1 부

iBATIS 소개

iBATIS에 대한 고수준(high-level)의 소개를 하면서 이 책을 시작한다. 1, 2장에서는 iBATIS의 철학과 다른 퍼시스턴스 솔루션과의 차이점을 다룰 것이다. 자바에는 매우 많은 퍼시스턴스 솔루션들이 있다. 그리고 어떤 것을 언제 사용해야 할지 제대로 이해하기가 녹록치 않다. 1부의 1,2장을 읽고 나면, iBATIS의 기반이 되고 개발자들이 사용하게 될 기본 원리와 가치를 이해하게 될 것이다.

iBATIS

1 장

iBATIS의 탄생 철학

: iBATIS 역사

: iBATIS 이해

: 데이타베이스 종류

SQL[Structured Query Language]이 나온 지도 상당히 오래되었다. 에드가 F. 코드가 데이터를 관계형 테이블의 집합으로 나타낼 수 있다는 아이디어를 제안한 것이 이미 35년이 넘었다. 그 이후로, IT 회사들은 관계형 데이터베이스 관리 시스템(RDBMS)에 수십억 달러를 투자했다. 하지만 몇 안 되는 소프트웨어 기술만이 관계형 데이터베이스와 SQL에 걸린 시간만큼 오랜 기간 동안의 시험 과정을 통과할 수 있었을 뿐이다.

이렇듯 오랜 기간이 지났음에도, 관계형 데이터베이스는 여전히 엄청난 힘을 뿜어내고 있으며 세계에서 가장 큰 규모를 가진 소프트웨어 회사들의 시스템을 구성하는 중요한 기반 구조의 역할을 하고 있다. 모든 지표들은 SQL이 향후 30년을 더 버텨내리라는 것을 예견하고 있다.

iBATIS는 관계형 데이터베이스와 SQL의 가치를 인정하고 산업 전반에 걸친 SQL에 대한 투자를 그대로 이어가겠다는 생각을 기반으로 하고 있다. 우리는 데이터베이스와 SQL 그 자체가 애플리케이션의 소스코드나 혹은 여러 번의 업그레이드를 거친 소스코드보다도 더 오래 살아남는 것을 보곤한다. 어떤 경우에는 애플리케이션이 다른 언어로 새로 작성됨에도 불구하고 SQL과 데이터베이스는 대부분 변경 없이 그대로 사용되는 것을 볼 수 있다. 이것이 iBATIS가 SQL을 피하거나 숨기려 들지 않는 까닭이다. 대신 iBATIS는 SQL로 더 쉽게 작업할 수 있도록 SQL 사용을 기꺼이 받아들였고, 현대적인 객체 지향 소프트웨어와의 통합을 더 쉽게 만들어 주는 퍼시스턴스 계층(persistence layer) 프레임워크로 자리매김했다. 최근에는 데이터베이스와 SQL이 객체 모델을 위협한다는 루머가 돌지만 그렇게 될 리가 없다. iBATIS가 그렇게 놔두지 않을 것이다.

1장에서는 iBATIS의 역사와 작동 원리를 알아보고, iBATIS의 탄생에 영향을 끼친 것들에 대해 공부해 본다.

1.1 복합적인 솔루션 : 최고 중의 최고들로 구성하기

현대 세계에서는 복합적인 솔루션을 어디서든 찾아볼 수 있다. 겉으로 보기엔 서로 반대되는 두 생각을 받아들여 중간에서 그것들을 합치면 간극을 메워주는 아주 효율적인 방법이 만들어질 수 있음이 증명되었다. 때로는 이러한 조합이 완전히 새로운 산업을 창조해 버리는 결과를 낳기도 한다. 자동차 산업에서 이러한 예를 확실히 볼 수 있다. 자동차 산업에서 대부분의 혁신은 여러 가지 아이디어를 조합해서 나온 것이다. 승용차를 화물차와 합쳐서 멋진 가족용 미니밴을 만들어 냈다. 트럭을 전천후 차량(all-terrain vehicle)과 결합시켜서 도시적 신분의 상징이 된 SUV를 낳았다. 핫로드hotrod[1]와 스테이션 웨건[2]을 접목시켜서 아버지가 운전하기에 창피하지 않을 법한 가족용 자동차를 만들어 냈다. 휘발유 엔진에 전기 모터를 결합시켜 북미의 오염 문제에 대한 훌륭한 해결책을 제시하기도 했다.

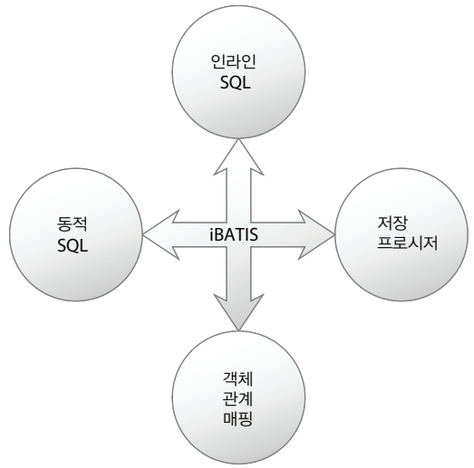
복합적인 솔루션은 IT 산업에 있어서도 그 효과를 증명했다. iBATIS가 소프트웨어 애플리케이션의 퍼시스턴스 계층을 위한 바로 그러한 복합적인 솔루션이다. 그 동안 애플

1. **역자주** | 양산차를 기본으로 하여 엔진이나 기타 부품들을 개조 또는 교환해서 가속 성능이나 개성적인 디자인을 추구하는 작업. 이런 작업을 즐기는 사람을 핫로더라 한다.
2. **역자주** | 자동차 차체 형태의 한 종류로 세단형과 유사하나 차체 뒷쪽에 화물 공간이 있는 것이 특징이다. http://ko.wikipedia.org/wiki/%EC%8A%A4%ED%85%8C%EC%9D%B4%EC%85%98_%EC%9B%A8%EA%B1%B4

리케이션에서 데이터베이스에 SQL을 실행시키는 다양한 방법들이 개발되었다. iBATIS는 여러 가지 접근 방법들로부터 다양한 개념을 빌려다 만든 독보적인 솔루션이다. 어떤 접근법을 사용했는지 살펴보는 것으로 시작해보자.

1.1.1 iBATIS의 기원 답사

iBATIS는 관계형 데이터베이스에 접근하는 가장 잘 알려진 방법들로부터 가장 좋은 특징과 아이디어들을 차용하고, 그것들로부터 시너지를 이끌어낸다. 그림 1.1은 iBATIS 프레임워크가 몇 년 간에 걸쳐 데이터베이스 통합을 위해 서로 다른 여러 접근방법을 사용하여 개발하며 배운 것들을 취합하고, 복합적인 솔루션을 만들기 위해 최고의 지혜들을 모아 조합하는 것을 보여준다.



그 림 1.1 | iBATIS가 개발 프로세스를 단순화하기 위해 끌어다 모은 몇몇 개념들

다음 절은 데이터베이스와 소통하는 다양한 접근 방법들을 논의하고, iBATIS가 응용한 각 부분들에 대해 알아볼 것이다.

SQL

iBATIS의 심장부에는 SQL이 자리잡고 있다. 정의에 따르면, 모든 관계형 데이터베이스는 데이터베이스와 소통하는 1차적인 방법으로 SQL을 지원한다. SQL은 단순하고 비절차

적(non-procedural)인 언어로 데이터베이스와 함께 작동하며 사실은 두 가지 언어를 하나로 합친 것이다.

첫째는 데이터 정의 언어(Data Definition Language, DDL)로 CREATE, DROP, ALTER 같은 구문을 포함하고 있다. 이 구문들은 테이블, 칼럼, 인덱스, 제약조건, 프로시저 그리고 외래키 관계 등을 포함한 데이터베이스의 구조와 설계를 정의하는데 사용한다. DDL은 iBATIS가 직접적으로 지원하는 것은 아니다. 비록 많은 사람들이 iBATIS를 이용해서 DDL을 성공적으로 실행시키고 있지만 사실 DDL은 일반적으로 데이터베이스 관리자들을 위한 것이며 개발자들의 영역 밖에 있는 것이다.

SQL의 두 번째 부분은 데이터 조작 언어(Data Manipulation Language, DML)이다. 이것은 SELECT, INSERT, UPDATE, DELETE 같은 구문들을 포함하고 있다. DML은 데이터를 직접 조작하기 위해 사용한다. 원래 SQL은 최종 사용자들이 사용하기에도 충분할 정도로 단순하게 설계된 언어이다. SQL을 사용하기 위해 복잡한 사용자 인터페이스의 프로그램은 필요 없으며 아니면 아예 애플리케이션을 사용할 필요조차도 없게 설계한 것이다. 물론 이건 우리의 최종 사용자들에게 더 큰 희망을 품어볼 수 있었던 흑백 터미널을 사용하던 시절 얘기다![3]

요즘에는 데이터베이스가 너무 복잡해져서 최종 사용자가 데이터베이스에 SQL을 직접 날리는 것을 허용해 줄 수는 없다. 상상할 수 있겠는가? "이봐요, BSHEET 테이블에서 당신이 찾는 정보를 찾을 수 있을 겁니다."라고 말하며 회계 관리 부서 직원들에게 SQL 목록을 넘겨주는 모습을...

SQL을 직접 사용하는 것이 최종사용자에게는 더이상 효과적인 방법이 될수는 없지만, 개발자들에게는 매우 강력한 도구이다. SQL은 데이터베이스에 접근하는 진정한 최적의 도구이다. SQL이 아닌 다른 데이터 접근 방식들은 SQL이 가진 전체 기능들 중의 일부만을 수행할 수 있을 뿐이다. 이런 까닭에 iBATIS는 관계형 데이터베이스에 접근하는 1차적인 방법으로 SQL 사용을 채택하였다. 동시에 iBATIS는 아래에서 논의할 저장 프로시저와 객체 관계 매핑(object/relational mapping, O/RM) 도구 등과 같은 다른 접근 방법들의 여러 장점들도 제공하고 있다.

3. **역자주** | 당시의 애플리케이션 사용자들은 매우 높은 컴퓨터 사용 능력을 보유하고 있었다는 의미임

옛날식의 저장 프로시저(Stored Procedure)

저장 프로시저는 아마도 관계형 데이터베이스로 애플리케이션 프로그램을 작성하는 가장 오래된 방법일 것이다. 많은 옛날 애플리케이션들이 요즘에는 2-티어설계라고 불리는 방식을 사용했다. 2-티어 설계는 리치 클라이언트 인터페이스에서 데이터베이스에 있는 저장 프로시저를 직접 호출하도록 구성돼 있다. 저장 프로시저는 데이터베이스에서 실행시킬 SQL을 포함하고 있었다. 또 SQL과 함께 비즈니스 로직을 포함할 수도 있었으며 종종 실제로 포함하기도 했다. SQL과는 달리 저장 프로시저 언어는 절차적이며 조건문이나 반복문같은 흐름 제어부를 가지고 있었다. 확실히 다른 어떤 것도 필요 없이 저장 프로시저만 가지고도 전체 애플리케이션을 작성하는 것이 가능할 정도였다. 많은 소프트웨어 업체들이 2-티어 데이터베이스 애플리케이션 개발을 위한 리치 클라이언트 도구들을 개발했는데, Oracle Forms, PowerBuilder, Visual Basic 등이 바로 그것이다.

하지만 2-티어 애플리케이션은 성능과 확장성에 문제가 있었다. 비록 데이터베이스가 엄청나게 강력한 장비라고 하더라도 수백, 수천 혹은 수백만이 될 수도 있는 사용자들을 감당하기에 최적의 선택이라고 보기는 어렵다. 현대 웹애플리케이션에서는 이러한 확장성을 요구하는 경우가 일반적이다. 동시 사용자 라이센스, 하드웨어 자원 그리고 네트워크 소켓 등을 포함한 제약 사항들 때문에 2-티어 아키텍처는 대용량 작업에서 성공하기 힘들다. 게다가 2-티어 애플리케이션을 배포하는 것은 악몽이라 할만하다. 리치 클라이언트 애플리케이션의 배포와 관련된 일반적인 문제들과 더불어 복잡한 실행 데이터베이스 엔진까지도 클라이언트 장비에 배포해야 하는 경우도 생겼다.

현대적인 저장 프로시저

어떤 영역에서는 저장 프로시저가 여전히 웹 애플리케이션과 같은 3-티어나 N-티어 애플리케이션 개발에서 가장 좋은 해결책으로 꼽히고 있다. 요즘 저장 프로시저는 보통 중간 티어(middle tier)에서 호출되는 원격 프로시저처럼 사용되고 있으며, 성능에 관한 많은 제약들이 커넥션 풀이나 데이터베이스의 자원을 관리함으로써 해결되었다.

저장 프로시저는 여전히 현대적인 객체 지향 애플리케이션의 데이터 접근 계층 전체를 구현하는데 사용할 수 있는 유효한 방법이다. 또 저장 프로시저는 다른 방법들에 비해 데이터베이스의 데이터를 가장 빨리 관리할 수 있는 성능상 이점을 가지고 있다. 하지만 단순한 성능만 아니라 여러 가지 고려해야 할 사항이 더 있다. 업계에서는 비즈니스 로직을 저장 프로시저에 두는 것을 좋지 않은 개발 방식으로 간주하고 있다. 그렇게 여기는 큰

이유는 저장 프로시저가 현대 애플리케이션 아키텍처와 잘 맞아 떨어지게 개발하기 어렵기 때문이다. 저장 프로시저는 작성하고 테스트하고 배포하기가 어렵다. 더더욱 안 좋은 점은 데이터베이스가 개발 팀이 아닌 다른 팀의 영역인 경우가 종종 있고 엄격한 변경 관리가 어렵다는 점이다. 저장 프로시저는 현대 소프트웨어 개발 방법들을 따라잡기에는 변경을 받아들이는 속도가 충분하지 못하다. 게다가 저장 프로시저는 비즈니스 로직을 제대로 구현하기에 기능이 충분치 못하다는 문제도 있다. 비즈니스 로직이 다른 시스템이나 혹은 사용자 인터페이스를 제어하려 한다고 할 때 저장 프로시저는 그 모든 로직을 처리하기에는 역부족이다. 현대 애플리케이션은 매우 복잡하며 데이터를 조작하는데 최적화 되어있는 저장 프로시저보다는 좀 더 일반적인 프로그래밍 언어를 필요로 한다. 이런 점 때문에, 몇몇 벤더들은 자바같은 강력한 언어를 자신들의 데이터베이스 엔진에 추가하여 더 강력한 저장 프로시저를 만들 수 있게 하고 있다. 하지만 이것이 상황을 진정으로 해결해 주지는 못한다. 이것은 애플리케이션과 데이터베이스의 경계를 불분명하게 만들 뿐이고 데이터베이스 관리자들에게 자신의 데이터베이스에서 자바나 C#을 사용해야 하는 짐을 안겨줄 뿐이다. 이것은 문제를 해결하는 적절한 방법이 아니다.

소프트웨어 개발에서 자주 다루는 문제로 과잉수정(overcorrection)이라는 것이 있다. 어떤 문제를 발견했을 때 처음으로 시도한 해결책인 종종 실제 해결책에 대한 정반대의 접근법이기도 하다. 문제를 해결하는 대신 완전히 다른 문제들을 꼭 그 만큼 더 만들어내는 것이다. 이제 이와 관련해 인라인(Inline) SQL을 논의해 보려 한다.

인라인 SQL

저장 프로시저의 한계를 극복하는 접근 방법으로 일반 프로그래밍 언어에 SQL을 내장시키는 방법이 있다. 로직을 데이터베이스로 옮기는 대신, SQL을 데이터베이스에서 애플리케이션 코드로 옮기는 것이다. 이로 인해 SQL 구문이 프로그래밍 언어와 직접적으로 상호작용 하는 것이 가능하다. 어떤 점에서는 SQL이 프로그래밍 언어의 기능 중에 하나가 되는 것이다. 이것은 COBOL, C, 게다가 자바 등을 포함한 많은 프로그래밍 언어에서 지원된다. 아래는 자바 에서 사용하는 SQLJ에 대한 예제이다.

```
String name;
Date hiredate;
#sql {
      SELECT emp_name, hire_date
      INTO :name, :hiredate
      FROM employee
      WHERE emp_num = 28959
};
```


인라인 SQL은 프로그래밍 언어와 잘 통합된다는 점에서 상당히 멋지다. 해당 언어의 변수가 SQL의 파라미터로 직접 넘어갈 수도 있고, SQL 실행 결과도 비슷한 방식으로 변수에 직접 값을 넘겨줄 수 있다. 어떻게 보면 SQL이 프로그래밍 언어의 일부가 되는 것이다.

하지만 불행하게도 인라인 SQL은 폭넓게 받아들여지지 못했고, 몇몇 중대한 문제들 때문에 그 영역을 확장하기가 어렵다. SQL에는 여러 확장 기능들이 있고, 그것들은 특정 데이터베이스에서만 작동한다. 이렇게 SQL이 여러 가지 형태로 파생되면서 모든 데이터베이스 플랫폼에서 완벽하고 이식성 있는 인라인 SQL 파서를 만들기가 어렵다. 인라인 SQL의 두 번째 문제는 이것이 보통 실제 프로그래밍 언어의 일부로써 구현되지 못한다는 점이다. 대신, 전처리기(precompiler)가 인라인 SQL을 특정 언어에 맞는 코드로 변환하는 방식을 사용한다. 이로 인해 IDE 같은 개발 도구들이 문법 강조 기능이나 코드 자동완성 같은 고급 기능들을 제공하기 위해 인라인 SQL 코드를 해석할 필요가 생긴다. 인라인 SQL을 포함한 코드는 전처리기가 없으면 컴파일조차 되지 않으며 이러한 의존관계는 코드의 유지보수성을 떨어뜨리는 요인이 된다. 프로그래밍 언어 측면에서 인라인 SQL의 이러한 문제를 해결하는 한 가지 방법은 애플리케이션에서 SQL을 문자열 같은 자료 구조로 나타내는 것이다. 이러한 접근 방법을 보통 동적(dynamic) SQL이라고 한다.

동적 SQL

동적 SQL은 전처리기를 사용하지 않고 인라인 SQL을 다루는 방법이다. 대신, 현대 프로그래밍 언어에서 다른 문자 데이터를 다루는 것과 마찬가지로 SQL을 문자타입으로 다룬다. SQL이 문자타입이므로 인라인 SQL처럼 직접 프로그래밍 언어의 특징을 사용할 수는 없다. 따라서 동적 SQL은 SQL 파라미터를 설정하고 결과 값을 가져오는 견고한 API를 구현할 필요가 있다.

동적 SQL은 유연하다는 장점이 있다. 서로 다른 파라미터나 동적인 애플리케이션의 기능에 따라서 실행 시간에 SQL이 변경될 수 있다. 예를 들어 Query-By-Example[4]

4. **역자주** | 사용자에게 각 칼럼 이름과 그 옆에 빈칸을 주고 각 칼럼의 검색 조건을 빈칸에 채우고 '검색' 버튼을 누르면, 각 칼럼의 값이 사용자가 입력한 조건을 만족하는 행을 찾아주는 SQL구문을 자동으로 생성해서 쿼리를 실행하는 방식

웹 폼은 사용자가 어떤 필드를 검색할지, 어떤 검색어로 검색할지를 선택할 수 있다. 이러한 기능을 만들려면 SQL 구문의 WHERE 절이 동적으로 변해야만 한다. 동적 SQL에서는 손쉽게 가능한 일이다.

동적 SQL은 현대 프로그래밍 언어에서 가장 많이 사용되는 관계형 데이터베이스 접근 방식이다. 대부분의 현대 프로그래밍 언어들은 데이터베이스 접근을 위한 표준 API를 포함하고 있다. 자바 개발자와 .NET 개발자들은 각각 JDBC와 ADO .NET이라는 표준 API에 익숙할 것이다. 이러한 표준 SQL API는 일반적으로 매우 견고하며 개발자들에게 높은 유연성을 제공해 준다. 다음은 자바에서 동적 SQL을 사용하는 간단한 예제이다.

```
String name;
Date hiredate;
String sql = "SELECT emp_name, hire_date"
      + " FROM employee WHERE emp_num = ? ";
Connection conn = dataSource.getConnection();
PreparedStatement ps = conn.prepareStatement (sql);
ps.setInt (1, 28959);
ResultSet rs = ps.executeQuery();
while (rs.next) {
name = rs.getString("emp_name");
hiredate = rs.getDate("hire_date");
}
rs.close();     | 이 두 라인은 try-catch
conn.close();   | 블록 안에 있어야 한다.
```

의심할 바 없이 동적 SQL은 우리가 예외 처리를 뺐음에도 불구하고 인라인 SQL이나 혹은 저장 프로시저처럼 세련되지 못하다. 이 API는 바로 위의 예제처럼 복잡하고 매우 길게 늘여 써야 한다. 이러한 프레임워크를 사용하면 일반적으로 반복적인 코드들을 양산하게 된다. 게다가 SQL 구문은 한 줄로 보기에 너무 길다. 그러므로 SQL 문자열을 여러 줄로 분리해서 작성하고 연결할 필요가 있다. 연결된 SQL 구문은 가독성이 많이 떨어지고 유지보수와 작업을 어렵게 만든다.

그러면 SQL이 저장 프로시저로 데이터베이스에 있어서도 안 되고, 인라인 SQL로 언어 내부에 둬도 안 되고, 데이터로써 애플리케이션 내에 두어도 안 된다면 어찌하란 말인가? 그래서 우린 이러한 문제들을 모두 피해가기로 했다. 현대 객체 지향 애플리케이션에서는 관계형 데이터베이스와 소통하는 가장 강력한 방법 중의 하나로 객체 관계 매핑(Object Relational Mapping) 도구를 이용한다.

객체 관계 매핑

객체 관계 매핑(ORM)은 SQL을 개발자의 책임 영역에서 완전히 제거함으로써 객체의 영구적인 저장을 단순화하도록 설계돼 있다. 대신 SQL은 자동 생성된다. 어떤 개발 도구들은 컴파일시에 SQL을 정적으로 자동 생성하기도 하지만, 보통은 실행시에 동적으로 자동 생성한다. SQL은 애플리케이션의 클래스와 관계형 데이터베이스 테이블 간의 매핑을 기반으로 하여 생성된다. ORM의 API는 SQL을 없앨 수 있을뿐만 아니라 전형적인 SQL API보다 훨씬 단순하기도 하다. 객체 관계 매핑은 새로운 개념은 아니고 객체 지향(또는 객체 중심의) 프로그래밍 언어만큼이나 오래 전에 등장한 것이다. 최근 몇 년 사이에 ORM을 퍼시스턴스 계층에 대한 매우 강력한 접근 방법이 되도록 힘쓴 결과, 큰 진전을 보았다.

현대 ORM 도구들은 단순히 SQL을 자동 생성하는 것보다 더 많은 일을 한다. 이들은 전체 애플리케이션의 개발 생산성을 향상시키는 완전한 퍼시스턴스 계층 아키텍처를 제공해 준다. 훌륭한 ORM들은 모두 트랜잭션 관리 기능을 제공한다. 또 로컬과 분산 트랜잭션 모두를 제어하는 간단한 API를 포함하고 있다. ORM 도구들은 일반적으로 불필요한 데이터베이스 접근을 피할 수 있도록 여러 다른 종류의 데이터를 다루는 다양한 캐시 전략을 제공한다. ORM 도구가 데이터베이스 접근을 줄이는 다른 방법으로 데이터의 적재 지연(lazy loading) 기법이 있다. 적재 지연은 데이터 사용이 꼭 필요한 바로 그 순간까지 데이터 가져오기를 미룬다.

이러한 기능들에도 불구하고 ORM은 모든 것을 해결해 주는 완벽한 솔루션이 아니다. ORM이 모든 상황에 다 대응할 수는 없다. ORM 도구는 가정과 규칙(assumptions and rules)에 기반하고 있다. 가장 일반적인 가정은 데이터베이스가 적절하게 정규화(normalized) 되어 있다는 것이다. 1.4절에서 논의하겠지만 가장 크고 값비싼 데이터베이스들조차 완벽하게 정규화 되어 있지 못하다. 이로 인해 매핑 하기가 복잡해지고 편법이 필요해지며 설계가 비효율적이 된다. 어떤 객체 관계 솔루션도 모든 데이터베이스 각각에 존재하는 모든 기능과 성능 그리고 설계상의 미비점들을 지원해 줄 수는 없다. 이미 말했듯이 SQL은 완전히 표준화 되어 있지는 않았다. 이런 까닭에 모든 ORM은 항상 특정 데이터베이스가 가진 기능 중 일부만을 지원할 수 있을 뿐이다.

그렇다면 복합적인 솔루션을 한 번 살펴보자.

1.1.2 iBATIS의 장점 이해하기

iBATIS는 여러 솔루션들로부터 최고의 아이디어들을 도출하여 그 시너지를 이끌어낸다. 표 1.1은 위에서 논의한 각 접근 방법 중에서 iBATIS가 차용한 몇몇 아이디어들을 정리한 것이다.

표 1.1 다른 솔루션이 제공하는 것과 유사하게 iBATIS가 제공하는 장점

방법	비슷한 이점	해결된 문제점
저장 프로시저	iBATIS는 SQL을 캡슐화하고 외부로 분리하여 애플리케이션 외부에 SQL을 둔다. iBATIS는 저장 프로시저와 비슷하지만 객체지향적인 API를 제공한다. iBATIS는 또한 저장 프로시저를 직접 호출하는 방법도 지원한다.	비즈니스 로직이 데이터베이스 밖에 있도록 한다. 배포하기 쉽다. 테스트하기 쉽다. 이식성이 좋다.
인라인 SQL	iBATIS는 SQL을 원형 그대로 작성할 수 있다. 문자열 연결이나 파라미터 값 설정, 결과값 가져오기 등이 불필요하다.	iBATIS는 애플리케이션 코드내에 삽입되지 않는다. 전처리기가 불필요하다. SQL의 일부가 아닌 모든 기능을 사용할 수 있다.
동적 SQL	iBATIS는 파라미터를 기반으로 하여 동적으로 쿼리를 구성하는 기능을 제공한다. 쿼리 구성 API는 필요 없다.	iBATIS는 애플리케이션 코드에 섞인 연결된 문자열 블록으로 SQL을 작성할 필요가 없다.
객체/관계 매핑	iBATIS는 ORM 도구의 적재 지연(lazy loading), 조인해서 값 가져오기(join fetching), 실시간 코드 생성, 상속 등과 같은 많은 기능들을 동일하게 제공한다.	iBATIS는 어떠한 데이터 모델과 객체모델의 조합도 사용할 수 있다. 이들을 어떻게 설계 할지에 대한 제약이나 규칙 같은 것은 거의 없다.

이제 iBATIS를 이해했으니 다음 절에서 iBATIS 퍼시스턴스 계층의 중요한 특징 두 가지를 논의해 보자. 바로 SQL의 외부 저장과 캡슐화이다. 이 두 개념이 합쳐져서 iBATIS가 이룩한 많은 가치와 고급 기능들을 제공해 준다.

외부로 뺀 SQL

소프트웨어 개발에 있어 지난 10년간 얻은 지혜 중의 하나는, 한 사람이 개발하는 시스템을 설계할 때에도 그 부속 시스템들을 마치 각자 서로 다른 개발자가 개발하는 것처럼 만드는 것이다. 사용자 인터페이스 설계, 애플리케이션 프로그래밍 그리고 데이터베이스 관리 같은 서로 다른 프로그래밍 역할에 따라 다뤄지는 사항들은 서로 분리하는 것이 좋다. 비록 이러한 역할 모두를 한 사람이 하더라도 이렇게 분리하면 시스템의 특정 부분에 집중할 수 있는 잘 계층화된 설계를 얻을 수 있다. 만약 SQL을 자바 소스 코드에 집어넣으면 데이터베이스 관리자나 혹은 .NET 개발자들이 동일한 데이터베이스에서 작업할 경우 불편하게 된다. 외부로 뺀 SQL은 SQL을 애플리케이션 소스 코드로부터 분리하여 둘 모두를 명확하게 유지할 수 있게 한다. 그렇게 함으로써 SQL은 상대적으로 특정 언어나 플랫폼에 독립적인 상태가 된다. 대부분의 개발 언어들은 SQL을 문자타입으로 나타내어 긴 SQL 구문을 연결해야 한다. 다음의 간단한 SQL 구문을 보자.

```
SELECT
  PRODUCTID,
  NAME,
  DESCRIPTION,
  CATEGORY
FROM PRODUCT
WHERE CATEGORY = ?
```

자바 같은 현대 프로그래밍 언어에서 문자타입으로 SQL 구문을 표현하면 위 예제처럼 미려한 SQL 구문이 많은 언어 특징들로 인해 지저분해지고 관리하기 힘든 코드가 된다.

```
String s = "SELECT"
  + " PRODUCTID,"
  + " NAME,"
  + " DESCRIPTION,"
  + " CATEGORY"
  + " FROM PRODUCT"
  + " WHERE CATEGORY = ?";
```

단순히 FROM 구절 앞의 공백을 잊어버려서 SQL 오류가 발생할 수도 있다. 이 밖에도 SQL이 복잡해질 경우 발생할 수 있는 문제들을 쉽게 생각해낼 수 있을 것이다.

여기서 iBATIS의 핵심 장점을 찾을 수 있다. SQL을 쓰고자 하는 대로 그대로 쓰면 된다는 점이다. 아래는 iBATIS가 SQL 구문을 매핑할 때 어떻게 하는지 보여준다.

```
SELECT
  PRODUCTID,
  NAME,
  DESCRIPTION,
  CATEGORY
FROM PRODUCT
WHERE CATEGORY = #categoryId#
```

SQL이 구조나 단순성 측면에서 변한 것이 없음을 볼 수 있다. 기존 SQL과 가장 다른 점

은 파라미터 #categoryId#의 형태가 특정 언어에 종속적인 세부사항에 따라 달라지는 점 등이다. iBATIS는 이를 더 이식성 좋고 읽기 쉽게 해준다.

이제 SQL을 소스 코드로부터 분리해서 우리가 더욱 자연스럽게 작업할 수 있는 곳으로 옮겼으니, 이것을 애플리케이션과 연결하여 우리가 사용할 수 있는 방식으로 실행시킬 수 있게 해보자.

캡슐화된 SQL

오래된 개념 중 하나로 모듈화가 있다. 절차적인 애플리케이션에서 코드는 여러 파일, 함수 그리고 프로시저로 분리된다. 객체 지향 애플리케이션에서는 코드가 클래스와 메소드로 조직된다. 캡슐화는 코드를 응집성 있는 모듈로 조직하는 것뿐만 아니라 또한 세부적인 구현을 숨기고 호출하는 코드에게 인터페이스만을 노출시키는 모듈화의 한 형태이다.

이러한 개념은 퍼시스턴스 계층으로도 확대 적용할 수 있다. SQL의 입력과 출력을 정의(이게 인터페이스이다)하고 애플리케이션의 다른 부분으로부터 SQL 코드를 숨겨서 SQL을 캡슐화할 수 있다. 당신이 객체지향 소프트웨어 개발자라면 인터페이스를 구현으로부터 분리하는 것과 같은 방식이라고 보면 된다. 당신이 SQL 개발자라면 저장 프로시저 안에 SQL을 숨기는 것이 캡슐화라고 생각하면 된다.

iBATIS는 SQL을 캡슐화하기 위해 XML을 사용한다. 우리가 XML을 선택한 까닭은 일반적으로 모든 플랫폼에 이식가능하고, 산업 전반에 걸쳐 채용되었으며 그 어떤 언어나 파일 포맷보다 SQL처럼 오랫동안 살아남을 수 있을 것 같았기 때문이다.

iBATIS는 XML을 사용해서 SQL 구문의 입력과 출력을 정의한다. 대부분의 SQL 구문은 하나 혹은 그 이상의 파라미터를 가지며 테이블 형태의 결과를 만들어 낸다. 이는 결과가 여러 칼럼과 레코드의 연속으로 이루어짐을 뜻한다. iBATIS는 파라미터와 실행 결과를 객체의 프로퍼티로 매핑하게 한다. 다음 예제를 보자.

```
<select id="categoryById"
  parameterClass="string" resultClass="category">
      SELECT CATEGORYID, NAME, DESCRIPTION
      FROM CATEGORY
      WHERE CATEGORYID = #categoryId#
</select>
```

SQL이 XML 요소로 둘러싸여 있음을 보라. 이것이 SQL을 캡슐화한 것으로 단순한

〈select〉 요소가 SQL 구문의 이름과 파라미터 입력 타입과 결과 출력 타입을 정의하고 있다. 객체지향 소프트웨어 개발자들에게는 이것이 메소드의 시그너처와 비슷하게 보인다. SQL을 외부로 빼고 캡슐화함으로써 간결함과 일관성을 모두 이루어 냈다.

API 사용법과 매핑 문법의 자세한 사항은 2장에서 다룰 예정이다. 그러기 전에 iBATIS가 우리의 애플리케이션 아키텍처의 어느 부분에 적합한지 이해하는 것이 중요하다.

1.2 iBATIS가 적합한 곳

대부분의 잘 설계된 소프트웨어는 계층화된 설계를 사용한다. 계층화된 설계란 애플리케이션의 기술적인 역할들을 응집성 높은 부분들끼리 묶어서 상세한 구현 방법을 특정 기술이나 인터페이스로부터 분리하는 것이다. 계층화된 설계는 견고한(3GL/4GL) 어떤 프로그래밍 언어로도 구현할 수 있다. 그림 1.2는 많은 비즈니스 애플리케이션에서 유용하게 쓸 수 있는 전형적인 계층화된 전략을 상위 레벨 관점에서 보여 준다.

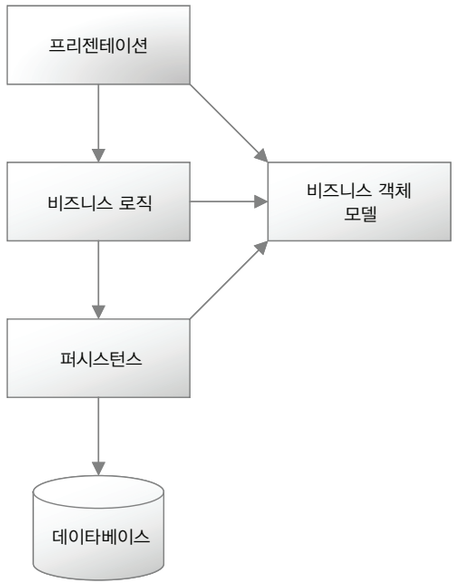


그림 1.2 | 디미터 법칙(Law of Demeter)을 따르는 전형적인 계층화 전략

그림 1.2에서 화살표는 "~에 의존한다", "~를 사용한다" 라고 해석하면 된다. 이러한 계

층화된 접근법은 디미터 법칙(Law of Demeter)[5]에 의해 영향을 받은 것인데, 이것은 다음의 한 문장으로 나타낼 수 있다.

"각 계층은 다른 계층에 대해 오직 제한된 정보만을 가질 수 있다: 오직 현재 계층에 인접한 계층에 대해서만."

이러한 개념은 각 계층이 오직 자기 바로 아래 계층과만 소통할 수 있다는 뜻이다. 이로 인해 의존성이 한 방향으로만 흐르는 것을 보장할 수 있다. 이는 계층화되지 않게 설계한 애플리케이션에서 흔히 나타나는 전형적인 '스파게티 코드'를 피할 수 있게 해준다.

iBATIS는 퍼시스턴스 계층 프레임워크이다. 퍼시스턴스 계층은 애플리케이션의 비즈니스 로직과 데이터베이스 사이에 자리잡고 있다. 이러한 구분은 퍼시스턴스 계층이 비즈니스 로직 코드와 (혹은 그 반대로) 섞이지 않도록 보장하는 데 매우 중요하다. 이러한 구분의 장점은 객체 모델이 데이터베이스 설계와 무관하게 변경될 수 있게 함으로써 코드의 유지보수성을 향상시켜준다.

비록 iBATIS가 퍼시스턴스 계층에 집중돼 있긴 하지만, 애플리케이션 아키텍처의 전체 계층을 이해하는 것도 중요하다. 비록 우리가 어떤 특정한 구현에 최소한으로 의존할 (혹은 전혀 의존하지 않을) 만큼 우리의 관심사를 분리시키더라도 이런 각 계층 간의 상호작용을 전혀 몰라도 된다고 믿는 것은 너무도 순진한 생각이다. 당신이 얼마나 애플리케이션을 잘 설계하든 간에, 각 계층 간에는 우리가 꼭 알아야 하는 간접적인 행위 결합이 존재하게 된다. 다음 절에서는 각 계층들과 iBATIS와 그 계층들이 어떤 관계가 있는지 알아본다.

1.2.1 비즈니스 객체 모델

비즈니스 객체는 애플리케이션의 비즈니스 계층을 제외한 다른 부분들의 토대가 되는 역할을 한다. 비즈니스 객체는 처리할 도메인을 객체 지향적으로 묘사한다. 이 때문에 비즈니스 객체 모델을 이루는 클래스를 도메인 클래스(domain class)라고 부르기도 한다. 다른 모든 계층들은 데이터를 나타내고 특정 비즈니스 로직의 기능을 수행하기 위해서 비즈니스 객체 모델을 사용한다.

5. **역자주** | 데메테르 Demeter 그리스 신화에 나오는 여신. 농업의 여신. 제우스의 누이이자 아내. 디미터 법칙은 '디미터 프로젝트'를 수행하는 과정에서 발견된 법칙이라 그렇게 이름 붙여졌다고 함.

애플리케이션 설계자들은 보통 다른 어떤 것보다도 비즈니스 객체 모델의 설계를 먼저 한다. 상위 레벨에서조차도 시스템에서 사용되는 용어 중 명사들로부터 이끌어낸 이름으로 클래스들을 구분한다. 예를 들면 도서관 애플리케이션에서 비즈니스 객체 모델은 장르(Genre)라고 불리는 클래스를 포함하고 있을 것이고, 그 인스턴스들로 공상과학소설, 미스테리, 어린이 도서 등이 있을 것이다. 또한 책(Book)이라 불리는 클래스에 The Long Walk, The Firm , Curious George 같은 인스턴스 등도 있을 것이다. 애플리케이션이 더 발전해나감에 따라 클래스들이 송장항목(InvoiceLineItem) 같은 좀 더 추상적인 개념을 나타내기 시작할 것이다.

비즈니스 객체 모델 클래스는 약간의 로직을 포함할 수도 있다. 하지만 다른 계층, 특히 프리젠테이션 계층과 퍼시스턴스 계층에 접근하는 코드는 절대로 들어가면 안 된다. 게다가, 비즈니스 객체 모델은 절대로 다른 계층에 의존해서는 안 된다. 다른 계층은 비즈니스 객체를 사용할 수 있지만, 그 반대는 절대로 안 된다.

iBATIS 같은 퍼시스턴스 계층은 보통 비즈니스 객체 모델을 사용하여 데이터베이스에 저장된 데이터를 나타낸다. 퍼시스턴스 계층의 메소드들은 파라미터나 반환 값으로 비즈니스 객체 모델의 도메인 클래스들을 사용한다. 이 때문에 이 클래스들은 때때로 데이터 전송 객체(data transfer object)라고 부르기도 한다. 비록 데이터 전송이 이 클래스들이 하는 유일한 역할은 아니지만, 퍼시스턴스 계층 프레임워크의 입장에서 보면 나름대로 납득할 수 있는 이름이다.

1.2.2 프리젠테이션 계층

프리젠테이션 계층은 애플리케이션의 데이터와 제어 화면을 출력하는 책임을 맡고 있다. 이는 모든 정보의 레이아웃을 지정하고 출력 형식을 정의하는 책임을 맡고 있음을 뜻한다. 요즘 가장 인기 있는 기업용 애플리케이션의 프리젠테이션 계층 접근 방식은 HTML과 자바스크립트를 이용해서 웹브라우저를 통해 보여주는 웹이다.

웹 애플리케이션은 플랫폼에 독립적이고, 배포와 확장이 쉬운 장점을 가지고 있다. 아마존 닷컴은 웹 애플리케이션을 통해 책을 살 수 있는 완벽한 예이다. 단지 책을 구입하기 위해서 모든 사람들이 애플리케이션을 다운로드해서 설치하는 것은 매우 비현실적이라는 점에서 볼 때, 웹 애플리케이션을 사용한 것은 훌륭한 결정이다.

웹 애플리케이션은 높은 수준의 사용자 제어 방식이나 복잡한 데이터 처리가 필요할 경

우 일반적으로 잘 작동하지 않는다. 이런 경우에는 탭, 표, 트리 그리고 내장 객체와 같은 운영 시스템 위젯을 사용하는 리치 클라이언트 방식이 더 낫다. 리치 클라이언트는 훨씬 더 강력한 사용자 인터페이스를 제공해주지만 배포가 다소 어렵고 웹 애플리케이션이 제공하는 성능이나 보안 수준을 제공하려면 더욱 많은 신경을 써야만 한다. 리치 클라이언트 기술의 예로는 자바에서는 Swing이 .NET에서는 WinForms 등이 있다.

최근 들어 두 개념을 합친 복합적인 클라이언트로 웹애플리케이션과 리치 클라이언트의 장점을 갖추려는 시도를 하고 있다. 고급 제어기능을 가진 매우 작은 리치 클라이언트를 웹 브라우저를 통해서 투명하게 사용자의 데스크탑으로 다운로드할 수 있다. 이러한 복합적인 리치 클라이언트는 어떠한 비지니스 로직이나 혹은 사용자 인터페이스가 들어가는 레이아웃조차도 포함하지 않는다. 대신, 애플리케이션의 겉 모양새와 비즈니스 기능등은 웹 서비스를 이용하거나 혹은 리치 클라이언트와 서버 간의 매개체로 XML을 사용하는 웹 애플리케이션을 통해서 처리하게 된다. 이러한 방식의 유일한 단점은 애플리케이션의 개발과 배치에 더 많은 소프트웨어가 필요하다는 점이다. 예를 들어 매크로미디어의 Flex와 Laszlo 시스템의 Laszlo는 매크로미디어의 Flash 브라우저 플러그인을 필요로 한다.

그리고 모든 복합적인 프리젠테이션 계층의 멋진 축소판인 Ajax가 있다. Ajax는 제시 제임스 가렛이 만든 신조어로, 비동기 자바스크립트와 XML의 약어(Asynchronous JavaScript And XML, Ajax)이다. 하지만 사실 꼭 비동기적(asynchronous)이거나 XML을 이용할 필요는 없다. 그래서 요즘엔 Ajax는 간단히 '정말 멋진 자바스크립트를 매우 많이 이용해서 구성한 정말로 리치한 웹 기반 사용자 인터페이스' 정도의 의미로 사용된다. 구글은 그들의 GMail, 구글 맵, 구글 캘린더 애플리케이션에서 Ajax를 잘 활용한 예를 보여주고 있다.

iBATIS는 웹 애플리케이션과 리치 클라이언트 애플리케이션 그리고 복합적인 애플리케이션에서 모두 이용할 수 있다. 비록 일반적으로 프리젠테이션 계층이 퍼시스턴스 계층에 직접적으로 관여하는 경우는 없지만, 사용자 인터페이스에 대한 어떤 결정이 퍼시스턴스 계층에 대한 요구사항에 영향을 끼친다. 예를 들어 5000여 항목의 큰 리스트을 다루는 웹 애플리케이션을 생각해 보자. 우리는 5000개를 한꺼번에 보여주려 하지는 않을 것이고 또한 지금 당장 사용하지도 않을 5000개나 되는 항목을 한 번에 데이터베이스에서 읽어오는 것도 좋은 생각이 아니다. 한 번에 10개 정도씩만 읽어서 보여주는 것이 더 나은 접근 방식이다. 따라서 우리의 퍼시스턴스 계층은 데이터의 일부만을 리턴하는 유연성이 있어야 하고, 우리가 원하는 정확히 10개의 항목만을 선택해서 가져올 수 있는 능력을 제공해 줄 수도 있어야 한다. 이러한 기능이 우리 애플리케이션에 불필요한 객

체 생성과 데이터 가져오기를 막고, 네트워크 트래픽과 메모리 사용량도 줄여주게 된다. iBATIS는 데이터의 특정 범위만을 쿼리하는 기능을 제공해서 이러한 목적을 이룰 수 있도록 도움을 준다.

1.2.3 비즈니스 로직 계층

애플리케이션의 비즈니스 로직 계층은 해당 애플리케이션이 제공하는 포괄적인(coarse grained) 서비스들을 표현한다. 이러한 이유로 가끔 '서비스' 클래스라고 부르기도 한다. 상위 레벨에서는 누구라도 비즈니스 로직 계층의 클래스와 메소드들을 보고 시스템이 무엇을 하는지 이해할 수 있어야 한다. 예를 들어 은행 애플리케이션에서는 비즈니스 로직 계층에 `TellerService`(은행원서비스)라는 클래스가 있을 것이고, 이 클래스는 `openAccount()`(계좌개설), `deposit()`(예금), `withdrawal()`(출금) 그리고 `getBalance()`(잔고조회) 같은 메소드들을 가지고 있을 것이다. 이것들은 데이터베이스와 그리고 아마도 다른 시스템들과도 얽혀 있는 매우 큰 단위의 기능들이다. 따라서 이것들을 도메인 클래스(비즈니스 객체 모델)에 두기에는 너무 무겁고, 만일 도메인 클래스에 두게 되면 코드가 바로 응집성을 잃고(incohesive) 결합도가 높아지며(coupled) 일반적으로 유지보수하기가 힘들어진다. 이에 대한 해결책은 큰 덩어리 비즈니스 기능들을 관련된 비즈니스 객체 모델에서 분리하는 것이다. 객체 모델 클래스를 로직 클래스로부터 분리하는 것을 때때로 명사-동사 분리(noun-verb separation)라고 부르기도 한다.[6]

순수 객체지향 주의자들은 이러한 설계가 메소드를 직접적으로 관련된 도메인 클래스에 두는 것보다 덜 객체 지향적이라고 항변할지도 모른다. 무엇이 더, 혹은 덜 객체지향적이냐를 떠나서 분리하는 방식이 더 나은 설계이다. 주된 이유를 보자면 비즈니스 기능들은 보통 매우 복잡하다. 이들은 한 개 이상의 클래스들을 다루고 데이터베이스, 메시지 큐 그리고 다른 시스템들을 포함해 많은 기반 컴포넌트들을 다룬다. 게다가 종종 하나의 비즈니스 기능이 여러 비즈니스 클래스에 관련되는 일도 발생해서 메소드가 대체 어느 클래스에 속해야 하는지 결정하기 힘든 경우도 생긴다. 이 때문에 포괄적으로 구현된 비즈니스 함수는 비즈니스 로직 계층에 속하는 클래스의 메소드로 따로 분리하여 구현하는

6. **역자주** | noun-verb separation 계좌라는 도메인 클래스가 있을 때, 계좌를개설하다()라는 메소드를 계좌 도메인 클래스가 아니라 은행서비스 비즈니스 로직 클래스에 두고, 계좌 도메인 클래스에는 소유자정보, 계좌 종류, 이자율 같은 정보만을 둔 상태에서, 은행서비스.계좌를개설하다(신규계좌의인스턴스); 와 같이, 계좌라는 도메인 클래스로부터 계좌 개설 로직을 분리해서 은행서비스 클래스에 따로 두고 도메인 클래스를 파라미터로 주어 로직을 수행하는 방식.

것이 낫다. 상세하게 구현된(fine grained) 비즈니스 로직을 관련된 도메인 클래스에 두는 것은 상관없다. 비즈니스 계층의 포괄적으로 구현된 서비스 메소드는 도메인 클래스에 있는 더 상세하게 구현된 순수 로직 메소드를 호출해 사용할 수 있다.[7]

계층화된 아키텍처에서 비즈니스 로직 계층은 퍼시스턴스 계층 서비스를 사용하는 측이 된다. 비즈니스 계층은 데이터를 가져오거나 변경하기 위해서 퍼시스턴스 계층을 호출한다. 비즈니스 로직 계층은 트랜잭션을 구분짓는 훌륭한 장소이기도 하다. 왜냐하면 여러 가지의 서로 다른 사용자 인터페이스나 웹 서비스 같은 다른 인터페이스들을 사용하는 큰 덩어리 함수를 비즈니스 로직 계층에서 정의하기 때문이다. 트랜잭션 구분에 대해서는 다른 견해를 가진 사람들도 있겠지만 이에 대해서는 8장에서 더 살펴보기로 하자.

1.2.4 퍼시스턴스 계층

퍼시스턴스 계층이 바로 iBATIS가 있을 곳이다. 그러므로 이 책은 여기에 중점을 둘 것이다. 객체지향 시스템에서 퍼시스턴스 계층의 주된 관심사는 저장소와 객체 가져오기, 또는 더 자세히 말하면 객체에 저장된 데이터라고 할 수 있겠다. 기업용 애플리케이션에서 퍼시스턴스 계층은 데이터를 저장하기 위해 주로 관계형 데이터베이스 시스템과 소통한다. 하지만 어떤 경우에는 다른 종류의 영구적인 저장이 가능한 데이터 구조나 매체들을 사용할 수도 있다. 어떤 시스템은 단순히 쉼표로 분리된 텍스트 파일이나 XML 파일을 이용할 수도 있다. 기업용 애플리케이션에서 사

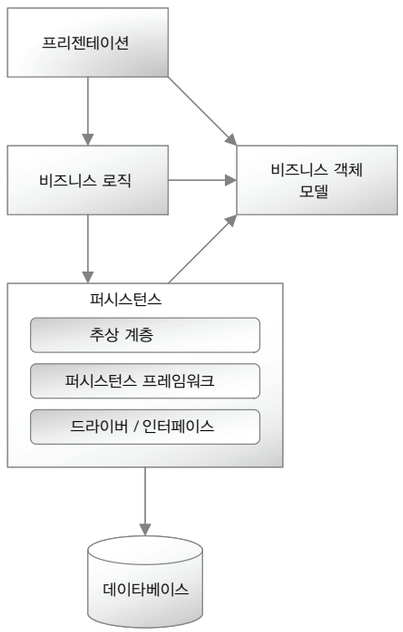


그림 1.3 | 퍼시스턴스 계층의 내부적인 계층적 설계

7. **역자주 |** coarse-grained와 fine-grained의 의미를 '포괄적으로 구현된'과 '상세히 구현된'으로 번역했는데, 이 말은 번역하기가 매우 까다롭다. coarse-grained는 듬성 듬성 탈곡된 곡식을 의미하고, fine-grained는 곱게 탈곡된 곡식을 의미한다. 이것을 IT 용어에 도입해서 coarse-grained method라는 것은 포괄적인 업무 단위를 수행하는 메소드를 의미하고, fine-grained method는 세부적인 업무 단위를 수행하는 메소드를 의미한다. 예를 들어 계좌이체를 생각해 보면, 은행서비스.계좌이체(이체정보);라는 메소드가 coarse-grained method이다. 이 메소드를 호출하면, 실제로는 계좌이체권한이있는사용자인지검사(), 이체금액이적합한지검사(), 받는사람의계좌가존재하는지검사(), 실제계좌이체()와 같은 세부적인 작업들이 실행된다. 이 세부적인 작업들을 fine-grained method라고 보면 될 것 같다.

용하는 데이터 저장 방식이 서로 공통점이 없고 다양하기 때문에 퍼시스턴스 계층의 두 번째 고려 사항으로 추상화가 필요하게 된다. 퍼시스턴스 계층은 데이터가 어떻게 저장되고 어떻게 전송되는지에 대한 세부 사항을 숨겨야 한다. 그러한 세부 사항들은 애플리케이션의 다른 계층으로 노출돼서는 안 된다.

이러한 사항들과 그것들이 어떻게 관리되는지에 대한 이해를 돕기 위해, 퍼시스턴스 계층을 세 가지 기본이 되는 부분으로 나눠보면 그림 1.3에서 볼 수 있는 것처럼 추상 계층, 퍼시스턴스 프레임워크 그리고 드라이버 혹은 인터페이스가 된다. 세 가지 부분을 좀 더 자세히 살펴보자.

추상 계층

추상 계층의 역할은 퍼시스턴스 계층에 일관성 있고 의미 있는 인터페이스를 제공해 주는 것이다. 이것은 클래스와 메소드의 집합으로 퍼시스턴스 구현의 세부 사항에 대한 퍼사드(façade–실제 구현 메소드를 감싸서 꾸며줌)의 역할을 한다. 추상 계층에 소속된 메소드는 특정 구현에 종속적인 파라미터를 요구하면 안 되고 특정 퍼시스턴스 구현에 종속적인 값을 리턴하거나 예외를 던져서도 안 된다. 적절한 추상 계층이라면 추상 계층을 수정하거나 추상 계층에 의존하는 다른 계층을 수정하는 일 없이 전체적인 퍼시스턴스 접근 방식(퍼시스턴스 API와 기반 저장소를 포함)을 변경하는 것이 가능해야 한다. 적절한 추상 계층 구현을 도와주는 패턴들이 존재하며 데이터 접근 객체(DAO[Data Access Object])패턴이 그 중 가장 많이 쓰인다. 이 패턴에 대한 프레임워크도 있으며, iBATIS도 이 패턴을 구현하고 있다. 11장에서 iBATIS DAO 프레임워크를 살펴볼 것이다.

퍼시스턴스 프레임워크

퍼시스턴스 프레임워크는 드라이버(혹은 인터페이스)와 소통하는 책임을 맡고 있다. 퍼시스턴스 프레임워크는 데이터를 저장하고 가져오고 수정하고 검색하고 관리하는 메소드들을 제공할 것이다. 추상 계층과는 달리 퍼시스턴스 프레임워크는 일반적으로 기반 저장소의 클래스에 종속적이다. 예를 들어 데이터를 저장하기 위해 XML 파일만을 전문적으로 다루는 퍼시스턴스 API가 있을 수 있다. 대부분의 인기 있는 프로그래밍 언어들은 관계형 데이터베이스를 다루는 표준 API를 가지고 있다. JDBC는 자바 애플리케이션에서 데이터베이스에 접근하는 표준 프레임워크이고 ADO .NET은 .NET 애플리케이션을 위한 표준 데이타베이스 퍼시스턴스 프레임워크이다. 표준 API들은 그 구현 결과물이 매우 완성도

가 높고 일반적인 목적으로 사용할 수 있다. 하지만 이를 사용할 때는 반복적이고 부차적인 코드가 많이 필요하다. 이러한 이유로 표준 프레임워크를 기반으로 하여, 보다 특수한 상황에서 보다 강력한 기능들로 확장한 많은 프레임워크들이 생겨났다. iBATIS는 자바와 .NET에서 일관성 있는 접근법을 통해 모든 관계형 데이터베이스를 전담하여 다루는 퍼시스턴스 프레임워크이다.

드라이버 혹은 인터페이스

기반 저장소는 쉼표로 분리된 텍스트 파일처럼 단순할 수도 있고, 수백만 달러의 기업용 데이터베이스 서버처럼 복잡할 수도 있다. 이 두 경우 모두 하위 레벨에서 기반 저장소와 통신하는 소프트웨어 드라이버가 사용된다. 네이티브 파일 시스템 드라이버처럼 기능은 매우 일반적이지만 플랫폼에 종속적인 드라이버들도 있다. 아마도 파일 I/O 드라이버를 직접 볼 일은 없겠지만 그것이 존재한다는 사실은 확실히 알고 있을 것이다. 한편 데이터베이스 드라이버는 매우 복잡한 편이고 구현, 규모 그리고 행동 등이 서로 다르다. 퍼시스턴스 프레임워크가 하는 일은 드라이버와 통신하여 드라이버들 간의 차이점들을 최소화하고 간소화하는 것이다. iBATIS는 오직 관계형 데이터베이스만을 지원하기 때문에 관계형 데이터베이스를 중점적으로 살펴볼 것이다.

1.2.5 관계형 데이터베이스

iBATIS는 관계형 데이터베이스에 쉽게 접근하기 위해 만들어진 것이다. 데이터베이스는 제대로 사용하려면 수많은 일을 해줘야 하는 복잡하기 짝이 없는 녀석이다. 데이터베이스는 데이터를 관리하고 변경하는 책임을 맡고 있다. 텍스트 파일 대신 데이터베이스를 사용하는 것은 데이터베이스가 주로 무결성, 성능 그리고 보안적인 영역에서 아주 많은 이점을 제공하기 때문이다.

무결성

무결성은 아마도 가장 중요한 이점일 것이다. 무결성을 보장하지 않는다면 다른 장점들은 별 의미가 없게 된다. 데이터가 일관성이 없고 믿을만하지 못하고 값이 틀리다면 우리에게 별 가치가 없거나 혹은 아예 쓸모가 없기도 하다. 데이터베이스는 강력한 데이터 타입 제약조건 엄수 그리고 트랜잭션 보장 등을 통해 무결성을 보장한다.

데이터베이스는 데이터 타입을 엄격하게 준수한다. 이것은 데이터베이스 테이블이 만들어질 때 그 칼럼이 특정 데이터 타입을 저장하도록 설정돼야 함을 의미한다. DBMS는 테이블에 저장된 데이터가 각 칼럼 타입을 준수한다는 것을 확실하게 보장한다. 예를 들어 테이블은 어떤 칼럼이 VARCHAR(25) NOT NULL이라고 정의할 수 있다. 이 타입은 그 값이 문자열 데이터이고, 그 길이가 25를 넘지 않음을 보장한다. 정의의 NOT NULL 부분은 데이터가 꼭 존재해야 하고 해당 칼럼에 그 값이 꼭 제공되어야 함을 의미한다.

데이터 타입 엄수뿐만 아니라 테이블에 다른 제약 조건도 줄 수 있다. 이러한 제약 조건은 보통 범위가 넓어 한 개 이상의 칼럼을 다루게 된다. 제약 조건은 보통 여러 레코드나 혹은 여러 테이블의 유효성 검증에 관계한다.

제약조건의 한 형태로 UNIQUE 제약 조건이 있는데, 이것은 테이블의 해당 칼럼에 특정한 값이 오직 한 번만 나올 수 있음을 의미한다. 다른 종류의 제약 조건으로 외래키(FOREIGN KEY) 제약 조건은 테이블의 한 칼럼에 있는 값이 다른 테이블의 비슷한 칼럼에 존재하는 값과 같아야 함을 의미한다. 외래키 제약 조건은 테이블 간의 관계를 표현하기 위해 사용하며 이것은 관계형 데이터베이스 설계와 데이터 무결성에 필수적인 요소이다.

데이터베이스가 무결성을 관리하는 가장 중요한 방법 중의 하나로 트랜잭션을 이용하는 것이 있다. 대부분의 비즈니스 기능들은 서로 다른 종류의 데이터를 아마도 서로 다른 데이터베이스로부터 가져다 사용할 것이다. 일반적으로 이 데이터들은 어떤 연관 관계를 가지고 있으며 그 때문에 일관성있게 수정해야만 한다. 트랜잭션을 사용하면 DBMS는 모든 관련된 데이터들이 일관성 있는 방식으로 수정됨을 보장할 수 있다. 더욱이 트랜잭션을 사용하면 시스템의 여러 사용자들이 충돌 없이 동시에 데이터를 수정할 수 있게 된다. 트랜잭션에 대해서는 알아야 할 것들이 훨씬 많이 있다. 이에 대해서는 8장에서 좀 더 자세히 살펴보겠다.

성능

관계형 데이터베이스를 이용하면 텍스트 파일로는 쉽게 도달하기 어려운 높은 수준의 성능 향상을 이뤄낼 수 있다. 하지만 데이터베이스의 성능은 공짜가 아니며 매우 오랜 시간과 경험을 통해서만 얻을 수 있는 것이다. 데이터베이스 성능은 세 가지 핵심 요인인 설계, 소프트웨어 튜닝 그리고 하드웨어에 의하여 좌우된다.

데이터베이스 성능 향상을 위한 첫 번째 고려 사항은 설계이다. 나쁜 관계형 데이터베이스 설계는 그 비효율성이 지대하여 어떠한 소프트웨어 튜닝이나 하드웨어 증설로도 이것을 바로잡을 수 없다. 나쁜 설계는 데드락, 기하급수적으로 증가하는 (테이블과 테이블 간의) 관계 계산 혹은 단순히 수백만 행의 레코드를 읽어야 하는 사태 등을 일으킨다. 올바른 설계는 매우 깊이 고려해야 하는 사항이며 이에 대해서는 1.3절에서 더 이야기 할 것이다.

소프트웨어 튜닝은 대용량 데이터베이스에서 성능 향상을 위해 두 번째로 중요하게 고려해야할 사항이다. 관계형 데이터베이스 관리 시스템을 튜닝하려면 특정 RDBMS 소프트웨어 사용에 대해 잘 교육받고 경험이 많은 사람이 필요하다. 비록 여러 제품들 간에 공통적으로 적용되는 몇몇 RDBMS의 특징들이 있긴 하지만, 일반적으로 각 제품은 나름대로 복잡한 사정이 있고 서로 약간씩의 차이가 있어서 특정 소프트웨어를 위한 전문가가 있어야 한다. 성능 튜닝은 큰 이익을 줄 수 있다. 데이터베이스 인덱스 하나만 제대로 튜닝해도 복잡한 쿼리를 몇 분이 아니라 몇 초만에 수행하게 할 수 있다. RDBMS에는 캐시, 파일 관리자, 다양한 인덱스 알고리즘 그리고 운영체제등 고려해야할 많은 부분들이 있다. 동일한 RDBMS 소프트웨어도 운영체제가 바뀌면 행동 방식이 바뀌고, 거기에 맞춰 다른 방식으로 튜닝해야 한다. 말할 것도 없이 데이터베이스 소프트웨어를 튜닝할 때는 많은 사항들을 고려해야 한다. 정확히 무엇을 고려해야 할지는 이 책의 범위를 벗어난다. 하지만 튜닝이 데이터베이스 성능 향상에 가장 중요한 요소 중의 하나임을 아는 것이 매우 중요하다. DBA와 함께 일하자!

대용량 관계형 데이터베이스 시스템은 보통 컴퓨터 하드웨어를 많이 요구한다. 이런 까닭에 회사 내의 가장 강력한 서버가 데이터베이스 서버인 경우가 보통이다. 많은 회사에서 데이터베이스는 가장 중요한 영역이다, 그러므로 데이터베이스 하드웨어에 큰 돈을 투자하는 것은 당연하다. 고속 디스크 어레이, 고속 I/O 컨트롤러, 고속 하드웨어 캐시 그리고 네트워크 인터페이스 등은 모두 대용량 데이터베이스 관리시스템의 성능에 중요한 요소들이다. 하지만 고성능 하드웨어만 믿고 품질이 떨어지는 데이터베이스 설계나 RDBMS 튜닝을 게을리하지는 말라는 말이 있다. 하드웨어 증설을 성능 문제를 푸는데 사용해서는 안되고 성능 요구사항을 만족시키는데 사용하여야 한다. 하드웨어에 관한 더 이상의 논의는 이 책의 범위를 벗어난다. 하지만 여러분이 대용량 데이터베이스 시스템으로 작업할 때는 중요하게 고려해야만 한다. 다시 한 번 말하지만, DBA와 함께 일하자!

보안

관계형 데이터베이스 관리 시스템은 또한 보안 측면에서 더 많은 이점을 제공해 준다. 우리가 매일 업무에서 사용하는 많은 데이터는 기밀에 속한다. 최근 몇 년간 보안이 일반화 되면서 프라이버시가 큰 관심사로 떠올랐다. 이런 까닭에 비록 사람의 이름처럼 단순한 것 조차도 기밀로 간주하게 되는데, 이름이 잠재적으로 '유일한 식별 정보'가 될 수도 있기 때문이다. 주민등록번호나 신용카드 번호 등의 다른 정보는 강력한 암호화 등을 통해 더 높은 보안 레벨로 보호해야만 한다. 대부분의 상업용 품질을 갖춘 관계형 데이터베이스는 데이터 암호화 뿐만 아니라 매우 세세한 보안이 가능한 고급 보안 기능들을 갖추고 있다. 각 데이터베이스는 자기만의 보안 요구사항을 갖고 있을 것이다. 그 사항들을 이해하는 것은 매우 중요하다. 애플리케이션 코드가 데이터베이스의 보안 정책을 약화시켜서는 안 되기 때문이다.

서로 다른 데이터베이스는 서로 다른 수준의 무결성, 성능 그리고 보안을 제공할 것이다. 일반적으로 데이터베이스의 크기, 데이터의 값, 그리고 이 데이터베이스에 의존하는 것들이 얼마나 많은가 등에 의해 이 수준이 결정된다. 다음 절에서 서로 다른 데이터베이스의 형태들을 알아보자.

1.3 여러 종류의 데이터베이스로 작업하기

모든 데이터베이스들이 비싼 데이터베이스 관리 시스템이나 고성능의 하드웨어를 필요로 할 만큼 복잡한 것은 아니다. 어떤 데이터베이스는 벽장에 숨겨둔 오래된 데스크탑 장비로 돌려도 충분할 만큼 소규모일 수도 있다. 모든 데이터베이스는 서로 다르다. 데이터베이스들은 서로 다른 요구사항과 서로 다른 목적을 가지고 있다. iBATIS는 거의 모든 관계형 데이터베이스를 지원하지만, 항상 여러분이 작업하는 데이터베이스가 어떤 종류인지 이해하고 있어야 한다.

데이터베이스는 데이터베이스 설계나 규모보다는 다른 시스템들과의 관계에 따라 분류한다. 하지만 데이터베이스의 설계나 규모도 또한 관계가 어떠냐에 의해 좌우될 수 있다. 데이터베이스의 설계나 규모에 영향을 미치는 또 다른 요소로 그 데이터베이스가 얼마나 오래 되었는가(age)가 있을 수 있다. 시간이 지남에 따라 데이터베이스는 여러 방식으로 변하는 경향이 있고, 데이터베이스의 변화 방식이 적절한 형태가 되지 못하는 경우도 있다. 이 절에서는 애플리케이션 데이터베이스, 기업용 데이터베이스, 독점적(proprietary) 데이터베이스, 그리고 레거시(legacy) 데이터베이스 등 네 가지 형태의 데이터베이스에 대해 살펴본다.

1.3.1 애플리케이션 데이터베이스

애플리케이션 데이터베이스는 일반적으로 가장 작고 단순하며 작업하기 가장 쉬운 편이다. 이런 데이터베이스는 일반적으로 우리 개발자들이 작업하길 그리 꺼려하지 않으며 아마도 작업하길 원하기도 한다. 애플리케이션 데이터베이스는 보통 해당 프로젝트의 일부로써 애플리케이션과 나란히 설계하고 구현된다. 이런 까닭에 일반적으로 설계 측면에서 더욱 자유롭고, 특정 애플리케이션에 적합하게 설계할 여력도 충분하다. 애플리케이션 데이터베이스는 외부의 영향은 거의 받지 않고, 대개 1~2개의 인터페이스만이 존재할 뿐이다. 첫 번째 인터페이스는 애플리케이션을 위한 것이고 두 번째는 아마도 Crystal Report 같은 단순한 리포팅 프레임워크나 툴 정보일 것이다. 그림 1.4은 매우 높은 레벨에서 본 애플리케이션 데이터베이스와 다른 시스템들과의 관계를 보여준다.

그림 1.4 | 애플리케이션 데이터베이스 관계

애플리케이션 데이터베이스는 때로는 애플리케이션과 같은 서버에 배치해도 될 만큼 작을 수도 있다. 애플리케이션 데이터베이스는 인프라스트럭처 구성에서도 더 많은 자유를 누릴 수 있다. 작은 애플리케이션 데이터베이스의 경우에는 일반적으로 회사가 Oracle이나 SQL Server 같은 데다 돈을 쓰는 대신 MySQL이나 PostgreSQL 처럼 더 싼 오픈소스 RDBMS 솔루션을 이용할 수 있다. 어떤 애플리케이션의 경우에는 애플리케이션 자체가 돌고 있는 동일한 가상 머신 환경에서 함께 작동하는 내장 애플리케이션을 사용해서 아예 분리된 RDBMS가 불필요해지는 경우까지도 있다.

iBATIS는 애플리케이션 데이터베이스의 퍼시스턴스 프레임워크로서의 역할을 아주 잘 수행한다. iBATIS의 단순함 때문에 개발팀은 애플리케이션 개발에 박차를 가할 수 있을 것이다. 단순한 데이터베이스의 경우에는 RDBMS에 포함된 관리 도구를 이용해 데이터베이스 스키마에서 SQL을 생성하는 것도 가능할 수 있다. 그리고 iBATIS SQL Map 파일을 모두 생성해 주는 개발도구도 있다.[8]

8. **역자주** | Abator, http://ibatis.apache.org/abator.html

1.3.2 기업용 데이터베이스

기업용 데이터베이스는 애플리케이션 데이터베이스보다 규모가 크고 외부의 영향도 훨씬 많이 받는다. 기업용 데이터베이스는 의존하는 다른 시스템들과의 관계가 더욱 복잡하다. 이런 복잡한 관계를 맺게 되는 것들에는 웹애플리케이션이나 리포팅 툴이 있을 수 있고, 이것들 또한 다른 시스템이나 데이터베이스와 복잡한 관계를 갖고 있을 수 있다. 기업용 데이터베이스는 외부 시스템과 인터페이스하는 종류가 많을 뿐만 아니라, 인터페이스가 작동하는 방식도 역시 다양하다. 어떤 것은 밤마다 일괄적인 데이터 적재 작업을 할 수도 있고, 어떤 경우에는 실시간의 트랜잭션을 수행하는 인터페이스일 수도 있다. 이 때문에 기업용 데이터베이스 자체는 사실상 한 개 이상의 데이터베이스를 조합한 경우도 있다. 그림 1.5는 기업용 데이터베이스를 상위 레벨에서 그린 것이다.

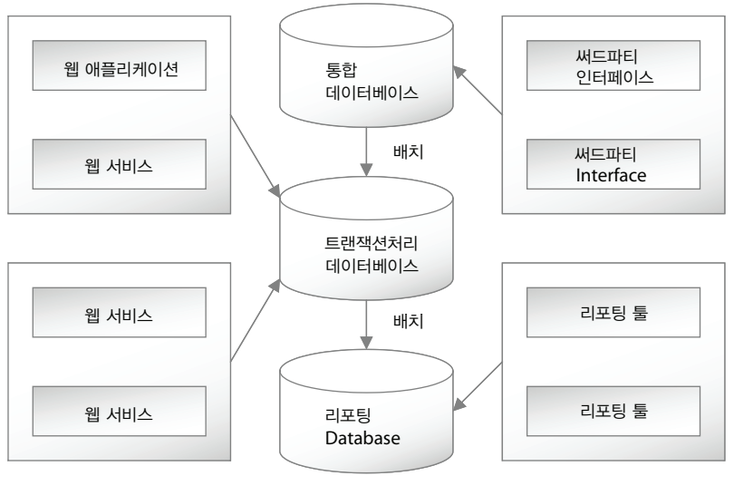


그림 1.5 | 기업용 데이터베이스 아키텍처의 예제

기업용 데이터베이스는 그 설계와 사용에 더 많은 제약 사항들을 내포하고 있다. 무결성, 성능 그리고 보안의 측면에서 신경써야 할 점들이 더욱 많다. 이런 까닭에 기업용 데이터베이스가 때로는 관심사와 요구사항을 분리하기 위해 종종 여러 개의 데이터베이스로 쪼개진다. 만약 한 개의 데이터베이스로 기업용 시스템의 모든 요구 사항을 수용하려고 한다면, 그것은 매우 비싸고 복잡하거나 전혀 실용적이지 못하거나 혹은 사실상 불가능한 시도가 될 뿐이다.

그림 1.5의 예제에서는 요구사항을 수평적으로 비기능적 요구사항에 따라 나누었다. 즉 데이터베이스를 통합하는 관점, 실시간 트랜잭션 관점 그리고 리포팅하는 관점으로 나눈 것이다. 통합 데이터베이스와 리포팅 데이터베이스 둘 다 일괄 적재 방식(batch load)으로 트랜잭션 시스템과 소통한다. 이것은 이 시스템에서 리포팅이 정확히 실시간으로 이뤄질 필요는 없음을 의미하고, 트랜잭션 데이터베이스는 단지 정기적으로 제 3의 시스템으로부터 데이터를 적재하면 된다는 것을 의미하기도 한다. 이를 통해 얻는 장점은 트랜잭션 시스템이 커다란 부하의 부담을 덜었고, 데이터베이스 설계 또한 단순해질 수 있다는 점이다. 일반적으로 통합, 트랜잭션 그리고 리포팅을 단일한 데이터베이스로 설계하는 것은 효과적이지 못하다. 각각 역할에 대해 최적의 성능과 설계를 보장해 주는 패턴은 모두 따로 있다. 그렇더라도 실시간 통합과 리포팅 기능을 요구하는 시스템이 있을 수도 있다. 이러한 경우에는 수평적 방식의 설계는 제대로 작동하지 않는다. 이럴 때는 기업용 데이터베이스를 비즈니스 기능에 따라 수직적으로 나누는 것이 나을 것이다.

기업용 데이터베이스가 어떻게 설계되었든지 간에 애플리케이션 데이터베이스와 기업용 데이터베이스의 차이점을 쉽게 알아볼 수 있다. 애플리케이션 데이터베이스를 효율적으로 사용하고, 동일한 데이터베이스를 사용하는 다른 애플리케이션들과 조화를 잘 이루도록 하려면 운영 환경의 특정 제약 사항이 무엇인지 정확하게 이해하고 있어야 한다.

iBATIS는 기업용 데이터베이스 환경에서도 매우 잘 작동한다. iBATIS에는 복잡한 데이터베이스 설계와 대용량 데이터를 최적으로 다룰 수 있게 도와주는 많은 기능들이 있다. iBATIS는 다중 데이터베이스와도 잘 작동하며 어떤 형태의 객체도 단 하나의 데이터베이스에서 가져온다고 가정하지 않는다. 또한 다중 데이터베이스가 단일 트랜잭션으로 묶이는 시스템처럼 복잡한 트랜잭션도 지원한다. 게다가 iBATIS는 실시간 트랜잭션 시스템뿐만 아니라 리포팅과 통합 시스템 구현에도 매우 잘 작동한다.

1.3.3 독점적 데이터베이스(Proprietary Database)

소프트웨어 개발 부문에서 얼마간 일했다면 당연히 '직접 제작 대 구입' 논쟁을 들어봤을 것이다. 이것은 비즈니스 문제를 해결하기 위해 솔루션을 직접 제작해야 하는가 혹은 이미 그 문제를 해결했다고 하는 패키지 소프트웨어를 구입하느냐 하는 것에 대한 논쟁이다. 비용은 거의 비슷하지만(그렇지 않다면 논쟁거리도 되지 못했을 것이다) 실제 트레이드 오프[9]

9. **역자주** | (동시에 달성할 수 없는 몇 개 조건을 취사 선택하여) 균형[평균]을 취하는 일

는 구현에 드는 시간과 문제에 대한 최적의 해결책인가라는 점에 달려있다. 직접 제작한 소프트웨어는 비즈니스의 요구에 정확히 부합하도록 만들 수 있지만 구현하는 데 더 많은 시간이 들게 된다. 패키지 소프트웨어는 비교적 빨리 구현할 수 있지만 때로는 요구사항에 정확하게 부합하지 않을 때도 있다. 그런 이유로 패키지 소프트웨어를 구입하기로 결정할 때 기업은 보통 양쪽 측면의 모든 장점을 얻기 위해서 해당 소프트웨어에 없는 기능들도 확장할 수 있는지, 소프트웨어의 독점적 영역까지 파고들어서 조사한다.

우리는 이런 상황이 얼마나 끔찍한 결과를 부르는지 논의해 볼 수도 있지만, 그냥 독점적 데이터베이스는 제 3의 소프트웨어에 의해 제어되도록 만들어진 것이 아님을 이해하는 수준에서 그치는 것이 나을 것 같다. 설계할 때 너무도 많은 가정과 제약 사항, 비표준 데이터 타입들 그리고 다른 경고의 징후들을 "스스로 위험 부담을 지도록 하시오"라는 의미로 받아들여야 한다. 이러한 경고에도 불구하고 어떤 기업들은 고작 몇 달러 아껴보자고 놀라운 짓들을 할 것이다. 그로 인해 소프트웨어 개발자들은 독점적 데이터베이스의 정글을 헤매는 데 시간을 허비하게 된다.

iBATIS는 독점적 데이터베이스와 작업하는 데 매우 훌륭한 퍼시스턴스 계층의 역할을 한다. 종종 그런 데이터베이스들은 읽기만 가능한데, 그럴 때 개발자들은 특정 SQL만 실행 가능하게 제한함으로써 iBATIS 사용에 자신감을 가질 수 있게 된다 iBATIS는 개발자가 명시적으로 수정을 요청하지 않았는데 몰래 데이터를 수정하는 일은 하지 않는다. 만약 수정이 필요할 경우에도 독점적 데이터베이스는 데이터의 구조를 까다롭게 제한한다. iBATIS는 그런 경우에 맞는 특별한 업데이트 구문을 사용할 수 있게 해준다.

1.3.4 레거시 데이터베이스(Legacy Database)

만약 현대 객체지향 애플리케이션 개발자들을 파멸로 몰아갈 수 있는 무언가가 있다면, 그것은 바로 레거시 데이터베이스일 것이다. 레거시 데이터베이스는 일반적으로 선사 시대의 기업용 데이터베이스가 아직도 남아 있는 것이라 보면 된다. 이것들은 기업용 데이터베이스의 복잡성과 뒤죽박죽인 설계 그리고 의존성 등을 모두 갖추고 있다. 이에 더하여 수년간에 걸친 수정과 긴급한 수리 작업, 오류 은폐작업, 오류 피해가기, 땜질식 문제 해결 그리고 기술적 제약 등의 전투에서 입은 상처들로 뒤덮여 있다. 게다가 레거시 데이터베이스는 종종 오래되었을 뿐만 아니라 아예 유지보수 지원도 되지 않는 매우 오래된 플랫폼에서 구현되었을 수도 있다. 이들은 현대의 개발자들이 작업할 수 있는 적합한 드라이버가 없을 수도 있다.

iBATIS는 이런 레거시 데이터베이스에도 도움이 될 수 있다. 작업할 시스템에 맞는 드라이버가 존재하는 한 iBATIS는 다른 어떤 데이터베이스들과도 동일한 방식으로 작업을 진행할 수 있게 해준다. 사실 iBATIS는 레거시 데이터베이스를 다루는 최적의 퍼시스턴스 프레임워크 중의 하나일 것이다. 왜냐면 iBATIS는 데이터베이스 설계에 대해 어떠한 가정도 하지 않기 때문에, 최악의 레거시 데이터베이스 설계에서조차도 잘 작동한다.

1.4 iBATIS는 데이터베이스의 공통적인 문제점들을 어떻게 다루나?

현대 소프트웨어 프로젝트에서 데이터베이스는 종종 레거시 컴포넌트 취급을 받기도 한다. 데이터베이스는 기술적인 측면과 비기술적인 측면 모두에서 다루기 어려운 존재라는 역사적인 인식이 있다. 대부분의 개발자들은 아마도 전체 데이터베이스를 완전히 재구축하고 다시 시작했으면 하는 희망을 품을 때도 있을 것이다. 데이터베이스를 그대로 둬야 한다면, 어떤 개발자는 DBA에게 이를 책임지게 하고 어디 방파제로라도 가서 길고 긴 산책을 하고 싶은 심정이 들지도 모른다. 이런 경우는 실용적이지도 않고 일어날 확률도 거의 없다. 믿거나 말거나 보통 데이터베이스가 그런 식으로 존재하게 되는 데는 이유가 있다. 비록 그 이유가 그다지 맘에 들지 않더라도... 아마도 그건 데이터베이스를 변경하기에는 비용이 너무 많이 들거나 변경을 가로막는 다른 의존관계들이 있기 때문일 것이다. 데이터베이스가 왜 잘못돼 있는지와는 무관하게, 우리는 비록 잘못된 데이터베이스라 하더라도 모든 데이터베이스를 효율적으로 다룰 수 있는 법을 터득해야 한다. iBATIS는 매우 복잡하게 혹은 형편없게 설계된 데이터베이스와도 잘 작동하도록 개발되었다. 다음 절에서는 몇몇 잘못된 데이터베이스의 일반적인 예와 iBATIS가 이를 어떻게 극복하는지를 보여준다.

1.4.1 소유권과 제어권

현대 기업용 시스템 환경에서 데이터베이스를 사용함에 있어 첫 번째이자 가장 어려운 점은 전혀 기술적인 문제가 아니다. 이것은 대부분의 기업들이 데이터베이스에 대한 소유권과 권한을 애플리케이션 개발팀에서 분리해 놓는다는 단순한 사실이다. 데이터베이스는 종종 기업 내부에서 분리된 전혀 다른 조직이 관리하곤 한다. 여러분이 운이 좋다면 이 조직이 소프트웨어 출시를 돕기 위해 개발팀과 함께 일할 수도 있다. 운이 좋지 않다면 데이터베이스 관리 조직과 여러분의 프로젝트 팀 사이에 벽이 존재하며, 벽 너머로 요구사항을 넘겨 보내고는 그들이 요구사항을 잘 받아 이해할 수 있기를 기대하는 수밖에 없는 경우도 있다. 슬픈 사실이지만 항상 일어나는 일이다.

데이터베이스 팀은 때론 함께 일하기 어려운 사람들이다. 주된 이유는 그들이 커다란 압력을 받고 있으며, 또한 한 개 이상의 다른 프로젝트 팀과 함께 일하기 때문이다. 그 들은 종종 여러 개의, 때로는 서로 상충되는 요구 사항을 다룰 때도 있다. 데이터베이스 관리는 매우 어려운 작업이며 많은 회사들은 이것을 매우 중요한 사항으로 다룬다. 기업의 데이터베이스에서 오류가 나면, 회사의 경영진이 이 사실을 통보받을 것이다. 그래서 데이터베이스 관리팀은 매우 조심스러워지게 된다. 데이터베이스를 변경하는 것이 애플리케이션을 변경하는 것보다 훨씬 더 엄격할 수도 있다. 데이터베이스에 대한 어떤 변경은 데이터 이전작업(migration)을 필요로 할 수도 있다. 다른 어떤 변경들은 성능에 나쁜 영향을 끼치지 않음을 보증하기 위한 주요한 테스트들을 필요로 할 수도 있다. 데이터베이스 팀과 함께 일하는 것이 어려운 것은 사실 그에 합당한 이유가 있는 것이니까 그들을 이해하고 조금이라도 도와주는 것이 좋을 것이다.

iBATIS는 데이터베이스 설계와 상호 작용에 대해 높은 유연성을 보여준다. DBA들은 실제 작동하는 SQL을 보길 원하고 또한 그들은 매우 복잡한 쿼리를 튜닝하게 도와줄 수도 있다. 그리고 iBATIS는 DBA들이 그렇게 할 수 있도록 해주는 프레임워크이다. iBATIS를 사용하는 어떤 팀은 심지어 DBA 혹은 데이터 모델러가 iBATIS SQL 파일을 직접 관리하도록 하기도 한다. 데이터베이스 관리자들과 SQL 프로그래머들은 iBATIS를 이해하는 데 어려움이 없을 것이다. iBATIS는 보이지 않는 곳에서 특별한 처리를 하지 않고, 있는 그대로의 SQL을 볼 수 있게 해주기 때문이다.

1.4.2 여러 이종 시스템들에 의한 접근

중요한 데이터베이스라면 의심할 여지없이 하나 이상의 시스템이 의존하게 마련이다. 비록 단순히 하나의 데이터베이스를 공유하는 두 개의 웹애플리케이션이라 하더라도 고려해야 할 사항이 여러 개 있다. 웹 쇼핑 장바구니라는 웹애플리케이션이 카테고리(Category) 코드를 가지고 있는 데이터베이스를 사용한다고 해보자. 웹 쇼핑 장바구니만 고려한다면 카테고리 코드는 정적이고 결코 변하지 않기 때문에 성능 향상을 위해 그 코드들을 캐싱할 것이다. 자, 이제 두 번째 웹애플리케이션인 웹 관리 시스템이 카테고리 코드를 수정하게 만들어져 있다고 하자. 웹 관리 시스템 애플리케이션은 다른 서버에서 작동하는 전혀 다른 프로그램이다. 웹 관리 시스템이 카테고리 코드를 수정할 때, 웹 쇼핑 장바구니 프로그램은 카테고리 코드를 캐시에 재적재해야 할지 어떻게 알 수 있을까? 이것이 때로는 매우 복잡한 문제가 되기도 하는 간단한 예이다.

서로 다른 시스템들이 서로 다른 방식으로 하나의 데이터베이스에 접근하고 사용한다. 한 애플리케이션은 웹기반 전자 상거래 시스템으로 매우 많은 데이터베이스 수정과 데이터 생성 작업을 수행한다. 다른 시스템은 데이터베이스의 테이블에 독점적인 접근을 필요로 하는 제 3의 시스템에서 데이터를 특정 시간마다 일괄 적재하는 작업을 할 것이다. 게다가 리포팅 엔진 역할을 수행하는 다른 시스템도 있어서, 끊임없이 복잡한 쿼리들을 날려대며 데이터베이스 테이블에 부하를 주기도 한다. 이처럼 복잡한 시스템이 있을 수 있음은 쉽게 생각해 볼 수 있다.

중요한 것은 데이터베이스가 한 개 이상의 시스템에 의해 사용되면 일이 어려워진다는 것을 알아야 한다는 것이다. iBATIS가 여러 방식으로 여기에 도움을 줄 수 있다. 우선 iBATIS는 트랜잭션 시스템, 일괄 처리 시스템 그리고 리포팅 시스템 등을 포함한 모든 종류의 시스템에서 잘 작동하는 퍼시스턴스 프레임워크이다. 이는 어떠한 시스템이 데이터베이스에 접근하든지 상관 없이, iBATIS는 훌륭한 도구라는 뜻이다. 두 번째로, 당신이 iBATIS를 사용할 줄 안다면, 혹은 자바 같은 견고한 시스템만 사용할 줄 알아도, 여러 다른 시스템들과 소통하는 분산 캐시를 사용할 수 있을 것이다. 마지막으로, 너무 복잡한 경우에는 간단하게 iBATIS의 캐싱 기능을 꺼버리고, 데이터베이스를 함께 사용하는 다른 데이터베이스의 쿼리가 캐시 동시 사용을 고려하지 않았더라도 상관없이 완벽하게 작동하는 명시적인 쿼리와 수정 구문을 작성하면 된다.

1.4.3 복잡한 키와 관계들

관계형 데이터베이스는 매우 엄격한 설계 규칙들을 따르도록 설계돼 있고, 또한 그런 의도로 만들어진 것이다. 때때로 이런 규칙이 어떤 때는 좋은 이유로, 또 어떤 때는 좋지 않은 이유로 깨지기도 한다. 복잡한 키와 관계들은 보통 규칙이 깨지거나 규칙을 잘못 이해하거나 혹은 규칙을 과도하게 적용할 때 발생하게 된다. 관계 설계의 규칙들 중 한 가지는 각각의 데이터 레코드는 기본키(primary key)로 유일하게 구분되어야 한다는 것이다. 하지만 어떤 데이터베이스 설계는 실세계의 데이터를 키로 사용하는 자연키(natural key)를 사용하기도 한다. 게다가 더욱 복잡하게 설계하면 둘 혹은 그 이상의 칼럼을 조합하여 키를 만들 수도 있다. 또한 기본키를 종종 서로 다른 테이블들과의 관계를 생성하기 위해 사용하기도 한다. 그래서 복잡하거나 잘못된 기본키 정의는 다른 테이블들 간의 관계에까지 문제를 파생시키게 된다.

때로는 기본키 규칙을 따르지 않을 때도 있다. 그것은 때때로 데이터가 기본키를 전혀

갖고 있지 않기 때문이다. 이렇게 되면 유일하게 데이터를 구분하는 것이 어려워지는 만큼 데이터베이스 쿼리도 복잡해지게 된다. 이로 인해 테이블 간의 관계를 설정하는 것이 아주 어렵고 너저분해지게 된다. 또한 보통 기본키가 성능을 향상시켜주는 인덱스를 제공하고 데이터의 물리적인 순서를 결정하게 해주는데, 이제는 그렇지 못하게 됨으로써 성능상의 악영향을 주게 된다.

다른 경우에는 기본키 규칙을 과도하게 준수할 때도 있다. 실용적이지 못한 이유로 자연키를 복합적으로 사용하는 데이터베이스도 있다. 이 설계는 규칙을 지나치게 심각하게 받아들이고 가능한 한 엄격하게 구현해서 생긴 결과이다. 자연키를 사용하여 테이블들 간의 관계를 설정하는 것은 사실상 실세계 데이터의 중복 사용을 발생시키며, 이는 데이터베이스 유지 보수에 언제나 나쁜 영향을 끼친다. 복합키는 또한 관계 맺기에 사용될 경우 더 많은 중복을 발생시켜, 이제는 관련된 테이블의 한 레코드를 단일하게 구분하기 위해서 여러 칼럼 정보를 유지해야만 하게 된다. 이렇게 되면 데이터베이스가 유연성을 잃게 된다. 자연키와 복합키 모두 관리하기 어렵고, 데이터 이전 작업을 악몽으로 만들 수 있기 때문이다.

iBATIS는 어떠한 복잡한 키 정의나 관계도 다룰 수 있다. 비록 데이터베이스 설계를 제대로 하는 것이 가장 좋겠지만, iBATIS는 의미 없는 키[10], 자연키, 복합키 혹은 아예 키가 없는 경우의 테이블까지도 다룰 수 있다.

1.4.4 비정규화된 혹은 과도하게 정규화된 모델

관계형 데이터베이스의 설계는 중복을 제거해가는 과정이다. 중복의 제거는 데이터베이스의 유연성, 관리 용이성 그리고 성능 향상을 보장하는 데 매우 중요한 역할을 한다. 데이터 모델에서 중복을 제거하는 과정을 정규화(normalization)라고 부르며 정규화에는 이를 완수하는 특정한 단계들이 존재한다. 표 형태의 원시 데이터는 매우 많은 중복을 포함하고 있고, 이를 비정규화된(denormalized) 상태로 간주한다. 정규화는 여기서 자세히 다루기에는 어려운 주제이다.

데이터베이스가 처음 설계될 때, 원시 데이터의 중복을 분석한다. 데이터베이스 관리자, 데이터 모델러 혹은 개발자는 원시 데이터를 수집하고 중복 제거를 위한 매우 특별한

10. **역자주 |** sequence 등을 통해 순차 적으로 자동 생성되는 키

규칙에 따라서 이를 정규화한다. 비정규화된 관계 모델은 각각 많은 레코드와 칼럼을 가진 여러 테이블에 중복된 데이터를 포함하고 있다. 정규화된 모델은 최소한의 중복이나 아예 중복이 없고 테이블 개수가 더 많아질지라도, 각 테이블은 더 적은 레코드와 칼럼을 가지게 된다.

정규화에 완벽한 수준이란 없다. 비정규화 상태도 단순성이나 때로는 성능상의 장점을 지닐 수 있다. 비정규화된 모델이 데이터를 정규화했을 때보다 데이터를 더 빨리 저장하고 가져올 수도 있다. 이런 장점은 단지 더 적은 쿼리 구문만을 필요로 하고 더 적은 조인(join)만 계산하면 되며, 일반적으로 부하가 적다는 사실로도 알 수 있다. 비정규화는 항상 예외적인 것(어쩌다 필요한 것)일 뿐이며, 비정규화해야 하는 규칙 같은 것은 없다.

데이터베이스 설계에 좋은 접근 방법은 '교과서적'인 정규화 모델로부터 시작하는 것이다. 그리고서 필요에 따라 해당 모델을 비정규화한다. 데이터베이스를 나중에 비정규화하는 것이 비정규화된 모델을 다시 정규화하는 것보다 훨씬 쉽다. 그러니 새로운 데이터베이스 설계를 시작할 때는 항상 정규화된 모델에서 시작하도록 한다.

데이터베이스를 과도하게 정규화할 수도 있는데 그렇게 하면 문제가 생긴다. 너무 많은 테이블이 생겨서 관리해야 할 관계가 많아지게 된다. 이렇게 되면 데이터를 조회할 때 매우 많은 테이블 조인을 해야 되고 이는 밀접하게 관계된 데이터들을 수정할 때 많은 수의 업데이트 구문이 필요함을 의미한다. 이러한 특성들 모두 성능상에 부정적인 영향을 주게된다. 또한 이는 객체 모델에 데이터를 매핑하는 것을 어렵게 만든다. 개발자들은 잘게 나뉜 데이터 만큼 잘게 나뉜 많은 클래스들을 원하지는 않는다.

비정규화된 모델도 과도하게 정규화된 모델보다 심하면 심했지 문제가 많기는 매한가지이다. 비정규화된 모델은 보통 많은 레코드와 칼럼을 갖게 된다. 지나치게 많은 레코드는 단순히 생각해도 검색해야 할 데이터가 많아지기 때문에 성능에 부정적인 영향을 끼친다. 지나치게 많은 칼럼 또한 각 레코드의 크기가 커지고 그로 인해서 수정이나 쿼리를 수행하는 각 작업 시간에 더 많은 리소스를 차지하기 때문에 마찬가지로 좋지 않다. 이렇게 칼럼이 많은 테이블을 다룰 때는 특정 작업시 필요한 칼럼들만 수정이나 조회구문에서 사용하도록 신경 써야 한다. 더욱이 비정규화된 모델은 효율적인 인덱싱을 불가능하게 만든다.

iBATIS는 비정규화된 모델과 과도하게 정규화된 모델을 둘 다 잘 다룬다. iBATIS는 데이터베이스와 객체 모델의 크기(잘게 나뉘었는지, 큰 덩어리인지)에 대해 어떠한 가정도 하지 않으며, 또한 두 모델이 완전히 동일한지 혹은 서로 그다지 비슷하지 않은지도 가정하지 않는다. iBATIS는 객체 모델과 관계 데이터 모델의 분리를 매우 잘 정의하고 있다.

1.4.5 빈약한 데이터 모델(Skinny Data Model)

빈약한 데이터 모델은 가장 악질적이고 문제 많은 관계형 데이터베이스의 오용 사례 중 하나이다. 불행히도, 이것도 가끔 꼭 필요할 때가 있다. 빈약한 데이터 모델이란 기본적으로 각 테이블을 일반적인 데이터 구조체로 만들어서 이름과 값의 쌍으로 된 데이터 집합을 저장하는데, 이는 자바의 프로퍼티 파일(properties file)이나 Windows의 옛날 INI 파일과 매우 흡사하다. 때때로 이 테이블에는 데이터 타입 정의 같은 메타 데이터를 저장하기도 한다. 이것은 데이터베이스가 한 칼럼에 대해 한 개의 데이터 타입만을 지정하게 되어 있기 때문에 필요하다. 빈약한 데이터 모델을 더 잘 이해하기 위해 표 1.2의 일반적인 주소 데이터 예제를 살펴보자.

표 1.2 일반적인 형태의 주소 데이터

ADDRESS_ID	STREET	CITY	STATE	ZIP	COUNTRY
1	123 Some Street	San Francisco	California	12345	USA
2	456 Another Street	New York	New York	54321	USA

분명히 이 주소 데이터는 더 정규화할 만한 것이 있다. 예를 들어 COUNTRY(국가), STATE(주) 그리고 CITY(시)와 ZIP(우편번호)을 관계형 테이블로 분리할 수도 있다. 하지만 지금 방식이 더 간단하고 효과적으로 설계되어, 대부분의 애플리케이션에서 잘 작동한다. 요구사항이 복잡하지만 않다면 이건 별로 문제될만한 설계는 아니다.

만약 우리가 이 데이터를 가져다 빈약한 데이터 모델로 재구성한다면 표 1.3와 같이 될 것이다.

표 1.3 빈약한 데이터 모델 형태의 주소 데이터

ADDRESS_ID	FIELD	VALUE
1	STREET	123 Some Street
1	CITY	San Francisco
1	STATE	California
1	ZIP	12345
1	COUNTRY	USA
2	STREET	456 Another Street
2	CITY	New York
2	STATE	New York
2	ZIP	54321
2	COUNTRY	USA

이 설계는 진정 악몽 그 자체이다. 시작하기 전에, 이 데이터는 지금 상태보다 더 정규화할 수 있는 가능성이 전혀 없이, 이 자체를 가장 정규화된 형태로 볼 수 있다. 여기에는 COUNTRY, CITY, STATE 혹은 ZIP 테이블 간의 관계를 맺어 줄 수 있는 것이 없다. 칼럼이 한 개인 테이블에 여러 개의 외래키를 설정할 수는 없기 때문이다. 이 데이터는 또한 쿼리를 하기가 매우 어렵고, 여러가지 주소 필드(예: 거리명과 도시명을 모두 검색조건으로 할 경우)를 필요로 하는 Query-by-Example 스타일의 쿼리를 수행할 때는 매우 복잡한 서브 쿼리를 필요로 하게 된다. 데이터를 수정하는 상황이 되면 이러한 설계는 성능면에서 특히 취약함을 보여준다. 한 개의 주소를 입력하기 위해서 한 번이 아닌 다섯 번의 insert 구문을 한 테이블에 날려야 하기 때문이다. 이것은 잠재적으로 더 많은 락 경합(lcok contention)을 일으키거나 심지어는 데드락(dead lock) 상태에 빠질 수도 있다. 더욱이 빈약한 데이터 모델의 레코드 수는 정규화된 모델의 5배가 된다. 이 늘어난 레코드의 수와 데이터 정의의 부족(칼럼 데이터 타입이 정확하게 정의되지 않음) 그리고 이 데이터를 수정하기 위한 update 구문 수의 증가 등으로 인해서 효율적인 인덱싱이 불가능해진다.

더 볼 것도 없이, 이러한 설계가 왜 문제가 많은지 그리고 왜 무슨 일이 있어도 이를 피해야만 하는지 쉽게 알 수 있다. 이 방식이 쓸모있는 경우는 애플리케이션에 동적인 필드가 있을 때이다. 어떤 애플리케이션의 경우에는 사용자가 레코드에 추가적인 데이터를 넣어야 할 때도 있다. 만약 사용자가 애플리케이션이 실행되고 있는 도중에 동적으로 추가적인 필드에 데이터를 삽입하고자 할 때 이 모델이 적합하다. 하지만 여전히 정해진 데이터들은 적절하게 정규화를 해야 하며, 그 이후에 추가적인 동적 필드를 부모 레코드와 연관시키면 된다. 이러한 설계는 여전히 위에서 논한 문제들을 가지고 있긴 하지만, 대부분의 데이터(아마도 중요한 데이터들)는 적합하게 정규화가 되어 있기 때문에 그 고통은 최소화된다.

비록 기업용 데이터베이스에서 빈약한 데이터 모델을 만나게 되더라도, iBATIS는 문제 없이 이 상황을 처리할 수 있다. 어떤 필드가 존재하는지 알 수 없기 때문에, 클래스를 빈약한 데이터 모델에 매핑하는 것은 매우 어렵고 어떨 때는 불가능하기도 하다. 운이 좋다면 이것들을 Hashtable에 매핑할 수 있을텐데, 다행히 iBATIS는 그런 기능을 지원한다. iBATIS를 이용하면, 모든 테이블을 사용자 정의 클래스로 매핑할 필요가 없다. iBATIS는 관계형 데이터를 원시타입(primitives), 맵(Map), XML 그리고 사용자 정의 클래스(예를 들어 자바빈즈)로 매핑하는 기능을 지원한다. 이러한 커다란 유연성으로 인해, iBATIS는 빈약한 데이터 모델을 포함한 여러 복잡한 데이터 모델에 대해서 매우 효율적으로 작동한다.

1.5 요약

iBATIS는 모든 문제를 해결하려 하기 보다는, 가장 중요한 문제들을 집중해서 해결하도록 설계된 복합적인 솔루션이다. iBATIS는 여러 접근 방법들로부터 아이디어를 차용했다. 저장 프로시저처럼 모든 iBATIS의 문장들은 시그너처(signature, 메소드의 입력 파라미터와 반환값 정의)를 가지고 있어서, 각 문장들에 이름을 부여하고 그것의 입력과 출력 파라미터들을 정의한다(캡슐화). 인라인 SQL과 유사하게, iBATIS는 SQL을 쓰고자 하는 그 형태 그대로 쓸 수 있으며, 각 프로그래밍 언어의 변수를 SQL의 파라미터와 결과로 직접 사용할 수 있다. 동적 SQL 처럼, iBATIS는 실행시에 동적으로 SQL을 변경할 수 있는 방법을 제공하여, 그런 쿼리들은 사용자의 요청에 따라 동적으로 생성될 수 있다. iBATIS는 객체 관계 매핑 도구들로부터 캐싱이나 적재 지연 그리고 고급 트랜잭션 관리 기능 등을 포함한 많은 개념들을 차용해 왔다.

애플리케이션 아키텍처에서 iBATIS는 퍼시스턴스 계층에 속한다. iBATIS는 애플리케이션의 모든 계층에서 필요로 하는 것들을 쉽게 구현할 수 있는 기능을 제공함으로써 다른 계층들을 지원하기도 한다. 예를 들어 웹 검색 엔진은 검색 결과를 페이징한 리스트를 필요로 한다. iBATIS는 쿼리에 시작 지점과 반환할 레코드의 수를 지정할 수 있게 함으로써 이런 기능을 지원한다. 이를 통해 데이터베이스의 세세한 사항은 애플리케이션으로부터 숨기면서 하위 레벨에서 페이징 작업을 할 수 있게 된다.

iBAITS는 어떤 규모나 목적을 가진 데이터베이스와도 잘 작동한다. iBATIS는 배우기에 간단하고 빠르게 적용 가능하기 때문에 작은 규모의 애플리케이션 데이터베이스와 잘 어울린다. iBATIS는 데이터베이스의 설계와 행위 그리고 의존관계 등 애플리케이션이 데이터베이스를 어떻게 사용하는지에 대해 영향을 끼칠 만한 사항들에 대해서 어떠한 가정도 하지 않기 때문에 대규모 기업용 데이터베이스와도 멋지게 작동한다. 데이터베이스의 설계가 엉망이고, 정치적 결정에 의한 혼란으로 도배가 돼 있더라도 iBATIS는 그러한 데이터베이스조차도 쉽게 사용할 수 있게 해준다. 위 모든 사항 외에도, iBATIS는 거의 모든 상황에 대처할 수 있도록 매우 유연하게 설계돼 있으며, 동시에 반복적이고 불필요한 코드들을 제거함으로써 개발자의 시간을 아껴주기도 한다.

1장에서 우리는 iBATIS의 탄생 철학과 기원을 살펴보았다. 2장에서는 iBATIS가 정확히 무엇이고 어떻게 작동하는지에 대해 설명할 것이다.