 분산 데이터베이스 이야기, 데이터웨어 하우스 이야기, 전체 텍스트 검색에 관한 이야기 등 실로 다양하다. 이들은 서로 독립적인 이야기가 아니라 내부적으로 서로 연결되어 있다.

새로운 기술 키워드가 나올 때 전혀 새로운 것이 등장하는 것으로 여기는 사람이

있을지도 모르겠다. 그러나 새로운 기술 요소라는 것은 기존의 기술 과제를 넘어서는 것을 주된 목적으로 태어나는 것이다. 즉, 기존 기술의 특징과 과제를 잘 이해해 두면 새로운 기술이 나올 때 그 의미를 제대로 이해할 수 있고, 그 기술이 본질적으로 유효한 것(장기간 활용할 수 있는 것)인지 일시적인 열풍에 지나지 않는 것인지도 판단할 수 있을 것이다. 그리고 앞으로 어떤 기술이 주목을 모을지도 예상할 수 있을 것이다. 야심이 있는 사람이라면 이러한 트렌드를 예측하여 스스로 새로운 제품을 내놓을 수도 있을지 모르겠다.

체계적으로 지식을 정리하고 전체적으로 바라볼 수 있도록 하는 것은 미래를 예측하는 힘을 기르는 연결점이 된다. 이 책은 이러한 「데이터베이스 기술의 나침반」이 될 것을 목표로 집필되었다.

1.2 데이터베이스가 없으면 무엇이 곤란한가?

Mobage나 Yahoo!와 같은 대규모 웹사이트는 물론 사내 애플리케이션과 같은 비교적 간단한 서비스에서도 데이터베이스는 필수라고 말해도 좋을 정도로 사용되고 있다. 데이터베이스에 액세스하려면 단순히 텍스트 파일을 읽고 쓰는 것에 비해 인증 프로세스와 SQL 문의 작성 처리, 결과 셋의 검색 처리 등의 특수 처리가 필요하며, 그 나름의 프로그래밍 지식이 필요하다.

왜 이런 번거로운 일을 하면서까지 데이터베이스를 사용하지 않으면 안 되는 것일까? 익숙한 사람에게는 자명한 이치일지도 모르지만, 이 점을 명확하게 해두지 않으면 "모두가 사용하고 있기 때문에 나도 사용한다"와 같이 겉보기에는 안전한 방법으로 보이지만 이는 단순히 유행을 좇는 것과 다를 바가 없게 된다. 단순히 유행을 따르는 것이 왜 문제인가 하면, 새로운 기술이 등장했을 때 그것을 제대로 평가 또는 확인할 수 없다는 점을 들 수 있다. 즉, 누군가가 "○○○는 대단하다"고 발언했을 때 왜 대단한지, 어떠한 점에서 우수하고 어떤 부분에서 뒤떨어지는지에 대한 평가를 할 수 없다는 것이다.

모든 기술에는 적성이 존재한다. 적합하지 않은 분야에서 사용했기 때문에 향후 장애에 시달리게 되었다는 사례는 얼마든지 있기 때문에 주의가 필요하다.

이 절에서는 데이터베이스가 없으면 어떠한 곤란한 일이 일어나는지 소개하고 있다. 데이터베이스를 이용함으로써 이러한 어려운 과제를 현실적으로 해결할 수 있도록 하자.

대량의 데이터 중에서 필요한 것을 빨리 반환할 수 없다

소셜 게임을 만든다고 가정하여 사용자 정보를 관리하고 싶다라고 하자. 사용자 정보를 관리함에 있어서는 먼저 사용자 수가 얼마나 되는지를 대략적으로 예측해야 한다. 크게 성공하면 사용자 수가 1,000만 건 정도 될지도 모른다. 1,000만 건이라고 하면 대단히 많다는 인상을 받을지 모르겠지만, Mobage와 같은 대형 소셜 게임은 일본 내 사용자 수만 3,000만 명을 초과하고 있다.

또한 각 사용자의 행동 정보를 관리하려고 하면 각 사용자별로 일기 등의 활동이 있기 때문에 수십억 개 정도의 레코드에 쉽게 도달하게 된다. 이런 상황을 감안해서 대외용 애플리케이션이라면 적어도 억 단위의 레코드가 취급 가능한 데이터 관리를 생각해 두어야 한다.

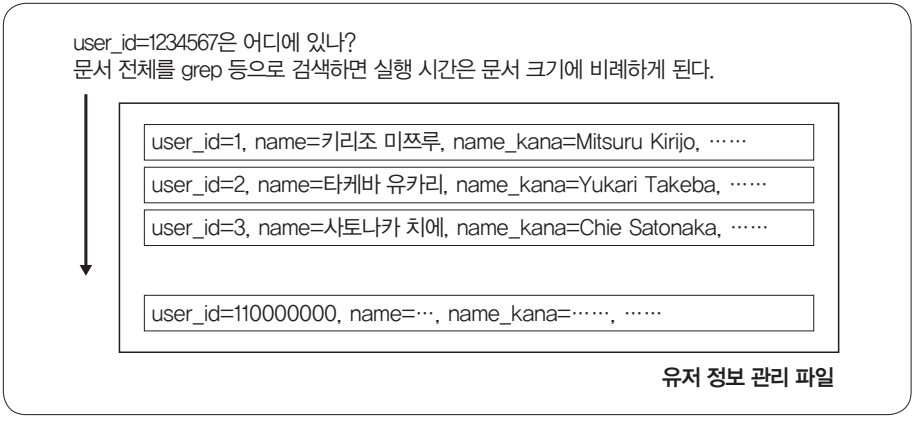
애플리케이션 처리의 관점에서는 이러한 수많은 건수의 데이터 중에서 원하는 것을 고속으로 추출해야 한다. Mobage나 mixi와 같이 인증을 동반하는 웹사이트에서는 인증 후 해당 사용자의 홈페이지로 전환한다. 홈페이지의 내용은 친구의 최신 일기 목록과 같이 해당 사용자에 입각한 정보가 포함되어 있다. 따라서 그 사용자를 키로 원하는 데이터를 추출하는 처리가 필요하게 된다.

그럼 1,000만 건이 넘는 데이터 중에서 자신의 사용자 ID와 관련된 정보를 도대체 어떻게 하면 빠르게 검색할 수 있을까?

아니, 근본적으로 1,000만 건의 데이터를 어떻게 관리하면 좋을까? 텍스트 파일로 그림 1-1과 같이 관리하면 좋을까? 아니면 보기 쉽게 Excel 시트 등으로 관리하면 좋을까?

이와 같은 방식으로 관리하면 원하는 데이터가 어디에 있는지 즉시 알 수 없기 때문에 하나씩 샅샅이 조사해 나가야 한다. 당연한 이야기지만, 이럴 경우 데이터의 양에 비례한 시간이 걸린다. 100만 건이라면 10만 건의 경우의 10 배, 1,000만 건이라면 10만 건의 경우의 100 배가 걸린다. 데이터가 증가함에 따라 머지않아 현실적인 대처 방법이 없어질 것이다.

● **그림 1-1** 샅샅이 조사하는 검색은 전혀 실용적이지 않다



따라서 보다 효율적인 데이터 구조로 관리해야 한다.

이것은 「Search(탐색)」라는 기술 영역으로 알려져 있으며, 고속으로 액세스하기 위해서는 「B+Tree 인덱스」나 「해시 인덱스」와 같은 데이터 구조로 이러한 레코드를 관리할 필요가 있다[주2]. 텍스트 파일은 물론 Excel과 같이 간단한 스프레드 시트에서는 인덱스의 구조를 가진 상태에서 다음의 레코드를 관리하는 방법이 없기 때문에 필요한 데이터를 신속하게 검색할 수 없다.

대량의 데이터를 메모리 내에서만으로는 취급할 수 없다

프로그래밍에 익숙한 사람이라면 인덱스 구조를 지원하는 클래스 라이브러리를 평소에 사용하고 있을지 모르겠다. 「Java의 HashMap 클래스는 키와 값의 쌍으로 관리할 수 있으며, 키를 지정하면 초고속으로 값을 취할 수 있다. 이것을 사용하면 데이터베이스는 필요 없지 않을까?」라고 생각할지도 모른다.

그러나 이러한 기초적인 유틸리티 라이브러리에 의해 관리되는 인덱스는 메모리 내에서밖에 사용할 수 없다. 용량이 증가했다고 해도 실제 메모리의 크기는 한정되

주2 인덱스는 제2장(19쪽)에서 설명할 것이다.

어 있기 때문에 대량의 사용자를 취급하는 것은 어려울 것이다. 또한 이보다 더 큰 문제라면, 메모리는 휘발성이기 때문에 시스템 다운 등이 일어나면 데이터가 날아가 버린다.

날아간 데이터를 어떻게 복구할 수 있을까? 이것도 중요한 주제다. 설사 날아가 버려도 곤란하지 않도록 정기적으로 디스크 등에 기록해 두면 될 것인가? 정기적으로 기록하여 보관한다 해도 기록한 후와 그 다음의 기록 사이에서 시스템 충돌이 발생하면 그 사이의 업데이트는 사라지게 된다. 그 사이의 변경 이력은 어떻게 취하면 좋은 것일까?

또한 디스크로의 기록은 얼마의 시간이 걸릴까? 값싼 제품이라면 HDD(하드 디스크 드라이브)의 읽고 쓰기 속도가 기껏해야 초당 50MB 정도다. 데이터 양이 500GB인 경우, 디스크의 기록시간이 500GB ÷ 50MB = 10,000초 ≒ 약 3시간 정도 소요된다. 또한 로딩하는 데에도 동일한 정도의 시간이 소요된다. 2012년 현재, 500GB 이상의 메모리를 탑재하고 있는 하드웨어는 거의 없지만 수십 GB 단위라 해도 수십 분이 필요하다는 것을 계산을 통해 알 수 있다. 만일 해당 하드웨어가 원격 서버라면, 원격 서버에 백업하기 위해 전송 등의 작업도 발생하므로 전체적으로 1시간 이상의 시간이 필요하게 될 것이다.

객체로서 메모리에 전개하는 것이 끝나지 않으면 후속 처리를 수행할 수 없으니 서버가 망가진 경우는 복구하는 데만 1시간 단위의 다운 타임을 필요로 한다는 것이다. 이것은 대부분의 서비스에서 허용되지 않는 것이다.

즉, 메모리만으로 모든 데이터를 갖는 것 자체는 무모한 아키텍처로, 디스크를 활용하는 것을 전제로 한 데이터 구조가 필요하게 된다. 데이터베이스는 이러한 전제로부터 만들어졌으며, 실제로 이러한 과제를 해결하고 있다. 한편, 메모리 내에서 여러 서버 간에서 동기화하여 데이터를 갖고 있는 데이터베이스도 등장하여 나름대로의 지위를 구축하고 있다.

장애가 발생했을 때 빠른 복구가 어렵다

데이터베이스가 사용되는 이유 중에서 가장 현실적인 이유는 처음에 설명한 성능 면에서의 이유와 여기에서 설명할 장애 대책이라고 생각한다. 개인이 취미로 사용하거나 만들거나 하는 소프트웨어와 기업에서 적지않은 돈이 움직이는 시스템에서의 장애 대책은 근본적으로 달리 대응해야 한다.

개인이 사용하는 소프트웨어는 결함으로 인해 동작이 멈추었다고 해도 어느 정도 참을 수 있다. 그러나 1분만 정지해도 수백만의 손실이 발생하는 수준의 기업 내 시스템은 개인이 책임질 수 있는 범위를 훨씬 초과하는 충격이 있기에 회사는 이에 대한 대책을 철저히 할 필요가 있다. 상장 기업의 경우, 회사의 간판 시스템이 다운되면 주가가 내려가는 것을 종종 목격하게 된다. 주주 대표 소송이 발생하거나 매출 하락으로 인원 삭감을 강요받는 등의 위험이 있다. 장애 대응을 위한 준비는 따분한 면이 있긴 하지만 안정적인 가동을 위해 빠뜨릴 수 없는 것이다.

장애 대응의 방법과 수준은 다양하다[주3]. 복구 노력이라는 관점에서 큰 요인이 되는 것은「데이터가 사라졌는지에 대한 여부」다. 웹 서버나 캐시 서버처럼 자체적으로 데이터를 갖고 있지 않는 것에 대해서는 기본적으로 다시 시작하거나 증설하거나 해서 서비스를 계속 수행할 수 있으므로 장애 대응의 수고는 그리 곤란하지 않은 수준이다.

한편, 필요한 데이터가 소실되는 경우도 있다. 개인이 소유하고 있는 PC의 HDD가 손상되거나 하면 저장하고 있던 데이터는 사라지게 된다. 노트북과 같이 자주 들고 다니는 하드웨어에서는 특히 그런 아픈 경험을 한 적이 있는 사람이 많을 것이다. 물리적인 고장 등에 의해 데이터가 날아가 버린 경우에 일반적인 복구 방법이「백업」이다. 백업이 있으면 그 데이터를 되돌림으로써 일단 전손(全損) 상태는 피할 수 있다. 그러나 단순한 백업만으로는 다음과 같은 문제가 있다.

주3 2010년 8월에 발생한 mixi의 장애는 캐시 서버인 memcached의 결함에 의한 것이었다. 그리고 이에 대한 복구에 며칠이 소요되었다는 보도가 있었다. 고부하 시에만 크래쉬가 발생하는 버그로 원인 규명 또한 꽤 어려움이 있었다고 한다. 이는 영구적인 데이터가 없는 서버라고 해서 대책을 소홀히 해서는 안 된다는 사실을 말해주는 사례라고 할 수 있다.

- 최종 백업 이후에 업데이트된 결과를 되돌릴 수 없다.
- 백업 데이터를 다시 되돌리는 처리에도 시간이 걸리고, 그 기간 동안은 다운 타임이 된다.
- 백업 중에 섣불리 업데이트를 하면 백업이 손상될 위험이 있다.

PC의 데이터를 자주 백업하고 있는 사람이라고 해도 그래 봐야 주1회~월1회 정도일 것이다. 웹 서비스에서 그러한 느슨한 방침을 취하고 있으면 1개월 정도의 데이터 손실이 발생할지도 모른다.

또한 백업 파일이라는 것이 TB급이 되는 경우도 많이 있다. 이러한 파일을 되돌리는 데에 걸리는 시간은 스토리지(HDD 등)의 전송 시간에 크게 의존한다. 초당 50MB 정도의 속도라면 500GB 전송에 3시간이 걸린다는 것은 앞서 설명하였다. 그러한 시간이 모두 다운 타임으로 되어 버리는 것은 문제라고 할 수 있다.

백업을 복원하는 경우뿐만 아니라 백업을 만들기 위해서도 당연히 데이터의 양에 비례한 시간이 필요하다. 백업 취득 중에 업데이트가 발생되는 경우는 그 백업의 내용이 일관성이 있는지에 대한 여부에도 신경을 쓸 필요가 있다. 예를 들어 유저가 어떤 상품을 포인트로 구입하는 경우, 시스템은 해당 유저가 갖고 있는 포인트를 줄이고 장바구니 내의 상품 수를 늘려서 출하 등의 상태 관리를 해야 할 것이다. 이러한 작업이 한창 진행 중인 상태에서 백업을 할 경우, 백업 내용에는 포인트만이 감소되어 있고 장바구니의 상품 수는 증가되지 않는 일이 발생하지 않도록 배려할 필요가 있다.

이러한 무결성 관리를 제대로 하려면 「트랜잭션」이라는 개념을 구현할 필요가 있는데, 애플리케이션 로직으로 구현하려면 많은 어려움이 있다. 그렇기에 데이터베이스를 사용하여 이러한 무결성 관리를 담당시킬 수 있는 것이다.

괜찮은 시스템에서는 이에 더하여 동일 데이터가 있는 서버의 **복제**(replication)를 갖게 하는 구성을 많이 취하고 있다. 백업을 하는 경우든 복제를 하는 경우든 서비스를 멈추지 않고 실행해야 한다.

◆◆◆

어느 경우에도 백업으로부터의 복구는 매우 시간이 걸리는 작업이며, 이러한 작업이 필요하지 않도록 설계하는 것이 서비스 운영자의 실력이다. HDD가 손상되었다는 등의 이유로 데이터가 날아가 버리는 것 이외에도 단순히 서버 룸의 전원이 꺼지는 등의 이유로 파일이 손상되는 장애

도 발생할 수 있다. 이러한 상황에서도 데이터베이스는 유용하다. 데이터베이스는 탄력성을 고려하여 설계되어 있고, 쓰기 도중에 크래쉬하는 경우에도 다시 시작할 때에는 자동으로 장애 부분을 감지하여 일관성 있는 상태로 복구시켜 준다. 이것도 배후에는 트랜잭션의 구조가 있다.

병렬성 제어가 어렵다

여러 사람이 이용하는 애플리케이션에서 어려운 기술적인 포인트 중 하나가 「배타 제어」다. 예를 들어 동일한 파일에 대해 여러 사람이 동시에 쓰면, 아무런 배타 제어를 하지 않은 경우에 먼저 쓴 사람의 결과는 나중에 쓴 사람에 의해 덮어 써지게 된다. 즉, 먼저 쓴 사람의 결과는 흔적도 없이 사라져 버리게 된다.

개인이 사용하는 PC의 애플리케이션에서는 자신 이외의 다른 사람으로부터 액세스가 없다는 전제가 있기 때문에 배타 제어를 생각하지 않는 경우가 많으나, 웹 서비스와 같이 매우 많은 사람들이 동시에 액세스하는 환경에서는 필수적인 기술이다.

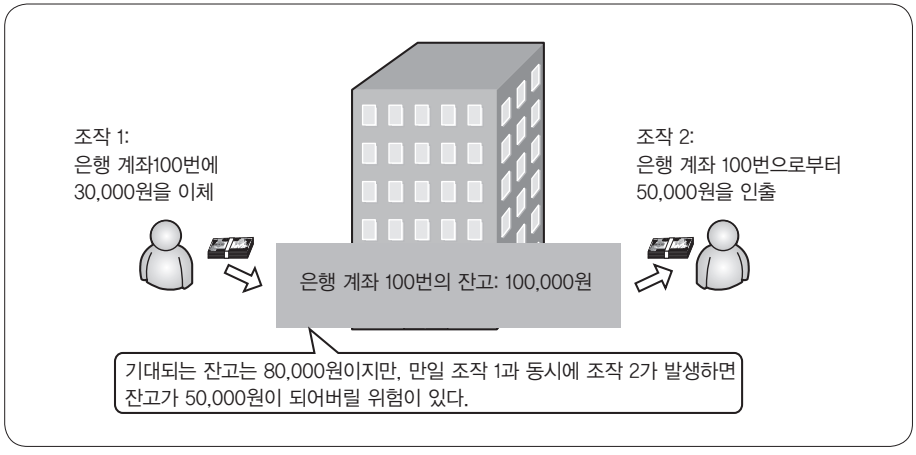
배타 제어가 필요한 것은 당연하지만 단순히 구현해서 될 일은 아니다. 특히 「배타의 범위」에 대해서는 고려해야 할 필요가 있다. 가장 단순한 형태로 알려져 있는 것은 「잠금 파일(Lock File)」이다. 이것은 여러 사람이 이용하는 데이터 소스(파일 등)를 업데이트하기 전에 전용의 이름으로 잠금 파일을 만들어 쓰기 예약을 하는 것이다.

쓰기 전에 이 잠금 파일이 있는지 체크하여 만약 존재하면 누군가가 작성하고 있다고 판단해 오류를 반환(또는 일정 시간 대기)하게 한다. 쓰기를 다 마치면 잠금 파일을 삭제한다. 데이터를 읽어들일 때에도 도중에 수정된 어중간한 상태의 데이터를 읽지 않도록 다른 잠금 파일을 도입하여 「읽는 당사자들 간에는 경쟁하지 않지만 읽고 있는 동안에는 업데이트를 하지 못하도록 하는」 제어를 한다. 애플리케이션 측에서 잠금 파일의 존재 확인 및 작성/삭제 처리를 해야 할 필요가 있어 약간은 수고스럽지만, 이러한 배타 제어도 로직으로서는 제대로 작동한다.

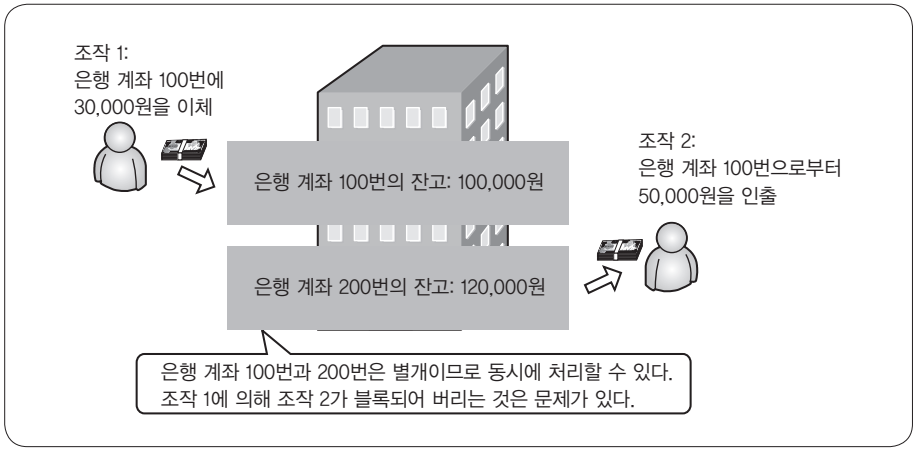
그러나 잠금 파일 방식의 경우 동시에 한 명만이 갱신 액세스를 할 수 있기 때문에 실행 성능 면에서 매우 큰 제약이 생기게 된다. 그래서 다른 레코드에 동시에 업데이트할 수 있는 「행 수준의 범위」를 갖게 하고 싶기도 하다. 이렇게 되면 단순한 파일로서의 제어가 어렵다. 데이터베이스를 사용하면 이러한 배타 제어도 간단히 할 수 있

다. 이러한 데이터 세트의 배타 제어 또한 데이터베이스의 전문적인 부분이다.

● **그림 1-2** 배타 제어를 하지 않으면 일관성이 무너진다



● **그림 1-3** 배타 제어의 범위가 넓으면 병렬성이 떨어진다



이러한 병렬성 문제를 은행에서의 계좌 거래를 예로 들어 그림 1-2와 그림 1-3으로 정리하였다. 배타 제어를 실시하지 않으면 일관성이 무너져 버리기 때문에 논외가 되는데(그림 1-2), 배타 제어의 범위가 넓으면 병렬성이 떨어지고 속도 면에서 실용성이 줄어든다(그림 1-3). 특히 최근에는 멀티코어 CPU가 대두되고 있음에도 동시에 하나의 처리밖에 실행할 수 없다는 것은 한 개 코어만을 사용해야 한다는 것이기도 하여 코어 수의 증가가 있더라도 전혀 효과가 없게 된다. 멀티코어 CPU를 활용하기 위해서는 병렬성 향상을 빼놓을 수 없다.

데이터 무결성을 보장하는 것은 어렵다

「병렬성 제어」와도 강하게 연관된 요소가 있는데, 데이터 자체가 불일치를 일으키지 않게 하는 구조를 만드는 것은 매우 중요하다. 예를 들어 사용자 ID의 경우, 다른 사람에 대해 동일한 ID가 생성되지 않도록 고유의 값으로 할당해야 할 필요가 있다. 프로그래밍으로는 「id = id + 1」 등 하나씩 늘려 가면 고유의 값이 되지만, 동시에 액세스가 발생했을 때 동일한 ID를 할당하지 않도록 상호 배제하는 것은 필수이며, 시스템이 멈춰서 다시 시작하는 경우에도 과거에 할당된 ID를 다시 할당하지 않도록 이력 관리가 필요하다. 이는 쉽게 보일지 모르겠지만 의외로 대단히 힘든 것이다. 상당수의 데이터베이스 제품은 이러한 영속성 있는 고유 ID를 빠르게 할당하는 기능을 가지고 있다.

그 밖에도 「일기를 쓸 수 있는 사용자는 사전에 등록된 사용자뿐이다」라는 것과 같이 값을 취할 수 있는 범위에서 제한의 설정에 대한 필요성도 생긴다. 이러한 처리는 애플리케이션 로직으로도 구현할 수 있지만, 데이터베이스가 가지는 「참조 무결성 제약 조건」 기능을 이용하면 데이터베이스 기능으로 값을 체크할 수 있다.

1.3

이 책에서는 무엇을 다루어 나갈 것인가?

이 책에서는 지금까지 언급한 기술적인 과제에 대해 **데이터베이스를 사용하면 왜 해결되는지, 어떻게 사용하면 해결할 수 있는지**에 대한 관점으로 설명을 진행시켜 갈 것이다. 또한 데이터베이스 기술은 결코 기존 테마가 아닌 지금도 눈부신 개선이 이루어지고 있는 분야이기도 하다. 예를 들어 최근 주목을 끌고 있는 「NoSQL」은 기존의 데이터베이스에서는 충분한 성능이 나오지 않거나 여러 서버를 구축하기 어려운 경우에 일부 기능을 삭제함으로써 이것들을 실현하는 것을 목표로 한 것이다. 한편, 종래의 데이터베이스에 비해 안정성이 떨어지는 점 등이 문제가 되고 있어 기존의 데이터베이스를 확장하여 이러한 문제를 해결해 나가는 노력도 이루어지고 있다.

또한 SSD(Solid State Drive)와 수십 개 이상의 CPU 코어가 대두됨에 따라 데이터베이스 자체의 개선도 이루어지고 있다. 그리고 데이터의 양이 증가함에 따라 전체 텍스트 검색 및 데이터웨어 하우스 같은 분석 시스템 분야에서의 활용도 요구되고 있다. 데이터베이스 기술은 이러한 문제에 대해 어떤 대답을 내놓을 것인가? 그러한 내용에 대해서도 설명하고 싶다.

CHAPTER 1 요약

본 장에서는 「데이터베이스가 없으면 무엇이 곤란한가」에 대해 소개했다. 잘 알고 있는 사람에게 있어서는 「뭘 새삼스럽게」라고 생각될지도 모르지만, 경험이 있는 사람들 중에도 의외로 간과하고 있는 사람이 많이 있는 것 같다. 데이터베이스 기술은 「여러 사람」이 「대량의 데이터를 처리」하고 「망가지면 큰일이 나는」 타입의 애플리케이션에서는 빼놓을 수 없는 것이다. 대학 과정 등에서는 대부분 그냥 넘어가 버리는 이유 중 하나가 이러한 애플리케이션을 개인적으로 개발할 기회가 없어 아무래도 사회에 나와 회사에서 일하게 되고 나서야 경험하게 되기 때문일 것이다. 그러한 본질을 낱낱이 소개하고자 한다.

본문에서 언급한 「대용량 데이터 중에서 필요한 것을 고속으로 반환한다」라는 것은 애플리케이션에서는 필수 요구 항목으로, 이를 실현하는 것이 「인덱스」다. 다음 장에서는 인덱스를 소개하겠다. 인덱스가 없는 경우와 인덱스가 있는 경우에 어느 정도의 속도 차이가 나오는지 확인해 보도록 하자.