**SQL 개발자를 위한 Oracle Berkeley DB 가이드**

*저자 – Margo Seltzer*

Oracle Berkeley DB의 SQL 기능을 활용하는 방법을 배워 보십시오.

게시일: 2007년 9월

Oracle Berkeley DB 팀원들은 "Berkeley DB에서 (이런 저런) SQL 쿼리를 실행하려면 어떻게 해야 하나요?"라는 질문을 매우 자주 접하곤 합니다. 본 문서를 통해 Oracle Berkeley DB에서 자주 사용되는 SQL 기능을 소개 드리고자 합니다. Oracle Berkeley DB는 신속하고, 안정적인 로컬 퍼시스턴스 환경을 제공하는 오픈 소스 임베디드 데이터베이스 엔진으로 별도의 관리를 전혀 필요로 하지 않습니다. Oracle Berkeley DB는 상대적으로 적은 수의 쿼리를 고정적으로 실행하는 환경에서 성능을 극대화하고자 하는 경우 최선의 선택이 될 수 있습니다.

그럼 가장 기본적인 내용부터 시작해 봅시다. 먼저 Berkeley DB에서 사용되는 용어를 알아 둘 필요가 있습니다.

SQL 프로그래머들을 위한 "번역 가이드"가 아래와 같습니다.

|  |  |
| --- | --- |
| **SQL 용어** | **Oracle Berkeley DB 용어** |
| 데이터베이스 | 환경 (Environment) |
| 테이블 | 데이터베이스 (Database) |
| 튜플/로우 | 키/데이터 페어 (Key/Data pair) |
| 1차 인덱스 | 키 (Key) |
| 2차 인덱스 | 2차 데이터베이스 (Secondary Database) |

이제 고전적이면서도 단순화된 형태의 직원 데이터베이스를 애플리케이션 도메인으로 선정해 봅시다. 또 Berkeley DB의 가장 기본적인 기능, 즉 동시성, 트랜잭션, 복구성 등에 관련된 기능을 모두 활용하는 것으로 가정하겠습니다.

**데이터베이스의 생성**

SQL에서는 아래와 같은 방법으로 데이터베이스를 생성합니다.

CREATE DATABASE personnel

Berkeley DB에서는 애플리케이션 데이터가 위치하는 환경(environment)을 생성하게 됩니다. 앞으로 코드 전반에 걸쳐 DB\_ENV 타입의 환경 핸들(environment handle)을 통해 환경을 참조하게 됩니다. 또 환경에 대한 작업을 수행할 때에도 핸들을 활용하게 됩니다. 예제에서는 에러 핸들링 코드를 생략하고 API에만 집중하여 코드를 구현하기로 합니다.

 DB\_ENV \*dbenv;

 int ret;

 /\* Create the handle. \*/

 DB\_ASSERT(db\_env\_create(&dbenv, 0) == 0);

 /\*

 \* If you wanted to configure the environment, you would do that here.

 \* Configuraition might include things like setting a cache size,

 \* specifying error handling functions, specifying (different)

 \* directories in which to place your log and/or data files, setting

 \* parameters to describe how many locks you'd need, etc.

 \*/

 /\* Now, open the handle. \*/

 DB\_ASSERT(dbenv->open(dbenv, "my\_databases/personnel",

 DB\_CREATE | DB\_INIT\_LOCK | DB\_INIT\_MPOOL | DB\_INIT\_TXN | DB\_THREAD, 0644);

이것으로 새로운 환경이 생성, 오픈되었습니다. 여기서 몇 가지 참고할 사항이 다음과 같습니다:

* 이 작업을 수행하기 전에 my\_databases/personnel 디렉토리가 미리 생성되어 있어야 합니다.
* 오픈 콜에 대한 최종 매개변수는 환경의 일부로서 생성하는 환경의 일부로서 생성되는 파일의 모드입니다.
* 이곳에 명시된 플래그를 이용하여 환경을 생성
(DB\_CREATE)하고, 락을 적용(DB\_INIT\_LOCK)하고, 공유 메모리 버퍼 풀을 설정(DB\_INIT\_MPOOL)하고, 트랜잭션을 사용(DB\_INIT\_TXN)하고 서로 다른 컨트롤을 갖는 쓰레드를 환경 내에서 동시 사용(DB\_THREAD)할 수 있습니다.

SQL 구문은 일반적으로 별도의 서버에 의해 처리되며, 이 서버는 DBA에 의해 별도의 튜닝 과정을 거칩니다. Berkeley DB은 애플리케이션에 임베드된 형태로 구현되므로 애플리케이션에서 이러한 튜닝 작업의 대부분을 수행할 수 있을 것입니다. 하지만 이러한 내용들은 데이터베이스 튜닝에 관련된 것이므로 별도의 문서를 통해 설명하는 것이 적절할 것입니다.

데이터베이스를 생성한 다음으로는 몇 가지 테이블을 생성해 주어야 합니다. Berkeley DB에서 테이블은 DB \* 타입의 핸들에 의해 참조됩니다. 일반적으로 애플리케이션의 각 테이블에 대해 핸들을 하나씩 오픈하고, 하나 또는 그 이상의 쓰레드에서 핸들을 이용하는 방법이 주로 사용됩니다.

SQL에서 테이블을 생성하는 방법이 아래와 같습니다:

CREATE TABLE employee

 (primary key empid int(8), last\_name varchar(20), first\_name varchar(15),

 salary numeric(10, 2) salary, street varchar (20), city varchar (15),

 state char(2), zip int(5))

위 SQL 구문을 Berkeley DB에서 구현하는 방법을 설명하기 전에 한 가지 꼭 알아 두어야 할 사항이 있습니다. SQL에서는 데이터베이스가 데이터 스키마의 구현 및 해석을 담당합니다. Berkeley DB에서는 이러한 해석 작업이 애플리케이션에 의해 수행됩니다. DML 구문을 통해 보다 분명하게 확인하게 되겠지만, Berkeley DB는 employee 테이블의 프라이머리 키에 대한 정보만을 가지고 있으며 데이터베이스의 각 필드에 대한 정보는 가지고 있지 않습니다.

먼저, 생성되는 테이블을 참조하기 위한 데이터베이스 핸들을 생성합니다. (여기에서도 에러 핸들링 코드는 생략되어 있습니다.)

 DB \*dbp;

 DB\_ENV \*dbenv;

 /\* Let's assume we've used the code from above to set dbenv. \*/

 ASSERT(db\_create(&dbp, dbenv, 0) == 0);

 /\*

 \* Like with the environment, tables can also be configured. You

 \* can specify things like comparison functions, page-size, etc.

 \* That would all go here.

 \*/

 /\* Now, we'll actually open/create the primary table. \*/

 ASSERT(dbp->open(dbp, NULL, "employee.db", NULL, DB\_BTREE,

 DB\_AUTO\_COMMIT | DB\_CREATE | DB\_THREAD, 0644) == 0).

위 코드를 실행하면 B-트리를 1차 인덱스 구조로 사용하는 테이블이 생성됩니다. 이 테이블은 employee.db라는 이름으로 my\_databases/personnel 디렉토리에 저장됩니다. 이 파일은 하나의 테이블만을 포함하며 최종 매개변수(0644)에 의해 명시된 파일 시스템 권한을 갖습니다. 테이블 생성 과정에서 설정된 플래그는 트랜잭션 기반 작업을 허용
(DB\_AUTO\_COMMIT)하고, 테이블이 존재하지 않는 경우 테이블을 생성하도록 허용(DB\_CREATE)하고 있으며, 생성된 핸들이 여러 쓰레드에 의해 동시에 사용될 수 있도록 허용(DB\_THREAD)하고 있습니다.

위의 코드에서 프라이머리 키(인덱스)를 구성하는 요소가 무엇인지, 또 테이블에 저장되는 데이터 필드가 무엇인지 전혀 정의되지 않고 있음을 참고하시기 바랍니다. 이러한 모든 작업은 애플리케이션에서 담당합니다. 자세한 사항은 INSERT, SELECT, UPDATE 작업을 설명하면서 좀 더 분명하게 확인하실 수 있을 것입니다.

이제, employee id를 기준으로 1차 인덱스를, 그리고 last name을 기준으로 2차 인덱스를 생성하는 작업을 수행해 봅시다.

이를 위한 SQL 구문이 아래와 같습니다:

CREATE INDEX lname ON employee (last\_name)

Berkeley DB에서는 2차 인덱스가 테이블과 동일한 형태를 가집니다. 따라서 2차 인덱스를 생성한 뒤 테이블과 연계하는 작업이 필요합니다. 이 작업을 위해, 애플리케이션이 사용하는 데이터 구조에 대해 좀 더 이해할 필요가 있습니다.

애플리케이션에서 C 언어의 데이터 구조를 사용하여 employee 테이블의 튜플(tuple)을 저장한다고 가정해 봅시다. 데이터 구조는 아래와 같이 정의될 수 있습니다:

typedef struct \_emp\_data {

 char lname[20];

 char fname[15];

 float salary;

 char street[20];

 char city[15];

 char state[2];

 int zip;

} emp\_data;

이제 employee ID가 정수 타입을 갖는다고 가정해 보겠습니다:

typedef int emp\_key;

Berkeley DB에서 키 또는 데이터 아이템을 처리하기 위해서는 DBT라는 이름의 구조를 사용해야 합니다. BBT는 바이트 문자열을 인캡슐레이션하고 포인터(pointer)와 길이(length)로 표현합니다. 포인터는 DBT의 데이터 필드에 의해 참조되며, 길이는 DBT의 size 필드에 저장됩니다. 직원 레코드를 가리키는 키/데이터 페어(key/data pair)를 처리하기 위해서는 emp\_key를 위한 DBT와 emp\_data를 위한 DBT를 함께 활용해야 합니다.

 DBT key\_dbt, data\_dbt;

 emp\_key ekey;

 emp\_data edata;

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\*

 \* Now make the key and data DBT's reference the key and data

 \* variables.

 \*/

 key\_dbt.data = &ekey;

 key\_dbt.size = sizeof(ekey);

 data\_dbt.data = &edata;

 data\_dbt.size = sizeof(edata);

위에서 SQL의 튜플은 키/데이터 페어로 표현되고 있지만, 이 키/데이터 페어를 어떻게 해석할 것인가는 애플리케이션에 의해 달라집니다.

이제 2차 인덱스의 문제로 다시 돌아가 보겠습니다. Berkeley DB는 키/데이터 페이어의 데이터 엘리먼트의 구조 또는 스키마를 이해하지 못하므로, 2차 인덱스로 사용할 필드가 무엇인지 확인하기 위해서는 애플리케이션의 도움이 필요합니다. 이를 위해 애플리케이션은 콜백 함수(callback function)를 사용합니다. 콜백 함수는 키/데이터 페어를 취하고 2차 키로 사용되는 값을 참조하는 DBT를 반환합니다.

따라서 last\_name을 기준으로 2차 인덱스를 생성하려면, 키/데이터 페어를 입력으로 받아들인 후 해당 데이터 아이템의 last\_name 필드를 참조하는 DBT를 반환하는 콜백 함수를 작성해야 합니다.

int

lname\_callback(DB \*dbp, const DBT \*key, const DBT \*data, DBT \*skey)

{

 emp\_data \*edata;

 /\*

 \* We know that the opaque byte-string represented by the data DBT

 \* represents one of our emp\_data structures, so let's cast it

 \* to one of those so that we can manipulate it.

 \*/

 edata = data->data;

 skey->data = edata->lname;

 skey->size = strlen((edata->lname);

 return (0);

}

콜백 함수를 작성했다면, 이제 2차 인덱스를 설정할 수 있습니다. 앞에서 설명한 것처럼 2차 인덱스는 테이블의 형태를 가집니다. 아래와 같이 테이블을 하나 생성합니다:

 DB \*sdbp;

 ASSERT(db\_create(&sdbp, dbenv, 0) == 0);

 /\* Configure sdbp. \*/

 ASSERT(sdbp->open(sdbp, NULL, "emp\_lname.db", NULL, DB\_BTREE,

 DB\_AUTO\_COMMIT | DB\_CREATE | DB\_THREAD, 0644) == 0);

last\_name의 인덱싱을 위해 B-트리 구조가 사용되므로, 앞에서 사용한 것과 동일한 플래그와 모드를 그대로 적용합니다.

마지막으로 2차 인덱스 테이블과 메인 테이블(employee 테이블)을 연계합니다. 앞에서 설명한 것처럼 employee 테이블을 처리하기 위한 핸들로는 dbp가, 2차 인덱스 테이블을 처리하기 위한 핸들로는 sdbp가 사용됩니다.

 ASSERT(dbp->associate(dbp, NULL, sdbp, lname\_callback, flags) == 0);

몇 가지 참고할 사항이 다음과 같습니다:

* 생성 가능한 2차 인덱스의 수에는 제한이 없습니다. 여기서 2차 인덱스가 INSERT 작업의 성능을 저하시키는(각 2차 인덱스별로 인덱스 엔트리가 생성되어야 하므로) 반면, 튜플에 대한 SELECT 쿼리의 성능은 극적으로 개선될 수 있다는 측면을 감안해야 합니다.
* • 2차 인덱스가 오픈, 연계된 상태에서 메인 테이블을 업데이트하면, 2차 인덱스는 동시에 업데이트 처리됩니다. 하지만 2차 인덱스를 오픈, 연계하지 않은 상태에서 베이스 테이블을 수정하면, 2차 인덱스의 최신 상태를 유지할 수 없게 됩니다.
따라서 이러한 문제가 발생하지 않도록 주의해야 합니다.

함께 고려해야 할 DDL 커맨드로 drop index, drop table, drop database가 있습니다.

SQL에서와 마찬가지로, Berkeley DB에서도 인덱스를 drop 처리하고 테이블을 delete 처리할 수 있습니다. SQL에서는 아래와 같은 명령이 사용됩니다.

DROP TABLE employee

or

DROP INDEX lname

SQL에서 테이블을 drop 처리하면 연계된 인덱스들도 함께 삭제됩니다. 하지만 Berkeley DB에서는 인덱스의 삭제를 명시적으로 수행해 주어야 합니다. Berkeley DB에서 테이블 또는 인덱스를 drop 처리하는 방법은 동일합니다.

테이블을 제거하기 전에 테이블에 대한 모든 데이터베이스 핸들을 close 처리해야 합니다. 테이블을 close 처리하는 방법은 간단합니다. employee 데이터베이스에서 2차 인덱스를 drop 처리하는 경우를 가정해 봅시다. 먼저 2차 인덱스를 close 처리합니다:

 sdbp->close(sdbp, 0)

데이터베이스 핸들에 대해 close 메소드를 실행하면, 이 핸들은 다시 사용될 수 없습니다.

2차 인덱스 테이블을 close 처리한 뒤에는 dbenv 핸들에 대해 dbremove 메소드를 사용하여 삭제할 수 있습니다:

 DB\_ENV \*dbenv;

 ASSERT(dbenv->dbremove(dbenv,

 NULL, "emp\_lname.db", NULL, DB\_AUTO\_COMMIT) == 0);

동일한 호출 과정(close 및 dbremoving)을 이용하여 테이블을 drop 처리하는 것도 가능합니다.

이번에는 테이블을 drop하는 것이 아니라 테이블 이름만을 변경해야 하는 경우를 가정해 봅시다. 이 작업도 비슷한 방법으로 진행됩니다.

remove 작업과 마찬가지로, 먼저 테이블 핸들을 close 처리해야 합니다:

 dbp->close(dbp, 0);

이제 테이블의 이름을 변경합니다:

 DB\_ENV \*dbenv;

 ASSERT(dbenv->dbrename(dbenv,

 NULL, "employee.db", NULL, "newemp.db", DB\_AUTO\_COMMIT) == 0);

마지막으로 데이터베이스를 완전하게 삭제해야 하는 경우입니다. SQL에서는 아래와 같은 명령이 사용됩니다.

DROP DATABASE personnel

Berkeley DB에서도 같은 작업을 수행할 수 있습니다.

먼저 환경을 close 처리합니다:

 ASSERT(dbenv->close(dbenv, 0) == 0);

테이블 핸들을 close하는 경우와 마찬가지로, 환경 핸들(environment handle)을 close 처리하면 이 핸들을 더 이상 사용할 수 없게 됩니다. 따라서 데이터베이스를 drop 처리하려면, 새로운 핸들을 생성하고 이 핸들을 이용하여 데이터베이스(환경)를 제거해야 합니다.

 ASSERT(db\_env\_create(&dbenv, 0) == 0);

 ASSERT(dbenv->remove(dbenv, "my\_databases/personnel", 0) == 0);

지금까지 SQL의 DDL 구문을 Berkley DB에서 사용되는 명령으로 변환하는 방법에 대해 설명했습니다. 다음으로는 SQL DML 구문을 Berkeley DB에 적용하는 방법에 대해 설명해 보겠습니다.

**Berkeley DB에서 SQL DML 작업 수행하기**

지금부터는 Berkeley DB에서 INSERT, UPDATE, DELETE 작업을 실행하는 방법에 대해 논의하겠습니다.

SQL에서 INSERT 구문을 이용하여 테이블에 데이터를 추가하는 방법이 아래와 같습니다:

INSERT INTO employees VALUES (00010002, "mouse", "mickey", 1000000.00,

 "Main Street", "Disney Land", "CA", 98765);

Berkeley DB에서는 INSERT 작업을 위해 put 메소드를 사용합니다.

앞에서 예로 든 employee 테이블을 참조하는 dbp 데이터베이스 핸들이 열려 있다고 가정해 봅시다. 이제, Mickey Mouse가 새로운 직원으로 채용되었습니다.

 DB \*dbp;

 DBT key\_dbt, data\_dbt;

 emp\_data edata;

 emp\_key ekey;

 /\* Put the value into the employee key. \*/

 ekey = 00010002;

 /\* Initialize an emp\_data structure. \*/

 strcpy(edata.lname, "Mouse");

 strcpy(edata.fname, "Mickey");

 edata.salary = 1000000.00;

 strcpy(edata.street, "Main Street");

 strcpy(edata.city, "Disney Land");

 strcpy(edata.state, "CA");

 edata.zip = 98765;

 /\* Initialize DBTs \*/

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, assign key and data values to DBTs. \*/

 key->data = &ekey;

 key->size = sizeof(ekey);

 data->data = &edata;

 data->size = sizeof(edata);

 /\* Finally, put the data into the database. \*/

 ASSERT(dbp->put(dbp, NULL, &key\_dbt, &data\_dbt, DB\_AUTO\_COMMIT) == 0);

employee 테이블과 2차 인덱스가 연계된 상태라면, INSERT 작업을 수행하면서 2차 인덱스도 자동으로 업데이트될 것입니다.

이제 테이블에 저장된 데이터의 일부를 변경하는 경우를 생각해 봅시다. Mickey의 연봉을 인상해 주려 합니다. 몇 가지 방법이 가능합니다.

첫 번째 방법은 위에서 설명한 INSERT 코드와 동일한 형태로 구현됩니다. 테이블에 put 메소드를 실행했을 때 키가 이미 존재한다면(또 테이블이 하나의 키에 대해 중복 데이터 값을 허용하지 않는다면), put 메소드는 기존 버전의 데이터를 새로운 데이터로 대체합니다. 아래의 코드를 실행하면 Mickey의 salary가 $1,000,000에서 $2,000,000으로 변경됩니다.

 /\* Put the value into the employee key. \*/

 ekey = 00010002;

 /\* Initialize an emp\_data structure. \*/

 strcpy(edata.lname, "Mouse");

 strcpy(edata.fname, "Mickey");

 edata.salary = 2000000.00;

 strcpy(edata.street, "Main Street");

 strcpy(edata.city, "Disney Land");

 strcpy(edata.state, "CA");

 edata.zip = 98765;

 /\* Initialize DBTs \*/

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, assign key and data values to DBTs. \*/

 key->data = &ekey;

 key->size = sizeof(ekey);

 data->data = &edata;

 data->size = sizeof(edata);

 /\* Finally, put the data into the database. \*/

 ASSERT(dbp->put(dbp, NULL, &key\_dbt, &data\_dbt, DB\_AUTO\_COMMIT) == 0);

하지만 이 방법은 실행하기 번거롭다는 문제가 있습니다. 데이터베이스의 다른 필드의 값을 미리 알고 있어야 하기 때문입니다. 아래 구문의 경우,

UPDATE employees SET salary = 2000000 WHERE empid = 000100002

employee ID만을 미리 알고 있으면 충분하지만, put 메소드를 이용한 방법에서는 모든 정보를 다 알고 있어야 합니다. Berkeley DB에서도 이런 방법이 가능하지 않을까요? 물론 가능합니다. 업데이트할 데이터 아이템의 바이트 정보를 정확하게 파악하고 있다면, UPDATE 커맨드와 유사한 방법으로 명령을 실행할 수 있습니다.

이를 위해 사용되는 것이 바로 커서(cursor)입니다. 커서는 테이블 상에서의 위치를 표현하기 위해 사용되며, 테이블 상의 현재 위치를 기준으로 반복적인 데이터 처리를 가능하게 합니다.

Berkeley DB에서 커서를 생성하는 것은 쉽습니다. 커서는 데이베이스 핸들을 이용한 메소드로서 지원됩니다:

 DBC \*dbc;

 DB \*dbp;

 ASSERT(dbp->cursor(dbp, NULL, 0) == 0);

이제 생성된 커서를 Mickey의 레코드에 위치시키고 업데이트 작업을 실행해 봅시다. 이를 위해 SQL 구문의 WHERE 절과 유사한 문법이 사용됩니다.

 DBT key\_dbt, data\_dbt;

 emp\_data \*edata;

 emp\_key ekey;

 /\* We'd like to look up Mickey's key. \*/

 emp\_key = 0010002;

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = &emp\_key;

 key\_dbt.size = sizeof(emp\_key);

 /\*

 \* We want the data returned, so we don't need to initialize the

 \* employee data data structure.

 \*/

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, set the cursor to the record with the key emp\_key. \*/

 dbc->c\_get(dbc, &key\_dbt, &data\_dbt, DB\_SET);

다음으로 salary를 변경할 차례입니다.

 /\* Change the salary. \*/

 edata = data\_dbt->data;

 edata.salary = 2000000;

마지막으로 UPDATE 명령을 실행합니다:

 dbc->c\_put(dbc, &key\_dbt, &data\_dbt, DB\_CURRENT);

지금까지 Mickey 레코드에 저장된 salary 값을 미리 알고 있지 않은 상태에서, 데이터를 가져와 업데이트하는 방법을 설명했습니다.

또는, 레코드를 아예 가져오지 않고도 업데이트를 수행할 수도 있습니다. DBT의 DB\_DBT\_PARTIAL 플래그는 레코드의 일부만을 가져오거나 업데이트할 때 사용되며, 이 플래그를 사용하면 해당 부분을 제외한 다른 모든 컨텐트를 무시할 수 있습니다.

다시 아래와 같이 실행합니다:

 emp\_data edata;

 float salary;

 /\* We'd like to look up Mickey's key. \*/

 emp\_key = 0010002;

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = &emp\_key;

 key\_dbt.size = sizeof(emp\_key);

전체 레코드를 가져오는 대신 "PARTIAL get"을 통해 데이터 아이템의 0 바이트를 가져오도록 설정함으로써 데이터의 포지셔닝만을 수행합니다.

 /\* We don't want the data, we just want to position the cursor. \*/

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 data\_dbt->flags = DB\_DBT\_PARTIAL;

 data\_dbt->dlen = 0;

 /\* Position the cursor on Mickey's record \*/

 dbc->c\_get(dbc, &key\_dbt, &data\_dbt, DB\_SET);

 /\*

 \* Now, prepare for a partial put. Note that the DBT has already

 \* been initialized for partial operations. We need to specify

 \* where in the data item we wish to place the new bytes and

 \* how many bytes we'd like to replace.

 \*/

 salary = 2000000.00;

 /\* The DBT contains just the salary information. \*/

 data\_dbt->data = &salary;

 data\_dbt->size = sizeof(salary);

 /\*

 \* dlen and doff tell Berkeley DB where to place this information

 \* in the record. dlen indicates how many bytes we are replacing --

 \* in this case we're replacing the length of the salary field in

 \* the structure (sizeof(emp\_data.salary)). doff indicates where

 \* in the data record we will place these new bytes -- we need to

 \* compute the offset of the salary field.

 \*/

 data\_dbt->dlen = sizeof(emp\_data.salary);

 data\_dbt->doff = ((char \*)&edata.salary - (char \*)&edata);

 /\* Now, put the record back with the new data. \*/

 dbc->c\_put(dbc, &key\_dbt, &data\_dbt, DB\_CURRENT);

**데이터 인출**

지금까지는 테이블에 데이터를 입력하는 방법을 설명했습니다. 이제는 데이터를 가져오는 방법을 설명할 차례입니다. 먼저 가장 간단한 방법부터 알아봅시다. 프라이머리 키를 기준으로 값을 조회하는 방법입니다.

 SELECT \* FROM employees WHERE id=0010002

커서를 이용하여 데이터를 조회하는 방법은 앞에서도 설명한 바 있습니다.

 DBT key\_dbt, data\_dbt;

 emp\_data \*edata;

 emp\_key ekey;

 /\* We'd like to look up Mickey's key. \*/

 emp\_key = 0010002;

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = &emp\_key;

 key\_dbt.size = sizeof(emp\_key);

 /\*

 \* We want the data returned, so we don't need to initialize the

 \* employee data data structure.

 \*/

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, set the cursor to the record with the key emp\_key. \*/

 dbc->c\_get(dbc, &key\_dbt, &data\_dbt, DB\_SET);

앞에서는 레코드 업데이트를 위해 커서를 사용한 바 있습니다. 단순히 레코드를 조회하는 목적이라면 커서를 사용할 필요가 없습니다. dbp 핸들에 대해 get 메소드를 사용하기만 하면 됩니다.

 DBT key\_dbt, data\_dbt;

 emp\_data \*edata;

 emp\_key ekey;

 /\* We'd like to look up Mickey's key. \*/

 emp\_key = 0010002;

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = &emp\_key;

 key\_dbt.size = sizeof(emp\_key);

 /\*

 \* We want the data returned, so we don't need to initialize the

 \* employee data data structure.

 \*/

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, use the dbp method. \*/

 dbp->get(dbp, NULL, &key\_dbt, &data\_dbt, 0);

이 코드는 위에서 사용한 SELECT 구문과 동일합니다.

지금까지 프라이머리 키를 기준으로 레코드를 조회하는 방법을 설명했습니다. 하지만 키를 미리 알고 있지 못하다면 어떻게 해야 할까요? 몇 가지 방법이 아래와 같습니다:

* 2차 키 값을 기준으로 레코드를 조회
* 키를 공유하는 일련의 아이템에 대해 반복(iteration) 작업을 실행
* 데이터베이스에 대해 반복 실행

각각에 대해 좀 더 자세히 설명하겠습니다.

***2차 키의 사용***

SQL에서와 마찬가지로, 2차 키(secondary key)를 기준으로 데이터를 조회하는 방법은 프라이머리 키를 이용하는 방법과 매우 유사합니다.

SQL 쿼리에서 달라지는 부분은 WHERE 절 뿐입니다.

SELECT \* FROM employees WHERE last\_name = "Mouse"

Berkeley DB 호출은 방법은 프라이머리 키를 사용한 경우와 유사합니다.

프라이머리 키를 사용한 예제에서 dbp를 사용한 것과 달리, 2차 키를 이용하는 경우에는 sdbp를 사용합니다:

 DBT key\_dbt, data\_dbt;

 emp\_data \*edata;

 /\* We'd like to look up by Mickey's last name. \*/

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = "Mouse";

 key\_dbt.size = strlen((char \*)key\_dbt.data);

 /\*

 \* We want the data returned, so we don't need to initialize the

 \* employee data data structure.

 \*/

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, call the get method. \*/

 sdbp->get(sdbp, NULL, &key\_dbt, &data\_dbt, 0);

여기서 중요한 것은 data\_dbt에서 반환되는 레코드가 운영 데이터베이스의 데이터라는 사실입니다. 다시 말해, 프라이머리 키를 사용하거나 2차 키를 사용하거나 DBT에 의해 반환되는 데이터는 똑같습니다.

하지만 2차 키를 이용해서 데이터를 조회할 때에는 프라이머리 키 또는 SQL 구문을 이용할 때와 그 결과가 똑같이 표시되지 않음을 확인할 수 있습니다. 프라이머리 키가 제공되지 않았기 때문에 프라이머리 키를 반환할 수가 없는 것이 당연합니다. 위의 코드는 실제로는 다음과 같은 SQL 구문으로 구현됩니다.

 SELECT last\_name, first\_name, salary, street, city, state, zip FROM

 employees WHERE last\_name="Mouse"

프라이머리 키가 필요한 경우는 어떻게 해야 할까요? 그럴 때는 dbp->pget 또는 dbc->pget 메소드를 이용하면 됩니다. 이 두 가지는 get 메소드와 동일한 역할을 수행하지만 2차 인덱스 쿼리에서 프라이머리 키를 반환할 수 있도록 설계되었다는 차이점을 갖습니다. 이 메소드를 실행하면 그 결과로 프라이머리 키, 2차 키, 그리고 데이터 엘리먼트가 함께 반환됩니다.

 DBT key\_dbt, pkey\_dbt, data\_dbt;

 emp\_data \*edata;

 /\* We'd like to look up by Mickey's last name. \*/

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = "Mouse";

 key\_dbt.size = strlen((char \*)key\_dbt.data);

 /\* Set up the dbt into which to return the primary. \*/

 memset(&pkey\_dbt, 0, sizeof(pkey\_dbt));

 /\*

 \* We want the data returned, so we don't need to initialize the

 \* employee data data structure.

 \*/

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, get the record and the primary key. \*/

 sdbp->pget(sdbp, NULL, &key\_dbt, &pkey\_dbt, &data\_dbt, 0);

This code is now equivalent to our SQL secondary selection:

 SELECT \* FROM employees WHERE last\_name="Mouse"

***여러 레코드에 대한 반복 작업***

지금까지는 하나의 레코드에 대해 작업하는 경우만을 설명했습니다. SQL을 이용하면 여러 개의 레코드를 동시에 반환할 수 있습니다. (예를 들어 'Mouse'라는 last\_name을 갖는 모든 직원들을 조회할 수 있습니다.) Berkeley DB에서는 어떻게 할 수 있을까요?

두 가지 경우를 생각해 봅시다. 첫 번째 경우에서는 키를 기준으로 여러 개의 아이템을 조회합니다. 그리고 두 번째 경우에서는 "non-keyed" 필드를 기준으로 데이터베이스 내에서 아이템을 검색합니다.

'Mouse'라는 last\_name을 갖는 모든 직원들을 조회해야 한다고 가정해 봅시다. Mouse라는 성을 가진 직원들은 여러 명이 있을 수 있고, 따라서 last\_name에 대한 2차 인덱스는 중복 데이터를 허용해야 합니다. 따라서 데이터베이스를 열기 전에 중복 데이터를 허용하도록 설정해 주어야 합니다.

 sdbp->set\_flags(sdbp, DB\_DUP);

 ASSERT(sdbp->open(sdbp, NULL, "emp\_lname.db", NULL, DB\_BTREE,

 DB\_AUTO\_COMMIT | DB\_CREATE | DB\_THREAD, 0644) == 0);

2차 인덱스를 기준으로 데이터를 조회하면서 커서를 사용할 필요가 있어 보입니다. 앞에서 사용한 것과 같은 코드를 사용하되, 루프를 추가하여 동일한 2차 키를 공유하는 모든 아이템에 대해 반복 작업이 수행될 수 있도록 합니다.

 DBT key\_dbt, data\_dbt;

 DBC \*sdc;

 emp\_data \*edata;

 /\* We'd like to look up by Mickey's last name. \*/

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = "Mouse";

 key\_dbt.size = strlen((char \*)key\_dbt.data);

 /\*

 \* We want the data and primary key returned, so we need only

 \* initialize the DBTs for them to be returned.

 \*/

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 memset(&pkey\_dbt, 0, sizeof(pkey\_dbt));

 /\* Now, create a cursor. \*/

 sdbp->cursor(sdbp, NULL, &sdbc, 0);

 /\* Now loop over all items with the specified key. \*/

 for (ret = sdbc->pget(sdbc, &key\_dbt, &pkey\_dbt, &data\_dbt, DB\_SET);

 ret == 0:

 ret = sdbc->pget(sdbc, &key\_dbt, &pkey\_dbt, &data\_dbt, DB\_NEXT\_DUP) {

 /\* Do per-record processing in here. \*/

 }

커서는 명시된 키를 기준으로 첫 번째 아이템을 조회하는 시점에 초기화됩니다. 그런 다음, 같은 키를 가진 모든 아이템에 대해 같은 작업을 반복합니다.

키를 기준으로 반복 작업을 수행하는 또 방법으로 아래와 같은 쿼리 형식을 생각해 볼 수 있습니다.

SELECT \* FROM employees WHERE id >= 1000000 AND id < 2000000

여기에서도 반복 작업을 위해 커서가 사용되어야 합니다. 하지만 이번에는 시작 지점과 종료 지점을 설정해 줄 필요가 있습니다. 시작 지점은 Berkeley DB에서 쉽게 설정할 수 있습니다. 종료 지점은 애플리케이션에 의해 설정됩니다.

 DBT key\_dbt, data\_dbt;

 DBC \*dc;

 emp\_key ekey;

 /\* Set the starting point. \*/

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 ekey = 1000000;

 key\_dbt.data = &ekey;

 key\_dbt.size = sizeof(ekey);

 key\_dbt.flags = DB\_DBT\_USERMEM;

 key\_dbt.ulen = sizeof(ekey);

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 /\* Now, create a cursor. \*/

 dbp->cursor(dbp, NULL, &dbc, 0);

 /\* Now loop over items starting with the low key. \*/

 for (ret = dbc->get(dbc, &key\_dbt, &data\_dbt, DB\_SET\_RANGE);

 ret == 0:

 ret = dbc->get(dbc, &key\_dbt, &data\_dbt, DB\_NEXT)) {

 /\* Check if we are still in the range. \*/

 if (ekey >= 2000000)

 break;

 /\* Do per-record processing in here. \*/

 }

여기서 두 가지 사실을 참고할 필요가 있습니다. 1) 루프는 DB\_SET\_RANGE 플래그와 함께 시작됩니다. 이 플래그는 설정된 키보다 크거나 같은 첫 번째 아이템에 커서를 위치시킵니다. 2) 애플리케이션에서 루프의 종료 지점을 확인해 주어야 합니다.

또 key\_dbt에 DB\_DBT\_USERMEM 플래그를 설정함으로써, 조회되는 키가 사용자가 설정한 메모리에 저장될 수 있도록 하고 있습니다. 따라서 ekey 변수를 이용한 키의 확인이 가능합니다.

이제 SELECT 작업에 대한 설명을 마무리하면서, 키를 기준으로 하지 않은 필드를 조건으로 하나 또는 그 이상의 아이템을 조회하는 쿼리를 작성해 봅시다. 아래와 같은 쿼리가 있습니다.

SELECT \* FROM employees WHERE state=ca

state 필드에는 키가 존재하지 않으므로, 전체 데이터베이스에 대해 조회 작업을 반복하는 것 말고는 다른 방법이 없습니다. 따라서 아래와 같은 커서 반복 실행 루프가 필요합니다.

 DBC \*dbc;

 DBT key\_dbt, data\_dbt;

 emp\_data \*edata;

 dbp->cursor(dbp, &key\_dbt, &data\_dbt, &dbc, 0);

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 memset(&data\_dbt, 0, sizeof(data\_dbt));

 for (ret = dbc->get(dbc, &key\_dbt, &data\_dbt, DB\_FIRST);

 ret == 0;

 ret = dbc->get(dbc, &key\_dbt, &data\_dbt, DB\_NEXT)) {

 /\* See if the state field is "ca". \*/

 edata = data\_dbt->data;

 if (strcmp(edata->state, "ca") == 0)

 /\* Keep this record. \*/

 }

이 작업은 효율적이지 않은 것으로 보이지만, 필드에 인덱스가 존재하기 때문에 다른 대안이 없습니다. 또, SQL 데이터베이스에서 인덱스가 적용되지 않은 필드에 대해 쿼리를 실행할 때에도 이와 동일한 작업이 수행됩니다.

**데이터의 삭제**

지금까지 데이터를 입력, 변경, 조회하는 방법에 대해 설명했습니다. 마지막으로 데이터를 삭제하는 방법을 알아 봅시다. 데이터베이스의 튜플을 삭제하는 방법은 기본적으로 두 가지가 있습니다. 삭제하고자 하는 아이템의 키 값을 알고 있다면 "keyed delete"를 수행할 수 있습니다. 키 값을 알지 못한다면 반복 실행을 이용한 "cursor delete" 작업이 필요합니다. 아주 간단한 예로 시작해 봅시다. Mickey Mouse를 해고하려 합니다.

DELETE FROM employees WHERE id= 0010002

SELECT 작업과 매우 유사한 코드를 이용하되, 이번에는 del 메소드를 사용합니다.

 DBT key\_dbt;

 emp\_key ekey;

 ekey = 0010002;

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = &ekey;

 key\_dbt.size = sizeof(ekey);

 dbp->del(dbp, NULL, &key\_dbt, 0);

이번에는 Mouse라는 last\_name을 갖는 모든 직원들을 해고하는 경우를 생각해 봅시다. last\_name에 2차 키가 설정되어 있으므로 기본적인 테크닉은 동일합니다.

DELETE FROM employees WHERE last\_name = "Mouse"

 DBT key\_dbt;

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = "Mouse";

 key\_dbt.size = strlen(key\_dbt.data);

 dbp->del(dbp, NULL, &key\_dbt, 0);

어쩌면 우리가 너무 심했는지도 모릅니다. 모든 Mickey들을 해고하는 대신 Minnie Mouse만 해고하는 것으로 충분할 것 같습니다. Minnie를 쉽게 해고할 수 있는 방법이 있을까요? 다시 말해, 아래의 SQL 작업을 어떻게 구현해야 할까요?

DELETE FROM employees where last\_name = "Mouse" AND first\_name = "Minnie"

당연한 얘기지만, 데이터 아이템에 대해 커서를 반복 실행하면서 삭제할 아이템을 찾아야 합니다.

 DBT key\_dbt, data\_dbt;

 DBC \*sdbc;

 sdbp->cursor(sdbp, NULL, &sdbc, 0);

 memset(&key\_dbt, 0, sizeof(key\_dbt));

 key\_dbt.data = "Mouse";

 key\_dbt.size = strlen(key\_dbt.data);

 for (ret = sdbc->get(sdbc, &key\_dbt, &data\_dbt, DB\_SET);

 ret == 0;

 ret = sdbc->get(sdbc, &key\_dbt, &data\_dbt, DB\_NEXT\_DUP)) {

 edata = data\_dbt->data;

 if (strcmp(edata->first\_name, "Minnie") == 0) {

 /\* OK, this is a record we want to delete. \*/

 sdbc->del(sdbc, 0);

 }

 }

 dbp->del(dbp, NULL, &key\_dbt, 0);

< 방법을 처리하는 통해 트랜잭션을 작업을 데이터베이스 마지막으로 이제 있습니다. 지원하고 설정을 옵션과 다양한 필요한 구현하는데 기능을 복잡한 보다 말고도 이것 DB는 Berkeley 설명했습니다. 개략적으로 작성하는 함수를 DB Oracle 위해 실행하기 커맨드를 SQL 기본적인>

**트랜잭션의 관리**

트랜잭션이 SQL 환경에서 동작하는 원리에 대해 잠시 생각해 봅시다. 사용자가 SQL에서 DML 구문을 실행했을 때, 구문은 현재 트랜잭션의 일부로서 수행됩니다. 이후 실행되는 모든 구문들 역시 같은 트랜잭션에 포함됩니다. 트랜잭션은 SQL 세션이 종료되거나 애플리케이션이 COMMIT 구문을 실행했을 때 커밋 처리됩니다. ROLLBACK 구문을 실행하면 어느 시점에서든 트랜잭션을 취소할 수 있습니다.

SQL 환경에 AUTOCOMMIT 기능이 활성화되어 있는 경우에는 각각의 DML 구문이 별도의 트랜잭션으로 간주됩니다. AUTOCOMMIT 모드가 활성화된 상태에서 아래의 구문은,

 statement 1

 COMMIT

 statement 2

 COMMIT

 statement 3

 COMMIT

아래 구문과 동일합니다.

 statement 1

 statement 2

 statement 3

Berkeley DB는 트랜잭션에 데이터베이스 작업을 인캡슐레이트 하는 기능을 지원합니다. SQL과 달리 트랜잭션을 사용하지 않고 Berkeley DB를 실행할 수도 있습니다. 실제로, Berkeley DB에서는 명시적으로 트랜잭션을 요청하지 않는 이상 트랜잭션이 사용되지 않습니다. 그렇다면 Berkeley DB에서 트랜잭션을 요청하려면 어떻게 해야 할까요?

환경을 오픈하면서 플래그를 설정하면 됩니다.

 DB\_ASSERT(dbenv->open(dbenv, "my\_databases/personnel",

 DB\_CREATE | DB\_INIT\_LOCK | DB\_INIT\_MPOOL | DB\_INIT\_TXN | DB\_THREAD,

 0644);

이 플래그들은 애플리케이션을 위해 Berkeley DB를 설정해 줍니다. 위에서는 DB\_INIT\_TXN 플래그가 명시되어 있으므로 트랜잭션이 활성화됩니다. 이 플래그가 설정되지 않는다면 애플리케이션은 트랜잭션을 사용하지 않은 상태에서 실행될 것입니다.

Berkeley DB는 SQL의 AUTOCOMMIT과 유사한 기능을 제고합니다. 환경 핸들(environment handle)에서 set\_flags 메소드를 사용하면 전체 데이터베이스(환경)에서 항상 autocommit를 사용하도록 설정할 수 있습니다.

 dbenv->set\_flags(dbenv, DB\_AUTOCOMMIT, 1);

또는 데이터베이스 오픈 시점에 DB\_AUTOCOMMIT을 설정하여, 이후 실행되는 모든 작업에서 트랜잭션을 실행하도록 할 수도 있습니다.

이번에는 AUTOCOMMIT 옵션을 사용하지 않고 애플리케이션에서 임의의 작업들을 논리적인 트랜잭션으로 묶는 방법에 대해 알아 보겠습니다. Mickey Mouse를 employees 테이블에 추가하고 Mickey의 관리자를 지정해 봅시다.

INSERT INTO employees VALUES (00010002, "mouse", "mickey", 1000000.00,

 "Main Street", "Disney Land", "CA", 98765);

INSERT INTO manages(00000001, 000100002)

COMMIT

(위의 구문은 id=00000001 조건을 만족하는 직원이 Mickey의 관리자임을 의미합니다.)

여러분들이 이미 데이터 작업을 실행하는 방법을 잘 알고 있다고 가정하고, 트랜잭션을 지정하는 방법에만 초점을 맞추어 설명하겠습니다.

먼저, 트랜잭션을 명시적으로 시작합니다. 트랜잭션을 생성하는 작업은 환경(environment) 레벨에서 수행되며, 따라서 환경 핸들의 메소드를 통해 처리되어야 합니다. 이 메소드는 트랜잭션 핸들(DB\_TXN)을 생성합니다.

 DB\_TXN \*txn;

 dbenv->txn\_begin(dbenv, NULL, &txn, 0);

이제 생성된 트랜잭션 핸들을 트랜잭션에 포함시킬 임의의 데이터베이스 작업에 전달할 수 있습니다.

 emp\_dbp->put(emp\_dbp, txn, &key, &data, 0);

 man\_dbp->put(man\_dbp, txn, &key, &data, 0);

그런 다음, 트랜잭션 핸들의 메소드를 호출하는 방법으로 트랜잭션을 커밋하거나 취소할 수 있습니다.

트랜잭션을 커밋하는 방법이 아래와 같습니다.

 txn->commit(txn, 0);

트랜잭션을 취소하는 방법이 아래와 같습니다.

 txn->abort(txn);

두 가지 메소드 모두 "destructor"로, 트랜잭션 핸들을 더 이상 사용할 수 없게 만듭니다.

SQL과 달리 Berkeley DB에서는 트랜잭션을 이용해서 DDL 작업을 보호하는 것도 가능합니다. 따라서 DB\_TXN 핸들을 dbenv->dbremove, dbenv->dbrename, dbp->open(… DB\_CREATE …)과 같은 작업에도 전달할 수 있습니다. DDL 작업은 설정된 트랜잭션의 컨텍스트 내에서 실행되며, 다른 트랜잭션과 마찬가지로 커밋 또는 취소가 가능합니다.

**결론**

Oracle Berkeley DB는 SQL 데이터베이스와 동일한 유형의 기능을 제공하고 있지만, 사용되는 패키지는 전혀 다릅니다. 개발자는 프로그램 코드를 이용하여 API를 호출해야 하며, 전체 데이터베이스는 애플리케이션 내부에 직접 "임베드" 됩니다. (다시 말해, 애플리케이션과 데이터베이스가 동일한 주소 공간에서 실행됩니다.) 그 결과로 수십 배에 이르는 성능 개선 효과를 거둘 수 있습니다. 하지만 이러한 성능 효과의 대가로 애플리케이션이 감당해야 할 부담이 커지는 문제가 있습니다. Oracle Berkeley DB는 애플리케이션이 매우 예외적인 수준의 고성능을 요구하는 경우, 또는 애플리케이션이 기본적으로 관계형 속성을 갖지 않는 데이터를 처리해야 하는 경우 유용하게 활용될 수 있습니다.

**Margo Seltzer**는 오라클 Berkeley DB 그룹의 아키텍트입니다. 그녀는 Berkeley DB를 최초 설계하고 Sleepycat Software를 공동 설립한 그룹의 일원입니다. Margo는 또 Herchel Smith의 컴퓨터 과학 교수이며 Harvard School of Engineering and Applied Sciences의 교수로 재직 중입니다. 그녀는 파일 시스템, 데이터베이스, 트랜잭션 프로세싱 시스템을 주요 연구 분야로 하고 있습니다.