http://www.yiiframework.com/doc/guide/1.1/id/database.overview

Bekerja dengan Database

Yii menyediakan dukungan berkemampuan untuk pemrograman database.

Dibangun di atas extension PHP Data Objects (PDO), Yii Data Access Objects (DAO) memungkinkan pengaksesan ke sistem manajemen database (DBMS) yang berbeda dalam satu antar muka tunggal yang seragam. Aplikasi yang dikembangkan menggunakan Yii DAO dapat dialihkan dengan mudah ke DBMS berbeda tanpa perlu memodifikasi data pengaksesan code.

Yii Query Builder menyediakan sebuah method berorientasi objek untuk membuat query SQL, yang bisa mengurangi resiko terserang SQL injection.

Dan Active Record Yii, diimplementasikan sebagai pendekatan Pemetaan Relasional-Obyek / Object-Relational Mapping (ORM) yang diadopsi secara luas, mempermudah pemrograman database. Tabel direpresentasikan dalam bentuk kelas dan baris dalam bentuk instance, Yii AR mengeliminasi tugas berulang pada penulisan SQL statement terutama yang berkaitan dengan operasi CRUD (create, read, update dan delete).

Meskipun Yii menyertakan fitur-fitur database yang dapat menangani hampir semua tugas-tugas terkait-database, Anda masih bisa menggunakan pustaka database Anda sendiri dalam aplikasi Yii Anda. Bahkan, Yii framework didesain secara hati-hati agar bisa dipakai bersamaan dengan pustaka pihak ketiga lainnya.

Data Access Objects (DAO)

Data Access Objects (DAO) atau Objek Akses Data menyediakan API generik untuk mengakses data yang disimpan dalam sistem manajemen database (DBMS) yang berbeda. Hasilnya, Lapisan DBMS dapat diubah ke yang lain yang berbeda tanpa memerlukan perubahan kode yang menggunakan DAO untuk mengakses data.

Yii DAO dibangun di atas PHP Data Objects (PDO) yang merupakan extension yang menyediakan akses data gabungan untuk beberapa DBMS populer, seperti MySQL, PostgreSQL. Oleh karena itu, untuk menggunakan Yii DAO, extension PDO dan driver database PDO tertentu (misalnya PDO_MYSQL) harus sudah terinstal.

Yii DAO terdiri dari empat kelas utama sebagai berikut:

- CDbConnection: mewakili koneksi ke database.
- CDbCommand: mewakili pernyataan SQL untuk dijalankan pada database.
- CDbDataReader: mewakili forward-only stream terhadap baris dari set hasil query.
- CDbTransaction: mewakili transaksi DB.

Berikutnya, kami memperkenalkan pemakaian Yii DAO dalam skenario berbeda.

1. Membuat Koneksi Database

Untuk membuat koneksi database, buat instance CDbConnection dan mengaktifkannya. Nama sumber data (DSN) diperlukan untuk menetapkan informasi yang diperlukan untuk menyambung ke database. Nama pengguna dan kata sandi juga diperlukan guna melakukan koneksi. Exception akan dimunculkan seandainya kesalahan terjadi selama pelaksanaan koneksi (misalnya DSN tidak benar atau username/password tidak benar).

```
$connection=new CDbConnection($dsn,$username,$password);
// melakukan koneksi. Anda dapat mencoba try...catch exception
yang mungkin
$connection->active=true;
......
$connection->active=false;  // tutup koneksi
```

Format DSN tergantung pada driver PDO database yang digunakan. Secara umum, DSN terdiri dari nama driver PDO, diikuti oleh titik dua, diikuti oleh sintaks koneksi spesifik-driver. Lihat Dokumentasi PDO untuk informasi lebih lengkap. Di bawah ini adalah daftar format DSN yang umum dipakai:

- SQLite: `sqlite:/path/to/dbfile`
- MySQL: `mysql:host=localhost;dbname=testdb`
- PostgreSQL: `pgsql:host=localhost;port=5432;dbname=testdb`
- SQL Server: `mssql:host=localhost;dbname=testdb`
- Oracle: `oci:dbname=//localhost:1521/testdb`

Karena CDbConnection diturunkan dari CApplicationComponent, kita juga dapat menggunakannya sebagai komponen aplikasi. Untuk melakukannya, konfigurasi dalam komponen aplikasi `db` (atau nama lain) pada konfigurasi aplikasi sebagai berikut,

```
array(
    ......
    'components'=>array(
        ......
        'db'=>array(
            'class'=>'CDbConnection',
            'connectionString'=>'mysql:host=localhost;dbname=testd
b',
            'username'=>'root',
            'password'=>'password',
            'emulatePrepare'=>true,  //  pada beberapa instalasi
MySQL, diperlukan
        ),
    ),
)
```

Selanjutnya kita dapat mengakses koneksi DB via `Yii::app()->db` yang sudah diaktifkan secara otomatis, kecuali dikonfigurasi secara eksplisit CDbConnection::autoConnect menjadi false. Menggunakan pendekatan ini, koneksi DB tunggal dapat dibagi dalam tempat multipel pada kode kita.

2. Menjalankan Pernyataan SQL

Setelah koneksi database terlaksana, pernyataan SQL dapat dijalankan menggunakan CDbCommand. Membuat instance CDbCommand dengan memanggil CDbConnection::createCommand() dengan pernyataan SQL yang ditetapkan:

```
$connection=Yii::app()->db;   // asumsi bahwa Anda memiliki
koneksi "db" yang terkonfigurasi
// Jika tidak, Anda bisa membuat sebuah koneksi secara eksplisit
// $connection=new CDbConnection($dsn,$username,$password);
$command=$connection->createCommand($sql);
// jika diperlukan, statement SQL dapat diupdate sebagai berikut
// $command->text=$newSQL;
```

Pernyataan SQL dijalankan via CDbCommand dalam dua cara berikut:

- execute(): melakukan pernyataan SQL non-query, seperti INSERT, UPDATE and DELETE. Jika berhasil, mengembalikan sejumlah baris yang dipengaruhi oleh eksekusi
- query(): melakukan pernyataan SQL yang mengembalikan baris data, seperti SELECT. Jika berhasil,mengembalikan instance CDbDataReader yang dapat ditelusuri baris data yang dihasilkan. Untuk kenyamanan, satu set metode queryXXX() juga diimplementasikan yang secara langsung mengembalikan hasil query.

Exception akan dimunculkan jika kesalahan terjadi selama eksekusi pernyataan SQL.

```
$rowCount=$command->execute();   // jalankan SQL non-query
$dataReader=$command->query();   // jalankan query SQL
```

```
$rows=$command->queryAll();      // query dan kembalikan seluruh
baris hasil
$row=$command->queryRow();       // query dan kembalikan baris
pertama hasil
$column=$command->queryColumn(); // query dan kembalikan kolom
pertama hasil
$value=$command->queryScalar();  // query dan kembalikan field
pertama dalam baris pertama
```

3. Mengambil Hasil Query

Setelah CDbCommand::query() membuat instance CDbDataReader, Anda bisa mengambil baris data yang dihasilkan oleh pemanggilan CDbDataReader::read() secara berulang. Ia juga dapat menggunakan CDbDataReader dalam konsrruksi bahasa PHP foreach untuk mengambil baris demi baris.

```
$dataReader=$command->query();
// memanggil read() secara terus menerus sampai ia mengembalikan
false
while(($row=$dataReader->read())!==false) { ... }
// menggunakan foreach untuk menelusuri setiap baris data
foreach($dataReader as $row) { ... }
// mengambil seluruh baris sekaligus dalam satu array tunggal
$rows=$dataReader->readAll();
```

Catatan: Tidak seperti query(), semua metode queryXXX() mengembalikan data secara langsung. Sebagai contoh, queryRow() mengembalikan array yang mewakili baris pertama pada hasil query.

4. Menggunakan Transaksi

Ketika aplikasi menjalankan beberapa query, setiap pembacaan dan/atau penulisan informasi dalam database, penting untuk memastikan bahwa database tidak tertinggal dengan hanya beberapa query yang dihasilkan. Transaksi diwakili oleh instance CDbTransaction dalam Yii, dapat diinisiasi dalam hal:

- Mulai transaksi.
- Jalankan query satu demi satu. Setiap pemutakhiran pada database tidak terlihat bagi dunia luar.
- Lakukan transaksi. Pemutakhiran menjadi terlihat jika transaksi berhasil.
- Jika salah satu query gagal, seluruh transaksi dibatalkan.

Alur kerja di atas dapat diimplementasikan menggunakan kode berikut:

```
$transaction=$connection->beginTransaction();
try
{
    $connection->createCommand($sql1)->execute();
    $connection->createCommand($sql2)->execute();
    //.... eksekusi SQL lainnya
```

```
    $transaction->commit();
}
catch(Exception $e) // exception dimunculkan jika query gagal
{
    $transaction->rollBack();
}
```

5. Mengikat Parameter

Untuk menghindari serangan injeksi SQL dan meningkatkan kinerja pelaksanaan pernyataan SQL secara terus menerus, Anda dapat "menyiapkan" sebuah pernyataan SQL dengan opsional penampung parameter yang akan diganti dengan parameter sebenarnya selama proses pengikatan parameter.

Penampung parameter dapat bernama (disajikan sebagai token unik) ataupun tidak bernama (disajikan sebagai tanda tanya). Panggil CDbCommand::bindParam() atau CDbCommand::bindValue() untuk mengganti penampung ini dengan parameter sebenarnya. Parameter tidak harus bertanda kutip: lapisan driver database melakukan ini bagi Anda. Pengikatan parameter harus dikerjakan sebelum pernyataan SQL dijalankan.

```
// SQL dengan dua penampung ":username" and ":email"
$sql="INSERT INTO tbl_user (username, email)
VALUES(:username,:email)";
$command=$connection->createCommand($sql);
// ganti penampung ":username" dengan nilai username sebenarnya
$command->bindParam(":username",$username,PDO::PARAM_STR);
// ganti penampung ":email" dengan nilai email sebenarnya
$command->bindParam(":email",$email,PDO::PARAM_STR);
$command->execute();
// sisipkan baris lain dengan set baru parameter
$command->bindParam(":username",$username2,PDO::PARAM_STR);
$command->bindParam(":email",$email2,PDO::PARAM_STR);
$command->execute();
```

Metode bindParam() dan bindValue() sangat mirip. Perbedaannya hanyalah bahwa bidParam mengikat parameter dengan referensi variabel PHP sedangkan bindValue dengan nilai. Untuk parameter yang mewakili memori blok besar data, bindParam dipilih dengan pertimbangan kinerja.

Untuk lebih jelasnya mengenai pengikatan parameter, lihat dokumentasi PHP relevan.

6. Mengikat Kolom

Ketika mengambil hasil query, Anda dapat mengikat kolom ke variabel PHP dengan demikian hasil akan dipopulasi secara otomatis dengan data terbaru setiap kali baris diambil.

```
$sql="SELECT username, email FROM tbl_user";
```

```
$dataReader=$connection->createCommand($sql)->query();
// ikat kolom ke-1 (username) ke variabel $username
$dataReader->bindColumn(1,$username);
// ikat kolom ke-2 (email) ke variabel $email
$dataReader->bindColumn(2,$email);
while($dataReader->read()!==false)
{
    // $username dan $email berisi username dan email pada baris
saat ini
}
```

7. Menggunakan Prefiks Tabel

Yii menyediakan dukungan terintegrasi untuk menggunakan prefiks tabel. Prefiks tabel merupakan sebuah string yang dipasang di depan nama tabel yang sedang terkoneksi ke database. Kebanyakan digunakan pada lingkungan shared-hosting yang mana berbagai aplikasi saling pakai sebuah database dan menggunakan prefiks tabel yang berbeda untuk saling membedakan. Misalnya, satu aplikasi bisa menggunakan tbl_ sebagai prefiks sedangkan aplikasi yang lain bisa saja menggunakan yii_

Untuk menggunakan prefiks tabel, mengkonfigurasi properti CdbConnection::tablePrefix menjadi sesuai dengan prefiks tabel yang diinginkan. Kemudia, dalam statement SQL gunakan {{NamaTabel}} untuk merujuk nama tabel, di mana NamaTabel adalah nama table tanpa prefik. Misalnya, jika database terdapat sebuah tabel bernama tbl_user di mana tbl_ dikonfigurasi sebagai prefiks tabel, maka kita dapat menggunakan kode berikut untuk query user.

```
$sql='SELECT * FROM {{user}}';
$users=$connection->createCommand($sql)->queryAll();
```

Query Builder

1. Mempersiapkan Query Builder
2. Membangung Query Penarik Data
3. Membuat Query Manipulasi Data
4. Membuat Query Manipulasi Schema

Yii Query Builder menyediakan cara berorientasi objek dalam menulis statement SQL. Fitur ini membantu pengembang untuk menggunakan property dan method kelas untuk menentukan bagian-bagian dari statement SQL yang kemudian menggabungkannya menjasi sebuah statement SQL yang valid yang bisa dieksekusi lebih lanjut oleh method DAO seperti yang dideskripsikan di Data Access Objects. Berikut akan ditampilkan penggunaan umum dari Query Builder untuk membangun sebuah statement SQL SELECT:

```
$user = Yii::app()->db->createCommand()
    ->select('id, username, profile')
    ->from('tbl_user u')
    ->join('tbl_profile p', 'u.id=p.user_id')
    ->where('id=:id', array(':id'=>$id))
    ->queryRow();
```

Query Builder sangat cocok digunakan ketika Anda memerlukan menggabungkan sebuah statement SQL secara prosedural, atau berdasarkan suatu kondisi logis dalam aplikasi Anda. Manfaat utama dalam menggunakan Query Builder termasuk:

- Memungkinkan membangun statement SQL yang kompleks secara programatik
- Fitur ini akan memberikan quote pada nama table dan kolom secara otomatis guna mencegah konflik dengan tulisan SQL ataupun karakter khusus.
- Fitur ini juga memberikan quote pada nilai parameter dan melakukan binding pada parameter ketika memungkinkan, sehingga mengurangi resiko terserang SQL injection.
- Fitur ini menyediakan sekian tingkatan abstraksi pada DB, yang menyederhanakan migrasi ke platform DB yang berbeda.

Menggunakan Query Builder bukanlah sebuah keharusan. Bahkan, jika query Anda cukup sederhana, akan leih gampang dan cepat menulis SQL-nya langsung.

Catatan: Query builder tidak bisa digunakan untuk memodifikasi query yang sudah ada sebagai statement SQL. Misalnya, code berikut tidak akan berjalan:

```
$command = Yii::app()->db->createCommand('SELECT * FROM tbl_user');
// baris berikut tidak akan menambahkan WHERE ke klausa SQL di atas.
$command->where('id=:id', array(':id'=>$id));
```

Oleh karena itu, jangan campur penggunaan SQL biasa dengan query builder.

1. Mempersiapkan Query Builder

Query Builder Yii disediakan oleh CDbCommand, kelas query DB utama di Data Access Objects.

Untuk menggunakan Query Builder, kita membuat sebuah instance baru dari CDbCommand dengan cara berikut,

```
$command = Yii::app()->db->createCommand();
```

Begitulah, kita menggunakan `Yii::app()->db` untuk mendapatkan koneksi DB, kemudian melakukan pemanggilanpada CDbConnection::createCommand() untuk membuat instance command yang diperlukan.

Perhatikan bahwa Sebagai ganti dari kita mem-pass semua statement SQL ke `createCommand()` seperti yang dilakukan di Data Access Objects, kita membiarkannya kosong. Ini dikarenakan kita akan membangun bagian-bagian

individu dari statement SQL dengan menggunakan method Query Builder yang akan dijelaskan pada bagian berikut.

2. Membangung Query Penarik Data

Query penarik data merujuk pada statement SELECT pada SQL. Query builder menyediakan sekumpulan method untuk membangun bagian dari statement SELECT. DIkarenakan semua method ini mengembalikan instance CDbCommand, kita dapat memanggil mereka dengan menggunakan method chaining, seperti pada contoh di awal.

- select(): menentukan bagian SELECT pada query
- selectDistinct(): menentukan bagian SELECT pada query serta mengaktifkan flag DISTINCT
- from(): menentukan bagian FROM pada query
- where(): menentukan bagian WHERE pada query
- join(): menambah pecahan query inner join
- leftJoin(): menambah pecahan left query left outer join
- rightJoin(): menambah pecahan query right outer join
- crossJoin(): menambah pecahan query cross join
- naturalJoin(): menambah pecahan query natural join
- group(): menentukan bagian GROUP BY pada query
- having(): menentukan bagian HAVING pada query
- order(): menentukan bagian ORDER BY pada query
- limit(): menentukan bagian LIMIT pada query
- offset(): menentukan bagian OFFSET pada query
- union(): menentukan bagian UNION pada query

Berikut, kami akan menunjukkan bagaimana menggunakan method-method query builder ini. Supaya sederhana, kami mengasumsi database yang digunakan adalah MySQL. Perhatikan bahwa jika Anda menggunakan DBMS yang lain, quote table/kolom/nilai akan berbeda dengan contoh.

select()

```
function select($columns='*')
```

Method select() menentukan bagian `SELECT` pada query. Parameter `$columns` menentukan kolom-kolom apa saja yang akan di-select, yang bisa berupa string dengan nama kolom dipisah koma, atau sebuah array dari nama kolom. Nama kolom dapat berisi prefiks table dan/atau alias kolom. Method ini akan secara otomatis memberikan quote pada nama kolom kecuali kolom tersebut mengandung tanda kurung (yang bararti kolom yang diberikan merupakan ekspresi DB).

Berikut ini merupakan beberapa contoh:

```
// SELECT *
select()
// SELECT `id`, `username`
```

```
select('id, username')
// SELECT `tbl_user`.`id`, `username` AS `name`
select('tbl_user.id, username as name')
// SELECT `id`, `username`
select(array('id', 'username'))
// SELECT `id`, count(*) as num
select(array('id', 'count(*) as num'))
```

selectDistinct()

```
function selectDistinct($columns)
```

Method selectDistinct() mirip dengan select(). Hanya saja selectDistinct mengaktifkan flag `DISTINCT`. Misalnya, `selectDistinct(id,username')` akan menghasilkan SQL berikut:

```
SELECT DISTINCT `id`, `username`
```

from()

```
function from($tables)
```

Method from() menentukan bagian `FROM` pada query. Parameter `$tables` menentukan table mana yang akan di-select. Yang ini juga bisa berupa string dengan nama table dipisahkan dengan koma, atau sebuah array dari nama table. Nama table dapat diambil dari prefiks skema (misalnya `public.tbl_user`) dan/atau alias table (misalnya `tbl_user u`). Method ini akan secara otomatis memberikan quote pada nama table kecuali nama table-nya mengandung huruf kurung (yang artinya berupa sub-query atau ekspresi DB).

Berikut merupakan beberapa contoh:

```
// FROM `tbl_user`
from('tbl_user')
// FROM `tbl_user` `u`, `public`.`tbl_profile` `p`
from('tbl_user u, public.tbl_profile p')
// FROM `tbl_user`, `tbl_profile`
from(array('tbl_user', 'tbl_profile'))
// FROM `tbl_user`, (select * from tbl_profile) p
from(array('tbl_user', '(select * from tbl_profile) p'))
```

where()

```
function where($conditions, $params=array())
```

Method where() menetapkan bagian `WHERE` pada query. Parameter `$conditions` menentukan kondisi query sedangkan `$params` menentukan parameter yang diikat pada keseluruhan query. Parameter `$conditions` dapat berupa sebuah string (misalnya `id=1`) atau sebuah array dengan format:

```
array(operator, operand1, operand2, ...)
```

dengan `operator`dapat bisa berupa :

- `and`: operan harus digabung dengan menggunakan `AND`. Misalnya `array('and', 'id=1', 'id=2')` akan menghasilkan `id=1 AND id=2`. Jika operan adalah array, maka akan diubah menjadi string dengan menggunakan aturan yang sama. Misalnya `array('and', 'type=1', array('or', 'id=1', 'id=2'))` akan menghasilkan `type=1 AND (id=1 OR id=2)`. Method ini tidak akan memberikan quote ataupun escape character.
- `or`: mirip dengan operator `and` hanya saja operan-operan akan digabung dengan OR.
- `in`: Operan satu harus berupa kolom atau ekspresi DB, dan operan 2 harus berupa array yang merepresentasikan kumpulan nilai yang harus dipenuhi oleh kolom atau ekspresi DB. Misalnya `array('in', 'id', array(1,2,3))` akan menghasilkan `id IN (1,2,3)`. Method ini akan memberikan quote pada nama kolom dan nilai escape di dalam range.
- `not in`: mirip dengan operator `in` kecuali tulisan `IN` akan diubah dengan `NOT IN` di dalam kondisi yang di-generate.
- `like`: operan 1 harus berupa kolom atau ekspresi DB, dan operan 2 harus berupa string atau sebuah array yang mewakili range dari nilai-nilai di kolom atau ekspresi DB yang mirip. Misalnya, `array('like', 'name', '%tester%')` akan menghasilkan `name LIKE '%tester%'`. Ketika range nilai diberikan sebagai array, maka beberapa predikat `LIKE` akan di-generate dan digabungkan dengan menggunakan `AND`. Misalnya `array('like', 'name', array('%test%', '%sample%'))` akan menghasilkan `name LIKE '%test%' AND name LIKE '%sample%'`. Method ini akan memberikan quote pada nama kolom dan nilai escape pada range nilai.
- `not like`: mirip dengan operator `like` kecuali tulisan `LIKE` akan diganti dengan `NOT LIKE` pada kondisi yang dihasilkan.
- `or like`: mirip dengan operator `like` hanya saja tulisan `OR` yang digunakan untuk menggabungkan beberapa predikat `LIKE`.
- `or not like`: mirip dengan operator `not like` kecuali `OR` yang digunakan untuk menggabungkan predikat `NOT LIKE`.

Berikut merupakan beberapa contoh yang menggunakan `where`:

```
// WHERE id=1 or id=2
where('id=1 or id=2')
// WHERE id=:id1 or id=:id2
where('id=:id1 or id=:id2', array(':id1'=>1, ':id2'=>2))
// WHERE id=1 OR id=2
where(array('or', 'id=1', 'id=2'))
// WHERE id=1 AND (type=2 OR type=3)
where(array('and', 'id=1', array('or', 'type=2', 'type=3')))
// WHERE `id` IN (1, 2)
where(array('in', 'id', array(1, 2))
// WHERE `id` NOT IN (1, 2)
where(array('not in', 'id', array(1,2)))
// WHERE `name` LIKE '%Qiang%'
where(array('like', 'name', '%Qiang%'))
// WHERE `name` LIKE '%Qiang' AND `name` LIKE '%Xue'
where(array('like', 'name', array('%Qiang', '%Xue')))
// WHERE `name` LIKE '%Qiang' OR `name` LIKE '%Xue'
where(array('or like', 'name', array('%Qiang', '%Xue')))
```

```
// WHERE `name` NOT LIKE '%Qiang%'
where(array('not like', 'name', '%Qiang%'))
// WHERE `name` NOT LIKE '%Qiang%' OR `name` NOT LIKE '%Xue%'
where(array('or not like', 'name', array('%Qiang%', '%Xue%')))
```

Perhatikan bahwa ketika menggunakan operator `like`, kita harus menentukan karakter wildcard secara eksplisit (seperti `%` dan _) . JIka polanya berasal dari input user, maka kita harus menggunakan code berikut untuk escape karakter spesial guna menghindarinya dianggap sebagai wildcard:

```
$keyword=$_GET['q'];
// escape % and _ characters
$keyword=strtr($keyword, array('%'=>'\%', '_'=>'\_'));
$command->where(array('like', 'title', '%'.$keyword.'%'));
```

order()

```
function order($columns)
```

Method order() menentukan bagian `ORDER BY` pada query. Parameter `$columns` menentukan kolom-kolom yang diurutkan. Dapat berupa sebuah string dengan kolom yang dipisahkan koma dan arah pengurutan (`ASC` atau `DESC`), atau sebuah array dari kolom dan arah pengurutan. Nama kolom dapat mengandung prefiks table. Method ini akan memberikan quote pada nama kolom secara otomatis kecuali kolom tersebut mengandung tanda kurung (yang berarti kolom tersebut merupakan ekspresi DB).

Berikut merupakan beberapa contohnya:

```
// ORDER BY `name`, `id` DESC
order('name, id desc')
// ORDER BY `tbl_profile`.`name`, `id` DESC
order(array('tbl_profile.name', 'id desc'))
```

limit() dan offset()

```
function limit($limit, $offset=null)
function offset($offset)
```

Method limit() dan offset() menentukan bagian `OFFSET` dan `LIMIT` pada query. Perhatikan bahwa beberapa DBMS mungkin tidak mendukung sintaks `LIMIT` dan `OFFSET`. Pada kasus tersebut, Query Builder akan menulis ulang seluruh statement SQL untuk mensimulasi fungsi limit dan offset.

Berikut merupakan beberapa contoh:

```
// LIMIT 10
limit(10)
// LIMIT 10 OFFSET 20
limit(10, 20)
// OFFSET 20
offset(20)
```

join() dan varian-variannya

```
function join($table, $conditions, $params=array())
function leftJoin($table, $conditions, $params=array())
function rightJoin($table, $conditions, $params=array())
function crossJoin($table)
function naturalJoin($table)
```

Method join() dan varian-variannya menentukan bagaimana melakukan join dengan table lain dengan menggunakan `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `CROSS JOIN`, atau `NATURAL JOIN`. Parameter `$table` menentukan table mana yang akan dijoin. Nama table akan mengandung prefiks skema dan /atau alias. Method ini akan memberikan quote pada nama table kecuali kolom tersebut mengandung tanda kurung yang artinya bis berupa ekspresi DB atau sub-query. Parameter `$conditions` menentukan kondisi join. Sintaksnya sama dengan where(). Dan `$params` menentukan parameter yang diikat pada keseluruhan query.

Perhatikan bahwa tidak seperti method query builder lainnya, setiap pemanggilan method join akan ditambahkan di belakang sebelumnya.

Berikut merupakan beberapa contohnya.

```
// JOIN `tbl_profile` ON user_id=id
join('tbl_profile', 'user_id=id')
// LEFT JOIN `pub`.`tbl_profile` `p` ON p.user_id=id AND type=1
leftJoin('pub.tbl_profile p', 'p.user_id=id AND type=:type',
array(':type'=>1))
```

group()

```
function group($columns)
```

Method group() menetapkan bagian `GROUP BY` pada query. Parameter `$columns` menentukan kolom-kolom yang dikelompokkan. Bisa berupa string yang berisi kolom dipisah koma, atau sebuah array dari kolom. Nama kolom bisa didapatkan pada prefiks table. Method ini akan secara otomatis memberikan quote pada nama kolom kecuali terdapat sebuah kolom yang mengandung tanda kurung (yang artinya kolom tersebut merupakan ekspresi DB).

Berikut beberapa contoh:

```
// GROUP BY `name`, `id`
group('name, id')
// GROUP BY `tbl_profile`.`name`, `id`
group(array('tbl_profile.name', 'id')
```

having()

```
function having($conditions, $params=array())
```

Method having() menetapkan bagian `HAVING` pada query. Penggunaannya sama dengan where().

Berikut contoh-contohnya:

```
// HAVING id=1 or id=2
having('id=1 or id=2')
// HAVING id=1 OR id=2
having(array('or', 'id=1', 'id=2'))
```

union()

```
function union($sql)
```

Method union() menentukan bagian `UNION` pada query. Method ini akan menambahkan `$sql` ke SQL yang sudah ada dengan menggunakan operator `UNION`. Memanggil `union()` beberapa kali akan menambahkan berkali-kali SQL-SQL-nya ke belakang SQL yang sudah ada.

Contoh:

```
// UNION (select * from tbl_profile)
union('select * from tbl_profile')
```

Menjalankan Query

Setelah melakukan pemanggilan method query builder di atas, kita dapat memanggil method DAO seperti yang dijelaskan pada Data Access Objects untuk mengeksekusi query. Misalnya, kita dapat memanggil CDbCommand::queryRow() untuk mendapatkan sebaris hasil atau CDbCommand::queryAll() untuk mendapatkan seluruhnya sekaligus. Berikut beberapa contohnya :

```
$users = Yii::app()->db->createCommand()
    ->select('*')
    ->from('tbl_user')
    ->queryAll();
```

Mengambil SQL-SQL

Selain menjalankan query yang dibuat oleh Query Builder, kita juga dapat menarik statement SQL bersangkutan. Untuk melakukannya gunakan fungsi CDbCommand::getText().

```
$sql = Yii::app()->db->createCommand()
    ->select('*')
    ->from('tbl_user')
    ->text;
```

Jika terdapat parameter tertentu yang terikat pada query, mereka dapat diambil melalui properti CDbCommand::params.

Sintaks Alternatif untuk Membentuk Query

Kadangkala, menggunakan method chaining bukanlah pilihan yang tepat. Query Builder Yii memungkinkan kita untuk membuat query dengan menggunakan assignment property object yang sederhana. Pada umumnya, untuk setiap method query builder, terdapat sebuah property yang memiliki nama yang sama. Meng-assign nilai ke property sama saja dengan memanggil method tersebut. Misalnya, berikut merupakan dua statement yang ekuivalen, dengan asumsi `$command` adalah objek CDbCommand:

```
$command->select(array('id', 'username'));
$command->select = array('id', 'username');
```

Selain itu, method CDbConnection::createCommand() dapat menerima array sebagai parameter. Pasangan nama-nilai di array akan digunakan untuk inisialisasi property instance CDbCommand yang dibuat. Ini artinya, kita dapat menggunakan code berikut untuk membuat sebuah query:

```
$row = Yii::app()->db->createCommand(array(
    'select' => array('id', 'username'),
    'from' => 'tbl_user',
    'where' => 'id=:id',
    'params' => array(':id'=>1),
))->queryRow();
```

Membangun Beberapa Query

Sebuah instance CDbCommand dapat dipakai ulang beberapa kali untuk membuat beberapa query. Namun, sebelum membuat query baru, harus memanggil method CDbCommand::reset() terlebih dahulu untuk menghapus query sebelumnya. Misalnya:

```
$command = Yii::app()->db->createCommand();
$users = $command->select('*')->from('tbl_users')->queryAll();
$command->reset();  // clean up the previous query
$posts = $command->select('*')->from('tbl_posts')->queryAll();
```

3. Membuat Query Manipulasi Data

Query manipulasi data adalah statement SQL untuk melakukan insert, update dan delete data dalam table database. Query builder menyediakan `insert`, `update` dan `delete` untuk tiap query tersebut. Tidak seperti method query SELECT yang dijelaskan di atas, setiap method query manipulasi data ini akan membuat sebuah statement SQL lengkap dan langsung menjalankannya.

- insert(): menyisipkan sebaris ke table
- update(): melakukan update data pada sebuah table
- delete(): menghapus data dari table

Di bawah ini kami akan memaparkan method-method query manipulasi data

insert()

```
function insert($table, $columns)
```

Method insert() membuat dan menjalankan statement SQL `INSERT`. Parameter `$table` menentukan table yang mana yang disisipkan, sedangkan `$columns` merupakan sebuah array dengan pasangan nama-nilai yang menjelaskan nilai-nilai kolom yang akan disisipkan. Method tersebut akan memberikan quote pada nama table dan akan menggunakan parameter-binding untuk nilai yang dimasukkan.

Berikut merupakan contohnya:

```
// buat dan jalankan SQL berikut :
// INSERT INTO `tbl_user` (`name`, `email`) VALUES (:name, :email)
$command->insert('tbl_user', array(
    'name'=>'Tester',
    'email'=>'tester@example.com',
));
```

update()

```
function update($table, $columns, $conditions='', $params=array())
```

Method update() akan membuat dan mengeksekusi statement `UPDATE` SQL. Parameter `$table` menentukan table mana yang akan di-update; `$columns` adalah sebuah array dengan pasangan nama-nilai yang menentukan nilai kolom yang akan di-update; `$conditions` dan `$params` mirip dengan where(), yang akan menetapkan klausa `WHERE` dalam statement `UPDATE`. Method ini akan memberikan quote pada nama dan menggunakan parameter-binding untuk nilai yang di-update.

Berikut merupakan contohnya:

```
// buat dan jalankan SQL berikut:
// UPDATE `tbl_user` SET `name`=:name WHERE id=:id
$command->update('tbl_user', array(
    'name'=>'Tester',
), 'id=:id', array(':id'=>1));
```

delete()

```
function delete($table, $conditions='', $params=array())
```

Method delete() membuat dan menjalankan statement SQL `DELETE`. Parameter `$table` menentukan table yang mana yang akan dihapus; `$conditions` dan `$params` mirip dengan where(), yakni menentukan `WHERE` di dalam statement `DELETE`. Method ini akan memberikan quote pada nama.

Berikut salah satu contoh:

```
//buat dan eksekusi SQL berikut:
// DELETE FROM `tbl_user` WHERE id=:id
$command->delete('tbl_user', 'id=:id', array(':id'=>1));
```

4. Membuat Query Manipulasi Schema

Selain query manipulasi dan penarikan normal, query builder juga menyediakan sekumpulan method yang digunakan untuk membuat dan menjalankan query SQL untuk manipulasi schema pada database. Query builder mendukung query-query berikut:

- createTable(): membuat table
- renameTable(): mengubah nama table
- dropTable(): drop (menghapus) table
- truncateTable(): mengosongkan table
- addColumn(): menambahkan sebuah kolom table
- renameColumn(): mengubah nama kolom table
- alterColumn(): mengubah sebuah kolom table
- dropColumn(): me-drop (hapus) kolom table
- createIndex(): membuat index
- dropIndex(): me-drop (hapus) index

Info: Walaupun statement SQL yang digunakan untuk manipulasi database schema sangat berbeda di antara DBMS, query builder mencoba menyediakan sebuah interface yang seragam untuk membuat query-query ini. Ini akan memudahkan proses migrasi database dari satu DBMS ke yang lainnya.

Tipe Data Abstrak

Query builder memperkenalkan sekumpulan tipe data abstrak yang dapat digunakan untuk mendefinisikan kolom table. Tidak seperti tipe data physical yang spesifik pada DBMS tertentu dan cukup berbeda di DBMS lainnya, tipe data abstrak bebas dari DBMS. Ketika tipe data abstrak digunakan untuk mendefinisikan kolom table, query builder akan mengubahnya menjadi tipe data physical bersangkutan.

Berikut tipe data abstrak yang didukung oleh query builder.

- `pk`: sebuah jenis primary key generik, akan diubah menjadi `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY` pada MySQL;
- `string`: jenis string, akan diubah menjadi `varchar(255)` pada MySQL;
- `text`: jenis teks (string panjang), akan diubah menjadi `text` pada MySQL
- `integer`: jenis integer, akan diubah menjadi `int(11)` pada MySQL;
- `float`: tipe angka floating, akan diubah menjadi `float` pada MySQL;
- `decimal`: tipe angka desimal, akan diubah menjadi `decimal` pada MySQL;
- `datetime`: tipe waktu tanggal, akan diubah menjadi `datetime` pada MySQL;
- `timestamp`: tipe timestamp, akan diubah menjadi `timestamp` pada MySQL;
- `time`: tipe waktu, akan diubah menjadi `time` pada MySQL;
- `date`: tipe tanggal, akan diubah menjadi `date` pada MySQL;
- `binary`: tipe data biner, akan diubah menjadi `blob` pada MySQL;
- `boolean`: tipe boolean, akan diubah menjadi `tinyint(1)` pada MySQL;

- `money`: tipe mata uang, akan diubah menjadi `decimal(19,4)` pada MySQL. Tipe ini sudah ada semenjak versi 1.1.8.

createTable()

```
function createTable($table, $columns, $options=null)
```

Method createTable() akan membuat dan menjalankan statement SQL untuk menghasilkan sebuah table. Parameter `$table` akan menentukan nama dari table yang dibuat. Parameter `$columns` menentukan kolom-kolom pada table baru. Kolom-kolom ini harus diberikan dalam bentuk pasangan nama-definisi (misalnya `'username'=>'string'`). Parameter `$options` menentukan pecahan SQL ekstra yang harus ditambahkan pada SQL yang dihasilkan. Query builder akan memberikan quote pada nama table dan nama kolom.

Ketika menentukan definisi sebuah kolom, kita dapat menggunakan tipe data abstrak seperti yang sudah dijelaskan di atas. Query builder akan mengubah tipe data abstrak tersebut menjadi tipe data physical bersangkuta, sesuai dengan DBMS yang digunakan. Misalnya, `string` akan diubah menjadi `varchar(255)` pada MySQL.

Sebuah definisi kolom juga bisa mengandung tipe data non-abstrak atau spesifikasi. Definisi kolom ini akan dmasukkan juga ke dalam SQL tanpa perubahan. Misalnya `point` bukanlah tipe data abstrak, dan jika digunakan di definisi kolom, maka akan keluar demikian pada SQL; dan `string NOT NULL` akan diubah menjadi `varchar(255) NOT NULL` (hanya tipe abstrak `string` yang diubah).

Berikut contoh bagaimana membuat sebuah table:

```
// CREATE TABLE `tbl_user` (
//     `id` int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
//     `username` varchar(255) NOT NULL,
//     `location` point
// ) ENGINE=InnoDB
createTable('tbl_user', array(
    'id' => 'pk',
    'username' => 'string NOT NULL',
    'location' => 'point',
), 'ENGINE=InnoDB')
```

renameTable()

```
function renameTable($table, $newName)
```

Method renameTable() membuat dan menjalankan statement SQL untuk mengubah nama table. Parameter `$table` menentukan nama dari table yang akan di-rename. Parameter `$newName` menentukan nama baru dari table. Query builder akan memberikan quote pada nama table.

Berikut contoh bagaimana melakukan perubahan nama pada table :

```
// RENAME TABLE `tbl_users` TO `tbl_user`
renameTable('tbl_users', 'tbl_user')
```

dropTable()

```
function dropTable($table)
```

Method dropTable() membuat dan menjalankan statement SQL untuk menghapus sebuah table. Parameter `$table` menentukan nama table yang akan di-drop (hapus). Query builder akan memberikan quote pada nama table.

Berikut contoh bagaimana menghapus sebuah table:

```
// DROP TABLE `tbl_user`
dropTable('tbl_user')
```

truncateTable

```
function truncateTable($table)
```

Method truncateTable() membuat dan menjalankan statement SQL untuk menghapus sebuah table. Parameter `$table` menentukan nama table yang ingin dikosongkan. Query builder akan memberikan quote pada nama table.

Berikut contoh bagaimana mengosongkan table:

```
// TRUNCATE TABLE `tbl_user`
truncateTable('tbl_user')
```

addColumn()

```
function addColumn($table, $column, $type)
```

Method addColumn() membuat dan menjalankan statement SQL untuk menambah kolom table baru. Parameter `$table` menentukan nama dari table yang akan ditambahkan kolom baru. Parameter `$column` akan menentukan nama dari kolom baru. Dan `$type` menentukan definisi kolom baru. Definisi kolom dapat mengandung tipe data abstrak, seperti yang dijelaskan pada sub-bagian "createTable" sebelumnya. Query builder akan memberikan quote pada nama table termasuk nama kolom.

Berikut contoh menambah sebuah kolom table:

```
// ALTER TABLE `tbl_user` ADD `email` varchar(255) NOT NULL
addColumn('tbl_user', 'email', 'string NOT NULL')
```

dropColumn()

```
function dropColumn($table, $column)
```

Method dropColumn() membuat dan menjalankan statement SQL untuk menghapus kolom table. Parameter `$table` menentukan nama dari table yang kolomnnya akan dihapus. Parameter `$column` menentukan nama dari kolom yang akan dihapus. Query builder akan memberikan quote pada nama table termasuk nama kolom.

Berikut contoh bagaimana menghapus sebuah kolom table:

```
// ALTER TABLE `tbl_user` DROP COLUMN `location`
dropColumn('tbl_user', 'location')
```

renameColumn()

```
function renameColumn($table, $name, $newName)
```

Method renameColumn() membuat dan mengeksekusi statement SQL untuk mengubah nama kolom table. Parameter `$table` menentukan nama table yang nama kolomnya akan diubah. Parameter `$name` menentukan nama kolom yang lama. Dan `$newName` menentukan nama kolom baru. Query builder akan memberikan quote pada nama table termasuk pada nama kolom.

Berikut contoh mengubah nama kolom table:

```
// ALTER TABLE `tbl_users` CHANGE `name` `username` varchar(255) NOT
NULL
renameColumn('tbl_user', 'name', 'username')
```

alterColumn()

```
function alterColumn($table, $column, $type)
```

Method alterColumn() membuat dan menjalankan statement SQL untuk mengubah kolom table. Parameter `$table` menentukan nama table yang kolomnya akan diubah. Parameter `$column` menentukan nama kolom yang akan diubah. Dan `$type` menentukan definisi baru pada kolom. Definisi kolom dapat mengandung tipe data abstrak, seperti yang sudah dijelaskan pada sub-bagian "createTable". Query builder akan memberikan quote pada nama table dan nama kolom.

Berikut contoh mengubah kolom table:

```
// ALTER TABLE `tbl_user` CHANGE `username` `username` varchar(255)
NOT NULL
alterColumn('tbl_user', 'username', 'string NOT NULL')
```

addForeignKey()

```
function addForeignKey($name, $table, $columns,
    $refTable, $refColumns, $delete=null, $update=null)
```

Method addForeignKey() akan membuat dan menjalankan statement SQL untuk menambahkan foreign key constraint pada sebuah table. Parameter `$name` menentukan nama foreign key. Parameter `$table` dan `$columns` menentukan nama table dan nama kolom yang akan ditetapkan sebagai foreign key. JIka lebih dari satu kolom, maka bisa dipisahkan dengan menggunakan koma. Parameter `$refTable` dan `$refColumns` menentukan nama dan kolom table yang akan menjadi reference foreign key. Parameter `$delete` dan `$update` menentukan opsi `ON DELETE` dan `ON UPDATE` dalam statement SQL. Kebanyakan DBMS mendukung opsi ini:`RESTRICT`, `CASCADE`, `NO ACTION`, `SET DEFAULT`, `SET NULL`. Query builder akan memberikan quote pada nama table, nama index dan nama kolom.

Berikut contoh menambah foreign key constraint,

```
// ALTER TABLE `tbl_profile` ADD CONSTRAINT `fk_profile_user_id`
// FOREIGN KEY (`user_id`) REFERENCES `tbl_user` (`id`)
// ON DELETE CASCADE ON UPDATE CASCADE
addForeignKey('fk_profile_user_id', 'tbl_profile', 'user_id',
    'tbl_user', 'id', 'CASCADE', 'CASCADE')
```

dropForeignKey()

```
function dropForeignKey($name, $table)
```

Method dropForeignKey() membuat dan menjalankan statement SQL untuk menghapus sebuah foreign key constraint. Parameter `$name` menentukan nama dari foreign key constraint yang akan dihapus. Parameter `$table` menentukan nama table yang foreign key constraintnya akan dihapus. Query builder akan memberikan quote pada nama table dan juga nama constraint.

Berikut contoh menghapus foreign key constraint:

```
// ALTER TABLE `tbl_profile` DROP FOREIGN KEY `fk_profile_user_id`
dropForeignKey('fk_profile_user_id', 'tbl_profile')
```

createIndex()

```
function createIndex($name, $table, $column, $unique=false)
```

Method createIndex() membuat dan menjalankan statement SQL untuk membuat sebuah index. Parameter `$name` menentukan nama index yang akan dibuat. Parameter `$table` menentukan nama table yang index-nya berada. Parameter `$column` menentukan nama kolom yang akan di-indeks. Dan parameter `$unique` menentukan apakah harus membuat unique index. Jika index terdiri dari beberapa kolom, maka harus dipisah dengan koma. Query builder akan memberikan quote pada nama table, nama index dan nama kolom.

Berikut contoh pembuatan index:

```
// CREATE INDEX `idx_username` ON `tbl_user` (`username`)
createIndex('idx_username', 'tbl_user')
```

dropIndex()

```
function dropIndex($name, $table)
```

Method dropIndex() membuat dan mengeksekusi statement SQL untuk menghapus indeks. Parameter `$name` menentukan nama dari yang index-nya akan dihapus. Parameter `$table` menentukan nama table yang index-nya akan dihapus. Query builder akan memberikan quote pada nama table sekaligus nama index.

Below is an example showing how to drop an index:

```
// DROP INDEX `idx_username` ON `tbl_user`
dropIndex('idx_username', 'tbl_user')
```

Active Record

Meskipun Yii DAO secara virtual dapat menangani setiap tugas terkait database, kemungkinan kita akan menghabiskan 90% waktu kita dalam penulisan beberapa pernyataan SQL yang melakukan operasi umum CRUD (create, read, update dan delete) tetap ada. Pemeliharaan kode kita saat dicampur dengan pernyataan SQL juga akan menambah kesulitan. Untuk memecahkan masalah ini, kita dapat menggunakan Active Record.

Active Record (AR) adalah teknik populer Pemetaan Relasional Objek / Object-Relational Mapping (ORM). Setiap kelas AR mewakili tabel database (atau view) yang atributnya diwakili oleh properti kelas AR, dan turunan AR mewakili baris pada tabel tersebut. Operasi umum CRUD diimplementasikan sebagai metode AR. Hasilnya, kita dapat mengakses data dengan cara lebih terorientasi-objek. Sebagai contoh, kita dapat menggunakan kode berikut untuk menyisipkan baris baru ke dalam tabel Post:

```
$post=new Post;
$post->title='sample post';
$post->content='post body content';
$post->save();
```

Selanjutnya kita mendalami bagaimana menyiapkan AR dan menggunakannya untuk melakukan operasi CRUD. Kita akan melihat bagaimana untuk menggunakan AR untuk bekerja dengan relasi database dalam seksi berikutnya. Demi kemudahan, kami menggunakan tabel database berikut sebagai contoh kita dalam bagian ini. Harap dicatat bahwa jika Anda menggunakan database MySQL, Anda harus mengganti AUTOINCREMENT dengan AUTO_INCREMENT dalam SQL berikut.

```
CREATE TABLE tbl_post (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
```

```
    title VARCHAR(128) NOT NULL,
    content TEXT NOT NULL,
    create_time INTEGER NOT NULL
);
```

Catatan: AR tidak ditujukan untuk memecahkan semua tugas-tugas terkait-database. AR paling cocok dipakai untuk memodelkan tabel database dalam konstruksi PHP dan melakukan query yang tidak melibatkan SQL yang kompleks. Yii DAO baru dipakai untuk skenario kompleks tersebut.

1. Membuat Koneksi DB

AR bergantung pada koneksi DB untuk melaksanakan operasi terkait-DB. Secara default, AR menganggap bahwa komponen aplikasi `db` adalah turunan CDbConnection yang dibutuhkan untuk bertindak sebagai koneksi DB. Konfigurasi aplikasi berikut memperlihatkan sebuah contoh:

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'system.db.CDbConnection',
            'connectionString'=>'sqlite:path/to/dbfile',
            // hidupkan cache schema untuk meningkatkan kinerja
            // 'schemaCachingDuration'=>3600,
        ),
    ),
);
```

Tip: Karena Active Record bergantung pada metadata mengenai tabel untuk menentukan informasi kolom, dibutuhkan waktu untuk membaca metadata dan menganalisanya. Jika skema database Anda sepertinya kurang perlu diubah, Anda bisa menghidupkan cache skema (schema caching) dengan mengkonfigurasi properti CDbConnection::schemaCachingDuration ke nilai lebih besar daripada 0.

Dukungan terhadap AR dibatasi oleh DBMS. Saat ini, hanya DBMS berikut yang didukung:

- MySQL 4.1 atau lebih tinggi
- PostgreSQL 7.3 atau lebih tinggi
- SQLite 2 dan 3
- Microsoft SQL Server 2000 atau lebih tinggi
- Oracle

Jika Anda ingin menggunakan komponen aplikasi selain `db`, atau jika Anda ingin bekerja dengan multipel database menggunakan AR, Anda harus meng-override CActiveRecord::getDbConnection(). Kelas CActiveRecord adalah basis kelas untuk semua kelas AR.

Tip: Ada dua cara untuk bekerja dengan multipel database dalam AR. Jika skema database berbeda, Anda dapat membuat basis kelas AR yang berbeda dengan implementasi berbeda pada getDbConnection(). Sebaliknya, secara dinamis mengubah variabel static CActiveRecord::db merupakan ide yang lebih baik.

2. Mendefinisikan Kelas AR

Untuk mengakses tabel database, pertama kita perlu mendefinisikan kelas AR dengan menurun CActiveRecord. Setiap kelas AR mewakili satu tabel database, dan instance AR mewakili sebuah record (baris) dalam tabel tersebut. Contoh berikut memperlihatkan kode minimal yang diperlukan untuk kelas AR yang mewakili tabel `Post`.

```
class Post extends CActiveRecord
{
    public static function model($className=__CLASS__)
    {
        return parent::model($className);
    }

    public function tableName()
    {
        return 'tbl_post';
    }
}
```

Tip: Karena kelas AR sering dirujuk di banyak tempat, kita dapat mengimpor seluruh direktori yang berisi kelas AR, daripada menyertakannya satu demi satu. Sebagai contoh, jika semua file kelas AR kita di bawah `protected/models`, kita dapat mengkonfigurasi aplikasi sebagai berikut:

```
return array(
  'import'=>array(
      'application.models.*',
  ),
);
```

Secara default, nama kelas AR sama dengan nama tabel database. Meng-override metode tableName() jika berbeda. Metode model() dideklarasikan seperti itu untuk setiap kelas AR (akan dijelaskan kemudian).

Info: Untuk menggunakan fitur prefiks tabel, metode tableName() untuk kelas AR harus di-override demikian,

```
public function tableName()
{
    return '{{post}}';
}
```

Sebagai ganti dari mengembalikan nama tabel secara lengkap, kita mengembalikan nama tabel tanpa prefiks dan membungkusnya dengan kurung kurawal ganda.

Nilai kolom pada baris tabel dapat diakses sebagai properti turunan kelas AR terkait. Sebagai contoh, kode berikut menyetel kolom title (atribut):

```
$post=new Post;
$post->title='a sample post';
```

Meskipun kita tidak pernah mendeklarasikan properti title secara eksplisit dalam kelas `Post`, kita masih dapat mengaksesnya dalam kode di atas. Ini disebabkan `title` adalah kolom dalam tabel `Post`, dan CActiveRecord membuatnya bisa diakses seperti layaknya properti dengan bantuan metode magic PHP `__get()`. Exception (exception) akan ditampilkan jika kita mencoba mengakses kolom yang tidak ada, dengan cara yang sama.

Info: Di dalam panduan ini, kita menggunakan huruf kecil untuk seluruh nama tabel dan kolom. Ini dikarenakan berbeda-bedanya cara DBMS dalam penanganan case-sensitive. Contohnya PostgreSQL tidak memperlakukan nama kolom case-sensitive (jadi huruf besar sama dengan huruf kecil) secara default, dan kita harus memberi tanda kutip pada kolom dalam query jika namanya memang mengandung campuran huruf besar dan kecil. Menggunakan huruf kecil saja akan menyelesaikan permasalahan ini.

AR bergantung pada pendefinisian primary key tabel yang baik. Jika sebuah tabel tidak memiliki primary key, maka AR membutuhkan kelas AR menentukan kolom mana yang dijadikan primary key dengan meng-overide fungsi primaryKey() sebagai berikut,

```
public function primaryKey()
{
    return 'id';
    // Kalau composite primary key, mak kembalikan nilai array
seperti demikian
    // return array('pk1', 'pk2');
```

```
}
```

3. Membuat Record

Untuk melakukan insert baris baru ke dalam tabel database, kita membuat instance baru dari kelas AR terkait, menyetel propertinya yang berkaitan dengan kolom tabel, dan memanggil metode save() untuk menyelesaikan proses insert.

```
$post=new Post;
$post->title='sample post';
$post->content='content for the sample post';
$post->createTime=time();
$post->save();
```

Jika primary key table bersifat auto-increment, setelah insert instance AR maka akan berisi primary key yang baru. Dalam contoh di atas, properti `id` akan merujuk pada nilai primary key tulisan yang baru saja disisipkan, meskipun kita tidak pernah mengubahnya secara eksplisit.

Jika kolom didefinisikan dengan beberapa nilai standar statis (misalnya string, angka) dalam skema tabel, properti terkait dalam instance AR akan secara otomatis memiliki nilai tersebut setelah instance dibuat. Satu cara untuk mengubah nilai default ini adalah dengan secara eksplisit mendeklarasikan properti dalam kelas AR:

```
class Post extends CActiveRecord
{
    public $title='please enter a title';
    ......
}

$post=new Post;
echo $post->title;  // ini akan menampilkan: please enter a title
```

Atribut dapat menempatkan nilai tipe CDbExpression sebelum record disimpan (baik skenario insert ataupun update) ke database. Sebagai contoh, untuk menyimpan timestamp yang dihasilkan oleh fungsi MySQL NOW(), kita dapat menggunakan kode berikut:

```
$post=new Post;
$post->createTime=new CDbExpression('NOW()');
// $post->createTime='NOW()'; tidak akan bekerja karena
// 'NOW()' akan dianggap sebagai string
$post->save();
```

Tip: Karena AR mengijinkan kita untuk melakukan operasi database tanpa menulis sejumlah pernyataan SQL, seringkali kita ingin mengetahui pernyataan SQL apa yang dijalankan oleh AR. Ini bisa dilakukan dengan menghidupkan fitur pencatatan

pada Yii. Sebagai contoh, kita dapat menghidupkan CWebLogRoute dalam konfigurasi aplikasi, dan kita akan melihat pernyataan SQL yang dijalankan untuk ditampilkan di akhir setiap halaman Web. Kita dapat menyetel CDbConnection::enableParamLogging menjadi true dalam konfigurasi aplikasi agar nilai parameter yang diikat ke pernyataan SQL juga dicatat.

4. Membaca Record

Untuk membaca data dalam tabel database, kita memanggil salah satu metode `find` seperti berikut.

```
// cari baris pertama sesuai dengan kondisi yang ditetapkan
$post=Post::model()->find($condition,$params);
// cari baris dengan primary key yang ditetapkan
$post=Post::model()->findByPk($postID,$condition,$params);
// cari baris dengan nilai atribut yang ditetapkan
$post=Post::model()->findByAttributes($attributes,$condition,
$params);
// cari baris pertama menggunakan pernyataan SQL yang ditetapkan
$post=Post::model()->findBySql($sql,$params);
```

Dalam contoh di atas, kita memanggil metode find dengan `Post::model()`. Ingat bahwa metode statis `model()` diperlukan oleh setiap kelas AR. Metode ini menghasilkan instance AR yang dipakai untuk mengakses metode tingkat kelas (mirip dengan metode kelas static) dalam konteks obyek.

Jika metode `find` menemukan baris yang sesuai dengan kondisi query, ia akan mengembalikan turunan `Post` yang propertinya berisi nilai kolom terkait dari baris table. Kemudian kita dapat membaca nilai yang diambil seperti yang kita lakukan pada properti obyek, sebagai contoh, `echo $post->title;`.

Metode `find` akan menghasilkan null jika tidak ada yang ditemukan dalam database dengan kondisi query yang diberikan.

Ketika memanggil `find`, kita menggunakan `$condition` dan `$params` untuk menetapkan kondisi query. Di sini, `$condition` dapat berupa string yang mewakili klausul `WHERE` dalam pernyataan SQL, dan `$params` adalah array parameter yang nilainya harus diikat ke tempat di dalam `$condition`. Sebagai contoh,

```
// cari baris dengan postID=10
$post=Post::model()->find('postID=:postID', array(':postID'=>10));
```

Catatan: Dalam contoh di atas, kita mungkin perlu meng-escape referensi pada kolom postID untuk DBMS tertentu. Sebagai contoh, jika kita menggunakan

PostgreSQL, kita harus menulis kondisi sebagai "postID"=:postID, karena PostgreSQL standarnya akan memperlakukan nama kolom tidak case-sensitive.

Kita juga bisa menggunakan `$condition` untuk menetapkan kondisi query lebih kompleks Sebagai ganti dari mengisi sebuah string, kita dapat mengatur `$condition` menjadi instance CDbCriteria yang mengijinkan kita untuk menetapkan kondisi selain klausul WHERE. Sebagai contoh,

```
$criteria=new CDbCriteria;
$criteria->select='title';  // hanya memilih kolom 'title'
$criteria->condition='postID=:postID';
$criteria->params=array(':postID'=>10);
$post=Post::model()->find($criteria); // $params tidak diperlukan
```

Catatan, saat menggunakan CDbCriteria sebagai kondisi query, parameter $params tidak lagi diperlukan karena ia bisa ditetapkan dalam CDbCriteria, seperti dicontohkan di atas.

Cara alternatif terhadap CDbCriteria adalah dengan mengoper array ke metode find. Kunci dan nilai array masing-masing berhubungan dengan properti kriteria nama dan nilai. Contoh di atas dapat ditulis ulang seperti berikut

```
$post=Post::model()->find(array(
    'select'=>'title',
    'condition'=>'postID=:postID',
    'params'=>array(':postID'=>10),
));
```

Info: Saat kondisi query menemukan beberapa kolom dengan nilai yang ditetapkan, kita dapat menggunakan findByAttributes(). Kita biarkan parameter $attributes menjadi array nilai yang diindeks oleh nama kolom. Dalam beberapa framework, tugas ini bisa dilaksanakan dengan memanggil metode seperti findByNameAndTitle. Meskipun pendekatan ini terlihat atraktif, sering menyebabkan kebingungan, konflik dan masalah seperti sensitifitas-jenis huruf pada nama-nama kolom.

Ketika lebih dari satu baris data memenuhi kondisi query yang ditetapkan, kita dapat mengambilnya sekaligus menggunakan metode `findAll`, masing-masing memiliki pasangan metode `find`, seperti yang sudah kami jelaskan.

```
// cari seluruh baris yang sesuai dengan kondisi yang ditetapkan
$posts=Post::model()->findAll($condition,$params);
// cari seluruh baris dengan primary key yang ditetapkan
$posts=Post::model()->findAllByPk($postIDs,$condition,$params);
// cari seluruh baris dengan nilai atribut yang ditetapkan
$posts=Post::model()->findAllByAttributes($attributes,$condition,
$params);
```

```
// cari seluruh baris dengan pernyataan SQL yang ditetapkan
$posts=Post::model()->findAllBySql($sql,$params);
```

Jika tidak ada yang sama dengan kondisi query, `findAll` akan mengembalikan array kosong. Ini berbeda dengan find yang akan mengembalikan null jika tidak menemukan apapun.

Selain metode find dan findAll seperti dijelaskan di atas, metode berikut juga disediakan demi kenyamanan:

```
// ambil sejumlah baris yang sesuai dengan kondisi yang ditetapkan
$n=Post::model()->count($condition,$params);
// ambil sejumlah baris menggunakan pernyataan SQL yang ditetapkan
$n=Post::model()->countBySql($sql,$params);
// periksa apakah ada satu baris yang sesuai denga kondisi yang
ditetapkan
$exists=Post::model()->exists($condition,$params);
```

5. Mengupdate Record

Setelah instance AR diisi dengan nilai kolom, kita dapat mengubah dan menyimpannya kembali ke tabel database.

```
$post=Post::model()->findByPk(10);
$post->title='new post title';
$post->save(); // simpan perubahan ke database
```

Seperti yang kita lihat, kita menggunakan metode save() yang sama untuk melakukan operasi insert maupun update. Jika instance AR dibuat menggunakan operator new, pemanggilan save() akan menyisipkan baris record baru ke dalam tabel database; jika turunan AR adalah hasil dari beberapa pemanggilan metode find atau findAll, memanggil save() akan mengupdate baris yang sudah ada dalam tabel. Bahkan, kita dapat menggunakan CActiveRecord::isNewRecord untuk mengetahui apakah turunan AR baru atau tidak.

Dimungkinkan juga untuk mengupdate satu atau beberapa baris dalam sebuah tabel database tanpa memanggilnya lebih dulu. AR menyediakan metode tingkat-kelas yang nyaman untuk tujuan ini:

```
// mutakhirkan baris yang sama seperti kondisi yang ditetapkan
Post::model()->updateAll($attributes,$condition,$params);
// mutakhirkan baris yang sama seperti kondisi dan primary key yang
ditetapkan
Post::model()->updateByPk($pk,$attributes,$condition,$params);
// mutakhirkan kolom counter dalam baris yang sesuai dengan kondisi
yang ditetapkan
Post::model()->updateCounters($counters,$condition,$params);
```

Dalam contoh di atas, $attributes adalah array nilai kolom yang diindeks oleh nama kolom; $counters adalah array nilai inkremental yang diindeks oleh nama kolom; sedangkan $condition dan $params seperti yang sudah dijelaskan dalam sub-bab sebelumnya.

6. Menghapus Record

Kita juga bisa menghapus baris data jika turunan AR sudah diisi dengan baris ini.

```
$post=Post::model()->findByPk(10); // menganggap ada tulisan yang
memiliki ID = 10
$post->delete(); // hapus baris dari tabel database
```

Catatan, setelah penghapusan, turunan AR tetap tidak berubah, tapi baris terkait dalam tabel database sudah tidak ada.

Metode tingkat kelas berikut disediakan untuk menghapus baris tanpa haris mengambilnya lebih dulu:

```
// hapus baris yang sesuai dengan kondisi yang ditetapkan
Post::model()->deleteAll($condition,$params);
// hapus baris yang sesuai dengan kondisi dan primary key yang
ditetapkan
Post::model()->deleteByPk($pk,$condition,$params);
```

7. Validasi Data

Ketika melakukan insert atau update baris, seringkali kita harus memeriksa apakah nilai kolom sesuai dengan aturan tertentu. Hal ini penting terutama jika nilai kolom diberikan oleh pengguna akhir. Pada dasarnya, kita seharusnya tidak boleh mempercayai apapun yang berasal dari sisi klien.

AR melakukan validasi data secara otomatis ketika save() sedang dipanggil. Validasi didasarkan pada aturan yang ditetapkan dalam metode rules() pada kelas AR. Untuk lebih jelasnya mengenai bagaimana untuk menetapkan aturan validasi, silahkan merujuk ke bagian Mendeklarasikan Aturan Validasi. Di bawah ini adalah alur kerja umum yang diperlukan oleh penyimpanan record:

```
if($post->save())
{
    // data benar dan insert/update dengan sukses
}
else
{
    // data tidak benar. panggil getErrors() untuk mengambil pesan
kesalahan
}
```

Ketika data untuk insert atau update dikirimkan oleh pengguna akhir dalam bentuk HTML, kita perlu menempatkannya ke properti AR terkait. Kita dapat melakukannya seperti berikut:

```
$post->title=$_POST['title'];
$post->content=$_POST['content'];
$post->save();
```

Jika terdapat banyak kolom, kita akan melihat daftar yang panjang dari penempatan tersebut. Ini dapat dipersingkat dengan pemakaian properti attributes seperti ditampilkan di bawah. Rincian dapat ditemukan dalam seksi Mengamankan Penempatan Atribut dan seksi Membuat Aksi.

```
// anggap $_POST['Post'] adalah array nilai kolom yang diindeks
oleh nama kolom
$post->attributes=$_POST['Post'];
$post->save();
```

8. Membandingkan Record

Seperti baris tabel, turunan AR secara unik diidentifikasi dengan nilai primary key. Oleh karena itu, untuk membandingkan dua instance AR, kita perlu membandingkan nilai primary key-nya, menganggap keduanya milik kelas AR yang sama. Cara paling sederhana adalah dengan memanggil CActiveRecord::equals().

Info: Tidak seperti implementasi AR dalam framework lain, Yii mendukung primary key composite dalam turunan AR-nya. Kunci primer terdiri dari dua atau lebih kolom. Secara bersamaan, nilai primary key disajikan sebagai array dalam Yii. Properti primaryKey memberikan nilai primary key pada turunan AR.

9. Kustomisasi

CActiveRecord menyediakan beberapa metode penampungan yang dapat di-overide dalam anak kelas untuk mengkustomisasi alur kerjanya.

- beforeValidate dan afterValidate: ini dipanggil sebelum dan sesudah validasi dilakukan.
- beforeSave dan afterSave: ini dipanggil sebelum dan sesudah penyimpanan instance AR.
- beforeDelete dan afterDelete: ini dipanggil sebelum dan sesudah instance AR dihapus.
- afterConstruct: ini dipanggil untuk setiap instance AR yang dibuat menggunakan operator `new`.

- beforeFind: ini dipanggil sebelum pencari AR dipakai untuk melakukan query (misal `find()`, `findAll()`).
- afterFind: ini dipanggil untuk setiap instance AR yang dibuat sebagai hasil dari query.

10. Menggunakan Transaksi dengan AR

Setiap instance AR berisi properti bernama dbConnection yang merupakan turunan dari CDbConnection. Kita dapat menggunakan fitur transaksi yang disediakan oleh Yii DAO jika diinginkan saat bekerja dengan AR:

```
$model=Post::model();
$transaction=$model->dbConnection->beginTransaction();
try
{
    // cari dan simpan adalah dua langkah yang bisa diintervensi
oleh permintaan lain
    // oleh karenanya kita menggunakan transaksi untuk memastikan
konsistensi dan integritas
    $post=$model->findByPk(10);
    $post->title='new post title';
    $post->save();
    $transaction->commit();
}
catch(Exception $e)
{
    $transaction->rollBack();
}
```

11. Named Scope

Catatan: Ide named scope berasal dari Ruby on Rails.

Sebuah *named scope* mewakili kriteria query *bernama* yang dapat digabung dengan named scope lain dan diterapkan ke query active record.

Named scope dideklarasikan terutama dalam metode CActiveRecord::scopes() sebagai pasangan nama-kriteria. Kode berikut mendeklarasikan tiga named scope, `published` dan `recently`, dalam kelas model `Post`

```
class Post extends CActiveRecord
{
    ......
    public function scopes()
    {
        return array(
            'published'=>array(
                'condition'=>'status=1',
            ),
            'recently'=>array(
                'order'=>'createTime DESC',
                'limit'=>5,
            ),
```

```
        );
    }
}
```

Setiap named scope dideklarasikan sebagai sebuah array yang dipakai untuk menginisialisasi instance CDbCriteria. Sebagai contoh, named scope `recently` menetapkan bahwa properti `order` adalah `createTime DESC` dan properti `limit` adalah 5 yang diterjemahkan ke dalam sebuah kriteria query yang harus menghasilkan paling banyak 5 tulisan terbaru.

Named scope dipakai sebagai pembeda pada pemanggilan metode `find`. Beberapa named scope dapat dikaitkan bersamaan dan menghasilkan set yang lebih terbatas. Sebagai contoh, untuk mencari tulisan terbaru yang diterbitkan, kita menggunakan kode berikut:

```
$posts=Post::model()->published()->recently()->findAll();
```

Secara umum, named scope harus berada di sebelah kiri pemanggilan metode `find`. Masing-masing menyediakan kriteria query, yang merupakan gabungan dengan kriteria lain, termasuk yang dioper ke pemanggilan metode `find`. Hal ini mirip dengan menambahkan sebuah daftar filter ke sebuah query.

Catatan: Named scope hanya bisa dipakai dengan metode tingkat-kelas. Yakni, metode harus dipanggil menggunakan `ClassName::model()`.

Parameterisasi Named Scope

Named scope dapat diparameterkan. Sebagai contoh, kita mungkin ingin mengkustomisasi sejumlah tulisan dengan named scope `recently`. Untuk melakukannya, daripada mendeklarasikan named scope dalam metode CActiveRecord::scopes, kita dapat mendefinisikan sebuah metode baru yang namanya sama seperti named scope tadi:

```
public function recently($limit=5)
{
    $this->getDbCriteria()->mergeWith(array(
        'order'=>'createTime DESC',
        'limit'=>$limit,
    ));
    return $this;
}
```

Selanjutnya, kita bisa menggunakan pernyataan berikut untuk mengambil 3 tulisan terbaru yang diterbitkan:

```
$posts=Post::model()->published()->recently(3)->findAll();
```

Jika kita tidak melengkapi parameter 3 di atas, secara standar kita akan mengambil 5 tulisan terbaru yang diterbitkan.

Default Scope

Kelas model dapat memiliki default scope yang akan diterapkan untuk semua query (termasuk yang relasional) berkenaan dengan model. Sebagai contoh, website yang mendukung multi bahasa mungkin hanya ingin menampilkan konten yang ditetapkan oleh pengguna saat ini. Karena di sana ada banyak query atas konten situs, kita dapat mendefinisikan default scope guna memecahkan masalah ini. Untuk melakukannya, kita meng-override metode CActiveRecord::defaultScope seperti berikut,

```
class Content extends CActiveRecord
{
    public function defaultScope()
    {
        return array(
            'condition'=>"language='".Yii::app()->language."'",
        );
    }
}
```

Sekarang, jika metode di bawah ini dipanggil, Yii akan secara otomatis menggunakan kriteria query seperti didefinisikan di atas:

```
$contents=Content::model()->findAll();
```

Note: Default scope dan named scope hanya berlaku pada query `SELECT`. Mereka akan mengabaikan query `INSERT`, `UPDATE`, dan `DELETE`. Juga, ketika mendeklarasikan scope (baik default maupun named), kelas AR tidak dapat digunakan untuk membuat query DB di dalam method yang mendeklarasikan scope.

Active Record Relasional

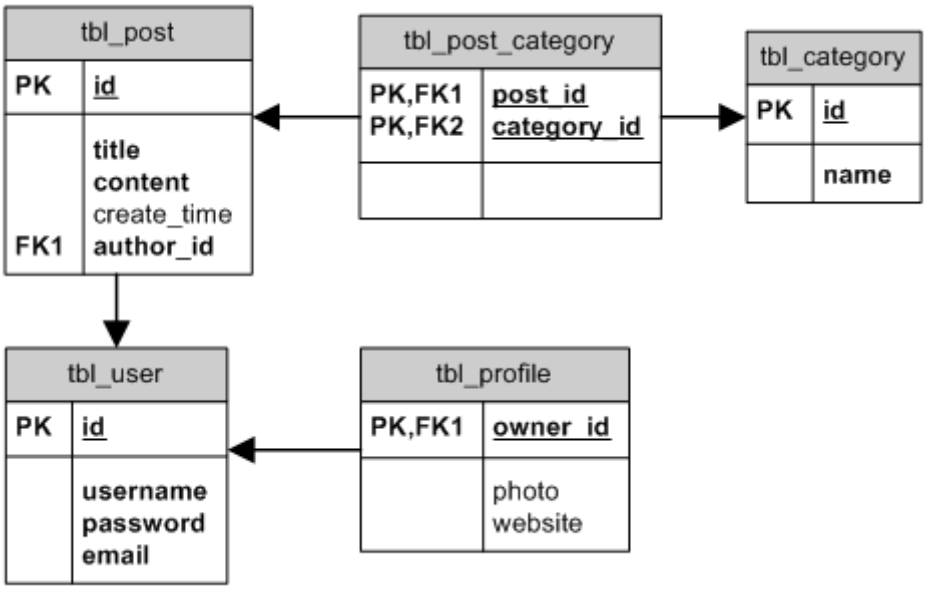
Kita sudah melihat bagaimana menggunakan Active Record (AR) untuk memilih data dari satu tabel database. Dalam bagian ini,kita akan melihat bagaimana

menggunakan AR untuk menggabung beberapa tabel database terkait dan mengembalikan hasil data yang telah digabungkan.

Untuk menggunakan AR relasional, disarankan untuk mendeklarasi constraint primary key-foreign key pada tabel yang ingin digabungkan(di-join). Contraint ini akan membantu menjaga konsistensi dan integritas data relasional.

Untuk menyederhanakan, kita akan menggunakan skema database yang ditampilkan dalam diagram entity-relationship (ER) atau hubungan-entitas berikut untuk memberi gambaran contoh pada bagian ini.

Diagram ER



Info: Dukungan untuk constraint foreign key bervariasi pada berbagai DBMS. SQLite < 3.6.19 tidak mendukung constraint foreign key, tetapi kita masih dapat mendeklarasikan constraint pada saat pembuatan tabel. Engine MySQL MyISAM tidak mendukung foreign key sama sekali.

1. Mendeklarasikan Hubungan

Sebelum kita menggunakan AR untuk melakukan query relasional, kita perlu membuat AR mengetahui bagaimana satu kelas AR dikaitkan dengan yang lain.

Hubungan antara dua kelas AR secara langsung dikaitkan dengan hubungan antara tabel-tabel database yang diwakili oleh kelas-kelas AR. Dari sudut pandang database, hubungan antar dua tabel A dan B memiliki tiga jenis: one-to-many/satu-ke-banyak (misal User dan Post), one-to-one/satu-ke-satu (misal User dan Profile) dan

many-to-many/banyak-ke-banyak (misal Category dan Post). Dalam AR, ada empat jenis hubungan:

- BELONGS_TO: jika hubungan antara tabel A dan B adalah satu-ke-banyak, maka B milik A (misal Post milik User);
- HAS_MANY: jika hubungan tabel A dan B adalah satu-ke-banyak, maka A memiliki banyak B (misal User memiliki banyak Post);
- HAS_ONE: ini kasus khusus pada HAS_MANY di mana A memiliki paling banyak satu B (misal User memiliki paling banyak satu Profile);
- MANY_MANY: ini berkaitan dengan hubungan banyak-ke-banyak dalam database. Tabel asosiatif diperlukan untuk memecah hubungan banyak-ke-banyak ke dalam hubungan satu-ke-banyak, karena umumnya DBMS tidak mendukung hubungan banyak-ke-banyak secara langsung. Dalam contoh skema database kita, tbl_post_category yang menjadi tabel asosiatif ini. Dalam terminologi AR, kita dapat menjelaskan MANY_MANY sebagai kombinasi BELONGS_TO dan HAS_MANY. Sebagai contoh, Post milik banyak Category dan Category memiliki banyak Post.

Mendeklarasikan hubungan dalam AR mencakup penimpaan metode relations() pada CActiveRecord. Metode tersebut mengembalikan array dari konfigurasi hubungan. Setiap elemen array mewakili satu hubungan dengan format berikut:

```
'VarName'=>array('RelationType', 'ClassName', 'ForeignKey', ...opsi tambahan)
```

dengan **VarName** sebagai nama hubungan/relasi; **RelationType** menetapkan jenis hubungan yang bisa berupa salah satu dari empat konstan: **self::BELONGS_TO**, **self::HAS_ONE**, **self::HAS_MANY dan self::MANY_MANY**; **ClassName** adalah nama kelas AR terkait dengan kelas AR ini; dan **ForeignKey** menetapkan kunci asing yang terkait dalam hubungan. Opsi tambahan dapat ditetapkan di akhir setiap relasi (dijelaskan nanti).

Kode berikut memperlihatkan bagaimana kita mendeklarasikan hubungan kelas User dan Post.

```
class Post extends CActiveRecord
{
    ......

    public function relations()
    {
        return array(
            'author'=>array(self::BELONGS_TO, 'User', 'author_id'),
            'categories'=>array(self::MANY_MANY, 'Category',
                'tbl_post_category(post_id, category_id)'),
        );
    }
}
```

```
class User extends CActiveRecord
{
    ......

    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id'),
            'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),
        );
    }
}
```

Info: Foreign key bisa saja berupa composite, yang artinya terdiri dari dua atau lebih kolom. Untuk hal ini, kita harus menggabungkan nama-nama kolom kunci dan memisahkannya dengan koma, atau dengan bentuk array seperti array('key1','key2'). Apabila Anda perlu menspesifikasikan asosiasi PK->FK sendiri, Anda bisa menulis array('fk'=>'pk').Untuk composite maka bentuknya akan berupa array('fk_c1'=>'pk_c1','fk_c2'=>'pk_c2'). Untuk jenis hubungan MANY_MANY, nama tabel asosiatif juga harus ditetapkan dalam foreign key. Contohnya, hubungan categories dalam Post ditetapkan dengan foreign key tbl_post_category(post_id, category_id). Deklarasi hubungan dalam kelas AR secara implisit menambahkan properti ke kelas untuk setiap hubungan. Setelah query relasional dilakukan, properti terkait akan diisi dengan instance dari AR yang dihubungkan. Sebagai contoh, jika $author mewakili turunan AR User, kita bisa menggunakan $author->posts untuk mengakses kaitannya dengan turunan Post.

2. Melakukan Query Relasional

Cara termudah melakukan query relasional adalah dengan membaca properti relasional turunan AR. Jika properti tidak diakses sebelumnya, query relasional akan diinisiasi, yang menggabungkan dua tabel terkait dan filter dengan primary key pada instance dari AR saat ini. Hasil query akan disimpan ke properti sebagai instance kelas AR terkait. Proses ini dikenal sebagai pendekatan *lazy loading*, contohnya, query relasional dilakukan hanya saat obyek terkait mulai diakses. Contoh di bawah memperlihatkan bagaimana menggunakan pendekatan ini:

```
// ambil tulisan di mana ID adalah 10
$post=Post::model()->findByPk(10);
// ambil penulis tulisan: query relasional akan dilakukan di sini
$author=$post->author;
```

Info: Jika tidak ada instance terkait pada hubungan, properti bersangkutan dapat berupa null atau array kosong. Untuk hubungan BELONGS_TO dan HAS_ONE, hasilnya

adalah null; untuk hubungan HAS_MANY dan MANY_MANY, hasilnya adalah array kosong. Catatan bahwa hubungan HAS_MANY dan MANY_MANY mengembalikan array obyek, Anda harus melakukan loop melalui hasilnya sebelum mencoba untuk mengakses setiap propertinya. Kalau tidak, Anda akan menerima pesan kesalahan "Trying to get property of non-object" ("Mencoba untuk mendapatkan properti non-obyek").

Pendekatan *lazy loading* sangat nyaman untuk dipakai, tetapi tidak efisien dalam beberapa skenario. Sebagai contoh, jika kita ingin mengakses informasi author (pengarang) untuk N post, menggunakan pendekatan lazy akan menyertakan eksekusi N query join. Kita harus beralih ke apa yang disebut pendekatan *eager loading* dalam situasi seperti ini.

Pendekatan *eager loading* mengambil instance AR terkait bersama dengan instance utama AR. Ini dilakukan dengan menggunakan metode with() bersama dengan salah satu metode find atau findAll dalam AR. Sebagai contoh,

```
$posts=Post::model()->with('author')->findAll();
```

Kode di atas akan mengembalikan sebuah array turunan Post. Tidak seperti pendekatan *lazy*, properti author dalam setiap instance Post sudah diisi dengan instance User terkait sebelum kita mengakses properti. Sebagai ganti dari menjalankan query join (gabungan) untuk setiap post, pendekatan eager loading membawa semua post bersama dengan author-nya ke dalam satu query join (gabungan)!

Kita dapat menetapkan nama multipel relasi dalam metode with() dan pendekatan *eager loading* akan mengembalikan semuanya dalam satu pekerjaan. Sebagai contoh, kode berikut akan mengembalikan post bersama dengan author dan category-nya (kategorinya):

```
$posts=Post::model()->with('author','categories')->findAll();
```

Kita juga bisa melakukan *nested eager loading*. Sebagai ganti dari mendaftar nama-nama relation, kita mengopernya dalam penyajian hirarki nama relasi ke method with(), seperti berikut,

```
$posts=Post::model()->with(
    'author.profile',
    'author.posts',
    'categories')->findAll();
```

Contoh di atas akan mengembalikan semua post bersama dengan author dan category-nya. Ini juga akan menghasilkan profil setiap author serta post.

Eager loading dapat dijalankan dengan menspesifikasi properti CDbCriteria::with, seperti di bawah ini:

```
$criteria=new CDbCriteria;
$criteria->with=array(
    'author.profile',
    'author.posts',
    'categories',
);
$posts=Post::model()->findAll($criteria);
```

atau

```
$posts=Post::model()->findAll(array(
    'with'=>array(
        'author.profile',
        'author.posts',
        'categories',
    )
));
```

3. Melakukan query relasional tanpa mengambil model bersangkutan

Terkadang kita perlu melakukan query dengan menggunakan relasi tetapi tidak ingin mengambil model bersangkutan. Misalnya, kita memiliki **`User`** yang memposting beberapa **`Post`**. Post dapat dipublikasi tetapi juga dapat dalam status draft. Status ini ditentukan oleh field **`published`** pada model post. Sekarang kita ingin mengambil semua user yang telah mempublikasikan post dan kita sendiri tidak tertarik pada post-post mereka. Maka, kita dapat mengambilnya dengan cara demikian:

```
$users=User::model()->with(array(
    'posts'=>array(
        //kita tidak ingin men-select post
        'select'=>false,
        // tetapi hanya ingin mengambil user yang telah publikasi
post
        'joinType'=>'INNER JOIN',
        'condition'=>'posts.published=1',
    ),
))->findAll();
```

4. Opsi Query Relasional

Telah kita sebutkan bahwa opsi tambahan dapat dispesifikasi dalam deklarasi relasi. Opsi ini ditetapkan sebagai pasangan name-value (nama-nilai), dipakai untuk mengkustomisasi query relasional. Rangkumannya adalah sebagai berikut.

- **`select`**: daftar kolom yang dipilih untuk kelas AR terkait. Standarnya adalah '*', yang artinya semua kolom. Nama-nama kolom yang direferensi dalam opsi ini tidak boleh ambigu.
- **`condition`**: klausul **`WHERE`**. Default-nya kosong. Nama kolom yang direferensikan di opsi ini juga tidak boleh ambigu.
- **`params`**: parameter yang diikat ke pernyataan SQL yang dibuat. Ini harus berupa array (larik) pasangan name-value.
- **`on`**: klausul **`ON`**. Kondisi yang ditetapkan di sini akan ditambahkan ke kondisi penggabungan menggunakan operator **`AND`**. Nama kolom direferensikan dalam opsi ini tidak boleh ambigu. Opsi ini tidak berlaku pada relasi **`MANY_MANY`**.
- **`order`**: klausul **`ORDER BY`**. Default-nya kosong. Nama kolom yang digunakan di sini tidak boleh ambigu.
- **`with`**: daftar dari child (turunan) objek terkait yang harus diambil bersama dengan objek ini. Harap berhati-hati, salah menggunakan opsi ini akan mengakibatkan pengulangan tanpa akhir.
- **`joinType`**: jenis gabungan untuk relasi ini. Standarnya **`LEFT OUTER JOIN`**.
- **`alias`**: alias untuk tabel terkait dengan hubungan ini. Default-nya null, yang berarti alias tabel sama dengan nama relasi.
- **`together`**: menentukan apakah tabel yang terkait dengan hubungan ini harus dipaksa untuk bergabung bersama dengan tabel primer dan tabel lainnya. Opsi ini hanya berarti untuk relasi HAS_MANY dan MANY_MANY. Jika opsi ini disetel false, setiap relasi yang berelasi HAS_MANY atau MANY_MANY akan digabungkan dengan tabel utama dalam query SQL yang terpisah, yang artinya dapat meningkatkan performa keseluruhan karena duplikasi pada data yang dihasilkan akan lebih sedikit. Jika opsi ini di-set true, maka tabel yang diasosiasi akan selalu di-join dengan tabel primer dalam satu query, walaupun jika tabel primer tersebut terpaginasi (paginated). Jika opsi ini tidak di-set, maka tabel yang terasosiasi akan di-join dengan tabel primer ke dalam query SQL tunggal hanya ketika tabel primer tidak terpaginasi. Untuk info selengkapnya, silahkan melihat bagian "Relational Query Performance".
- **`join`**: klausul **`JOIN`** ekstra. Secara default kosong. Opsi ini sudah ada semenjak versi 1.1.3
- **`group`**: klausul **`GROUP BY`**. Default-nya kosong. Nama kolom yang direferensi ke dalam opsi ini tidak boleh ambigu.
- **`having`**: klausul **`HAVING`**. Default-nya kosong. Nama kolom yang direferensi dalam opsi tidak boleh ambigu.
- **`index`**: nama kolom yang nilainya harus dipakai sebagai key (kunci) array yang menyimpan obyek terkait. Tanpa menyetel opsi ini, array obyek terkait akan menggunakan indeks integer berbasis-nol. Opsi ini hanya bisa disetel untuk relasi **`HAS_MANY`** dan **`MANY_MANY`**.
- **`scopes`**: nama scope yang ingin diaplikasikan. Jika ingin satu scope dapat digunakan seperti **`'scopes'=>'scopeName'`**. Dan jika lebih dari satu scope maka dapat digunakan seperti

'scopes'=>array('scopeName1','scopeName2'). Opsi ini sudah ada mulai dari versi 1.1.9.

Sebagai tambahan, opsi berikut tersedia untuk relasi tertentu selama lazy loading:

- limit: batas baris yang dipilih. Opsi ini TIDAK berlaku pada relasi BELONGS_TO.
- offset: offset baris yang dipilih. opsi ini TIDAK berlaku pada relasi BELONGS_TO.
- through: Nama dari relasi model yang digunakan sebagai penghubung ketika mengambil data relasi. Hanya bisa di-set untuk HAS_ONE dan HAS_MANY. Opsi ini sudah ada mulai dari versi 1.1.7.

Di bawah ini kita memodifikasi deklarasi relasi **posts** dalam **User** dengan menyertakan beberapa opsi di atas

```
class User extends CActiveRecord
{
    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                           'order'=>'posts.createTime DESC',
                           'with'=>'categories'),
            'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),
        );
    }
}
```

Sekarang jika kita mengakses **$author->posts**, kita akan mendapatkan post-post dari sang author yang tersusun berdasarkan waktu pembuatan secara terbalik (descending). Setiap instance post juga me-load category-nya.

5. Membedakan Nama Kolom

Ketika terdapat dua buah tabel atau lebih yang di-join ternyata memiliki sebuah kolom dengan nama yang sama, maka kita perlu membedakannya. Kita dapat melakukannya dengan menambah awalan pada nama kolom dengan nama alias tabel.

Dalam query AR relasional, nama alias untuk tabel utama ditetapkan sebagai **t**, sedangkan nama alias untuk sebuah tabel relasi adalah sama dengan nama relasi secara default. Misalnya, pada statement berikut, nama alias untuk **Post** adalah **t** dan untuk **Comment** adalah **comments**:

```
$posts=Post::model()->with('comments')->findAll();
```

Sekarang kita asumsi masing-masing Post dan Comment memiliki sebuah kolom bernama create_time yang menunjukkan waktu pembuatan post atau komentar, dan kita ingin mengambil nilai post beserta komentar-komentar dengan mengurutkan waktu pembuatan post baru kemudian waktu pembuatan komentar. Kita harus membedakan kolom create_time dengan kode berikut :

```
$posts=Post::model()->with('comments')->findAll(array(
    'order'=>'t.create_time, comments.create_time'
));
```

6. Opsi Query Relasional Dinamis

Kita dapat menggunakan opsi query relasional dinamis baik dalam with() maupun opsi with. Opsi dinamis akan menimpa opsi yang sudah ada seperti yang ditetapkan pada metode relations(). Sebagai contoh, dengan model User di atas, jika kita ingin menggunakan pendekatan eager loading untuk membawa kembali tulisan milik author (penulis) dalam *urutan membesar/ascending* (opsi order dalam spesifikasi relasi adalah urutan mengecil/descending), kita dapat melakukannya seperti berikut:

```
User::model()->with(array(
    'posts'=>array('order'=>'posts.create_time ASC'),
    'profile',
))->findAll();
```

Opsi query dinamis juga dapat dipakai saat menggunakan pendekatan lazy loading untuk melakukan query relasional. Untuk mengerjakannya, kita harus memanggil metode yang namanya sama dengan nama relasi dan mengoper opsi query dinamis sebagai parameter metode. Sebagai contoh, kode berikut mengembalikan tulisan pengguna yang memiliki status = 1:

```
$user=User::model()->findByPk(1);
$posts=$user->posts(array('condition'=>'status=1'));
```

7. Performa Query Relasi

Seperti yang dijelaskan di atas, pendekatan eager loading sering dipakai dalam skenario yang berkenan pada pengaksesan banyak objek yang berhubungan. Eager loading menghasilkan sebuah statement SQL kompleks dengan menggabungkan semua tabel yang diperlukan. Statement SQL yang besar lebih dipilih dalam beberapa kasus karena menyederhanakan pemfilteran berdasarkan kolom pada tabel yang ter-relasi. Namun, bisa saja untuk kasus-kasus tertentu cara tersebut tidak efisien.

Bayangkan sebuah contoh, kita perlu mencari post terbaru bersama dengan komentar-komentar post tersebut. Asumsi setiap post memiliki 10 komentar,

menggunakan sebuah statement SQL besar, akan mengembalikan data post yang redundan banyak sekali dikarenakan setiap post akan mengulangi setiap komentar yang dimilikinya. Mari kita menggunakan pendekatan lain: pertama kita melakukan query pertama untuk post terbaru, dan kemudian kueri komentarnya. Pada pendekatan ini, kita perlu mengeksekusi dua statement SQL. Manfaat dari pendekatan ini adalah tidak ada redudansi sama sekali dalam hasilnya.

Jadi pendekatan mana yang lebih efisien? Sebetulnya tidak ada jawaban pasti. Mengeksekusi sebuah statemen SQL besar mungkin lebih efisien karena mengurangi overhead pada DBMS untuk parsing dan executing pada statement SQL. Di lain pihak, menggunakan statement SQL tunggal, kemungkinan akan menghasilkan data redundansi sehingga perlu waktu lebih lama untuk baca dan memprosesnya.

Oleh karena itu, Yii menyediakan opsi query `together` sehingga kita dapat memilih di antara dua pendekatan ini jika diperlukan. Secara default, Yii menggunakan eager loading, seperti men-generate sebuah statement SQL tunggal, kecuali terdapat `LIMIT` di dalam model utama. Kita dapat mengeset opsi `together` di dalam deklarasi relasi menjadi true untuk memaksakan statement SQL walaupun `LIMIT` digunakan. Mengeset menjadi false, akan menyebabkan beberapa tabel akan di-join di statement SQL terpisah. Misalnya, untuk menggunakan statement SQL terpisah untuk mengquery post terbaru dengan komentar-komentarnya, kita dapat mendeklarasikan relasi `comments` di dalam kelas `Post` sebagai berikut

```
public function relations()
{
    return array(
        'comments' => array(self::HAS_MANY, 'Comment', 'post_id',
'together'=>false),
    );
}
```

Kita juga dapat mengeset opsi ini secara dinamis ketika melakukan eager loading:

```
$posts = Post::model()-
>with(array('comments'=>array('together'=>false)))->findAll();
```

8. Query Statistik

Selain query yang dijelaskan di atas, Yii juga mendukung apa yang disebut query statistik (atau query agregasional). Maksud dari query statistik adalah pengambilan informasi agregasional mengenai objek terkait, seperti jumlah komentar untuk setiap tulisan, rata-rata peringkat setiap produk, dll. Query statistik hanya bisa dilakukan untuk obyek terkait dalam `HAS_MANY` (misalnya sebuah tulisan memiliki banyak komentar) atau `MANY_MANY` (misalnya tulisan milik banyak kategori dan kategori memiliki banyak tulisan).

Melakukan query statistik sangat mirip dengan melakukan query relasional seperti dijelaskan sebelumnya. Pertama kita perlu mendeklarasikan query statistik dalam

metode relations() pada CActiveRecord seperti yang kita lakukan dengan query relasional.

```
class Post extends CActiveRecord
{
    public function relations()
    {
        return array(
            'commentCount'=>array(self::STAT, 'Comment', 'postID'),
            'categoryCount'=>array(self::STAT, 'Category',
'PostCategory(postID, categoryID)'),
        );
    }
}
```

Dalam contoh di atas, kita mendeklarasikan dua query statistik: `commentCount` menghitung jumlah komentar milik sebuah post, dan `categoryCount` menghitung jumlah kategori di mana post tersebut berada. Catatan bahwa hubungan antara `Post` dan `Comment` adalah `HAS_MANY`, sementara hubungan `Post` dan `Category` adalah `MANY_MANY` (dengan menggabung tabel `PostCategory`). Seperti yang bisa kita lihat, deklarasi sangat mirip dengan relasi yang kita lihat dalam subbagian sebelumnya. Perbedaannya jenis relasinya adalah `STAT` di sini.

Dengan deklarasi di atas, kita dapat mengambil sejumlah komentar untuk sebuah tulisan menggunakan ekspresi `$post->commentCount`. Ketika kita mengakses properti ini untuk pertama kalinya, pernyataan SQL akan dijalankan secara implisit untuk mengambil hasil terkait. Seperti yang sudah kita ketahui, ini disebut pendekatan *lazy loading*. Kita juga dapat menggunakan pendekatan *eager loading* jika kita harus menentukan jumlah komentar untuk multipel tulisan:

```
$posts=Post::model()->with('commentCount', 'categoryCount')-
>findAll();
```

Pernyataan di atas akan menjalankan tiga SQL untuk menghasilkan semua post bersama dengan jumlah komentar dan jumlah kategorinya. Menggunakan pendekatan lazy loading, kita akan berakhir dengan `2*N+1` query SQL jika ada `N` post.

Secara default, query statistik akan menghitung ekspresi `COUNT` (dan selanjutnya jumlah komentar dan jumlah kategori dalam contoh di atas). Kita dapat mengkustomisasinya dengan menetapkan opsi tambahan saat mendeklarasikannya dalam relations(). Opsi yang tersedia diringkas seperti berikut.

- `select`: ekspresi statistik. Standarnya `COUNT(*)`, yang berarti jumlah turunan objek.
- `defaultValue`: nilai yang diberikan ke record bersangkutan yang tidak menerima hasil query statistik. Sebagai contoh, jika ada sebuah post tidak memiliki komentar apapun, `commentCount` akan menerima nilai ini. Nilai standar untuk opsi ini adalah 0.
- `condition`: klausul `WHERE`. Default-nya kosong.

- `params`: parameter yang diikat ke pernyataan SQL yang dibuat. Ini harus berupa array pasangan nama-nilai.
- `order`: klausul `ORDER BY`. Default-nya kosong.
- `group`: klausul `GROUP BY`. Default-nya kosong.
- `having`: klausul `HAVING`. Default-nya kosong.

9. Query Relasional dengan Named Scope

Query relasional juga dapat dilakukan dengan kombinasi named scope. Named scope pada tabel relasional datang dengan dua bentuk. Dalam bentuk pertama, named scope diterapkan ke model utama. Dalam bentuk kedua, named scope diterapkan ke model terkait.

Kode berikut memperlihatkan bagaimana untuk menerapkan named scope ke model utama.

```
$posts=Post::model()->published()->recently()->with('comments')-
>findAll();
```

Ini sangat mirip dengan query non-relasional. Perbedaannya hanyalah bahwa kita memiliki panggilan `with()` setelah rantai named-scope. Query ini akan membawa kembali post terbaru yang dipublikasikan bersama dengan komentar-komentarnya.

Kode berikut memperlihatkan bagaimana untuk menerapkan named scope ke model bersangkutan.

```
$posts=Post::model()->with('comments:recently:approved')->findAll();
// atau kalau mulai dari 1.1.7
$posts=Post::model()->with(array(
    'comments'=>array(
        'scopes'=>array('recently','approved')
    ),
))->findAll();
// or semenjak 1.1.7
$posts=Post::model()->findAll(array(
    'with'=>array(
        'comments'=>array(
            'scopes'=>array('recently','approved')
        ),
    ),
));
```

Query di atas akan membawa kembali semua tulisan bersama dengan komentarnya yang sudah disetujui. Catatan bahwa `comments` merujuk ke nama relasi, sementara `recently` dan `approved` merujuk ke dua named scope yang dideklarasikan dalam kelas model `Comment`. Nama relasi dan named scope harus dipisahkan dengan titik dua.

Named scope dapat ditetapkan dalam opsi `with` pada aturan relasional yang dideklarasikan dalam CActiveRecord::relations(). Dalam contoh berikut, jika kita

mengakses `$user->posts`, maka akan mengembalikan semua komentar yang *disetujui* pada tulisan.

```
class User extends CActiveRecord
{
    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                'with'=>'comments:approved'),
        );
    }
}

// atau semenjak 1.1.7
class User extends CActiveRecord
{
    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                'with'=>array(
                    'comments'=>array(
                        'scopes'=>'approved'
                    ),
                ),
            ),
        );
    }
}
```

Catatan: Sebelum versi 1.1.7, named scope diaplikasikan ke model yang berelasi harus ditentukan di CActiveRecord::scopes. Oleh karenanya, tidak dapat diberi parameter.

Mulai dari versi 1.1.7, Yii memungkinkan passing parameter untuk relational named scopes. Misalnya, jika anda memiliki scope bernama `rated` di dalam `Post` yang menerima rating minimal post, Anda dapat menggunakannya dari `User` dengan cara berikut:

```
$users=User::model()->findAll(array(
    'with'=>array(
        'posts'=>array(
            'scopes'=>array(
                'rated'=>5,
            ),
        ),
    ),
));
```

10. Query Relasional dengan through

Ketika menggunakan `through`, definisi relasi harus dilihat seperti berikut:

```
'comments'=>array(self::HAS_MANY,'Comment',array('key1'=>'key2'),'th
rough'=>'posts'),
```

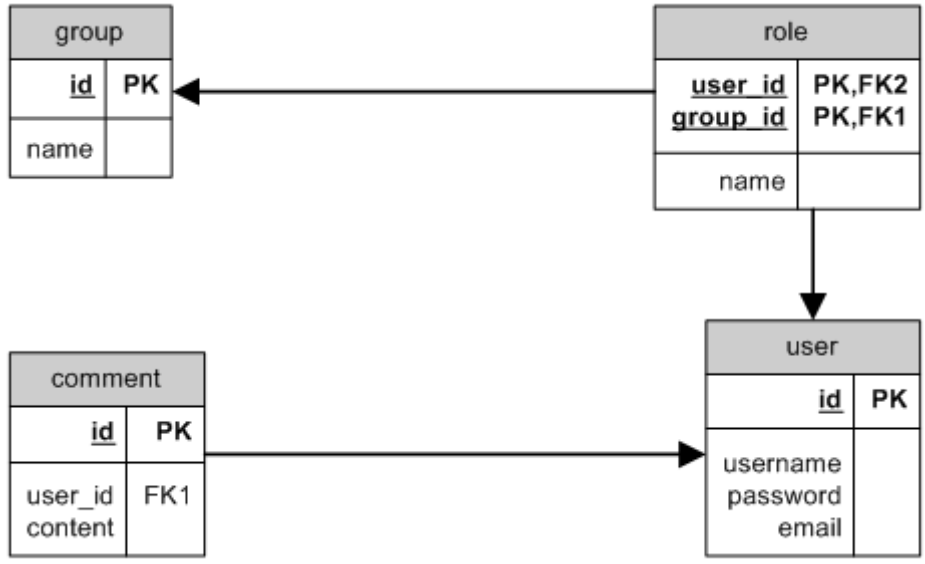
Di atas, nilai `foreign_key` adalah nama sebuah key yang:

- `key1` merupakan key yang didefinisikan di `throughRelationName`.
- `key2` merupakan key yang didefinisikan di `ClassName`.

`through` dapat digunakan oleh relasi `HAS_ONE` dan `HAS_MANY`.

HAS_MANY through

HAS_MANY through ER



Contoh `HAS_MANY` dengan `through` adalah mendapatkan user dari sebuah grup ketika user-user dimasukkan ke group tertentu melalui roles.

Contoh yang lebih kompleks misalnya mengambil semua komentar dari semua user dalam grup tertentu. Pada kasus ini kita harus menggunakan beberapa relasi melalui `through` dalam satu model:

```
class Group extends CActiveRecord
{
   ...
   public function relations()
   {
       return array(
           'roles'=>array(self::HAS_MANY,'Role','group_id'),
           'users'=>array(self::HAS_MANY,'User',array('user_id'=>'id
'),'through'=>'roles'),
           'comments'=>array(self::HAS_MANY,'Comment',array('id'=>'u
ser_id'),'through'=>'users'),
       );
```

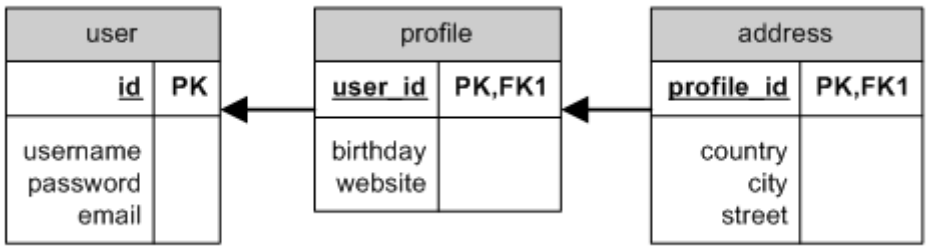
```
    }
}
```

Contoh Penggunaan

```
// mendapatkan semua grup dengan usernya
$groups=Group::model()->with('users')->findAll();

// mendapatkan semua group dengan usernya dan role-nya.
$groups=Group::model()->with('roles','users')->findAll();

// mendapatkan semua user dan role yang ID grup-nya 1
$group=Group::model()->findByPk(1);
$users=$group->users;
$roles=$group->roles;

// mendapatkan semua komentar yang grup ID-nya 1
$group=Group::model()->findByPk(1);
$comments=$group->comments;
```

HAS_ONE through

HAS_ONE through ER



Contoh penggunaan `HAS_ONE` dengan `through` adalah mendapatkan alamat user di mana user diikat dengan address menggunakan profile. Semua entiti ini (user, profile, address) masing-masing memiliki model:

```
class User extends CActiveRecord
{
    ...
    public function relations()
    {
        return array(
            'profile'=>array(self::HAS_ONE,'Profile','user_id'),
            'address'=>array(self::HAS_ONE,'Address',array('id'=>'pro
file_id'),'through'=>'profile'),
        );
    }
}
```

Contoh penggunaan

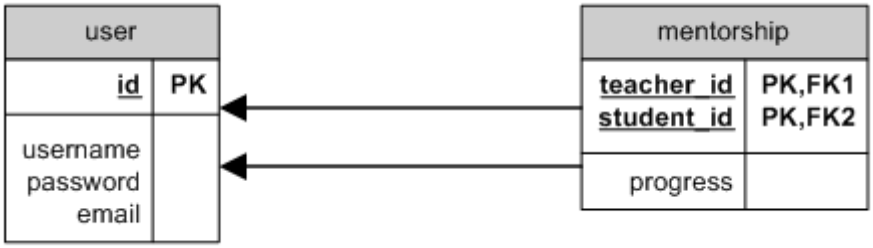
```
// mendapatkan alamat user yang ID-nya 1
```

```
$user=User::model()->findByPk(1);
$address=$user->address;
```

through pada diri sendiri

`through` dapat digunakan pada model yang diikat diri sendiri dengan menggunakan sebuah model penghubung. Pada kasus kita, seorang user merupakan mentor bagi user lain:

through self ER



Beginilah kita mendefinisikan relasi pada kasus ini:

```
class User extends CActiveRecord
{
    ...
    public function relations()
    {
        return array(
            'mentorships'=>array(self::HAS_MANY,'Mentorship','teacher
_id','joinType'=>'INNER JOIN'),
            'students'=>array(self::HAS_MANY,'User',array('student_id
'=>'id'),'through'=>'mentorships','joinType'=>'INNER JOIN'),
        );
    }
}
```

Contoh Penggunaan

```
// Dapatkan semua siswa yang diajarkan guru dengan ID 1
$teacher=User::model()->findByPk(1);
$students=$teacher->students;
```

Migrasi Database

1. Pembutan Migrasi
2. Transactional Migrations
3. Mengaplikasikan Migrasi
4. Me-Revert Migrations

5. Mengulangi Migrasi
6. Menampilkan Informasi Migrasi
7. Mengubah Histori Migrasi
8. Mengkustomisasi Perintah Migrasi

Catatan: Fitur migrasi database telah tersedia sejak versi 1.1.6

Sama seperti halnya dengan source code, struktur database selalu berkembang seiring kita mengembangkan dan merawat aplikasi database-driven. Misalnya, ketika saat pengembangan, kita ingin menambah sebuah tabel baru, atau setelah aplikasi sudah rampung, kita mungkin menyadari perlu menambah sebuah indeks pada sebuah kolom. Sangatlah penting untuk selalu menjaga track dari perubahan strutktural database ini (yang disebut dengan **migration (migrasi)** seperti halnya yang kita lakukan pada source code. Jika source code dan database tidak tersinkronisasi, besar peluangnya keseluruhan aplikasi akan rusak. Karena itulah, Yii menyediakan fungsi migrasi database yang bisa menjaga histori migrasi database, mengaplikasikan migrasi baru ataupun mengembalikannya.

Berikut merupakan langkah-langkah yang dilakukan untuk migrasi database pada masa pengembangan:

1. Budi membuat sebuah migrasi baru (misalnya membuat tabel baru)
2. Budi commit migrasi baru ke sistem pengaturan source (seperti SVN, GIT)
3. Joni mengupdate dari sistem pengontrolan source dan mendapatkan migrasi baru
4. Joni mengaplikasikan migrasi ke database pengembangan lokalnya

Yii mendukung migrasi database melalui perintah `yiic migrate`. Tool ini mendukung pembuatan migrasi baru, mengaplikasikan/mengubah/mengulangi migrasi, dan menampilkan histori migrasi dan migrasi baru.

Berikut ini, kita akan menjelaskan cara penggunaan tool ini.

Catatan: Lebih baik menggunakan yiic dari aplikasi (yakni `cd path/ke/protected`) ketika menjalankan command `migrate` daripada dari direktori `framework`. Pastikan Anda memiliki direktori `protected\migrations` dan writable(memiliki hak akses untuk tulis). Cek juga koneksi database di `protected/config/console.php`

1. Pembutan Migrasi

Untuk membuat sebuah migrasi baru (seperti membuat table news), kita jalankan perintah ini:

```
yiic migrate create <name>
```

Parameter wajib `name` digunakan sebagai deskripsi yang sangat singkat tentang migrasi (misalnya `create_news_table`). Seperti yang akan ditunjukkan berikut ini,

parameter `name` digunakan sebagai bagian dari nama kelas PHP. Oleh karena itu, seharusnya hanya mengandung huruf, angka atau garis bawah (underscore).

```
yiic migrate create create_news_table
```

Perintah di atas akan membuat sebuah file baru bernama `m101129_185401_create_news_table.php` di dalam `protected/migrations` yang mengandung code berikut ini:

```
class m101129_185401_create_news_table extends CDbMigration
{
    public function up()
    {
    }

public function down()
    {
        echo "m101129_185401_create_news_table tidak mendukung
migration down.\n";
        return false;
    }

    /*
    // implementasi safeUp/safeDown jika perlu transaction.
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

Perhatikan bahwa nama kelas sama dengan nama file dengan pola `m<timestamp>_<name>`, dengan `<timestamp>` merujuk pada timestamp UTC (dalam format `yymmdd_hhmmss`) ketika migrasi dibuat, dan `<name>` diambil dari parameter `name` dari command-nya.

Method `up()` harus mengandung code yang mengimplementasikan migrasi database aktual, sedangkan method `down()` bisa berisi code yang me-revert apa yang telah dilakukan `up()`.

Kadangkala, mengimplementasi `down()` adalah tidak mungkin. Misalnya, kita menghapus baris tabel di `up()`, kita tidak akan bisa mengembalikan mereka dengan `down()`. Dalam hal ini, migrasi disebut sebagai irreversible(tidak dapat dibalikkan), yang berarti kita tidak bisa roll back ke keadaan sebelumnya dalam database. Pada code yang di-generate di atas, method `down()` mengembalikan false untuk menandakan bahwa migration tersebut tidak dapat dibalikkan.

Info: Mulai dari versi 1.1.7, jika method `up()` atau `down()` mengembalikan `false`, maka semua migration selanjutnya akan dibatalkan. Sebelumnya pada versi 1.1.6, kita harus melempar exception untuk membatalkan migration selanjutnya

Contohnya, mari melihat tentang cara pembuatan sebuah tabel news.

```
class m101129_185401_create_news_table extends CDbMigration
{
    public function up()
    {
        $this->createTable('tbl_news', array(
            'id' => 'pk',
            'title' => 'string NOT NULL',
            'content' => 'text',
        ));
    }

    public function down()
    {
        $this->dropTable('tbl_news');
    }
}
```

Kelas dasar CDbMigration menyediakan sekumpulan method untuk manipulasi data dan skema database. Misalnya, CDbMigration::createTable akan membuat table database, sedangkan CDbMigration::insert akan menyisipkan sebuah baris data. Method-method ini semuanya menggunakan koneksi database yang dikembalikan oleh CDbMigration::getDbConnection(), yang secara default akan mengembalikan `Yii::app()->db`.

Catatan: Kamu mungkin menyadari bahwa method-method database yang disediakan oleh CDbMigration sangat mirip dengan method-method di CDbCommand. Memang mereka hampir sama kecuali method CDbMigration akan mengukur waktu yang digunakan oleh methodnya dan mencetak beberapa pesan tentang parameter method.

2. Transactional Migrations

Info: Fitur transactional migration didukung mulai versi 1.1.7.

Ketika melakukan migrasi DB yang kompleks, kita umumnya ingin memastikan apakah setiap migrasi cukup berhasil atau gagal sebagai kesatuan sehingga perawatan database bisa tetap konsisten dan integritas. Guna mencapai tujuan ini, kita dapat memanfaatkan DB transaction.

Kita dapat secara eksplisit memulai sebuah transaksi DB dan menyertakan sisa code yang berhubungan dengan DB, seperti berikut ini:

```
class m101129_185401_create_news_table extends CDbMigration
{
    public function up()
```

```
    {
        $transaction=$this->getDbConnection()->beginTransaction();
        try
        {
            $this->createTable('tbl_news', array(
                'id' => 'pk',
                'title' => 'string NOT NULL',
                'content' => 'text',
            ));
            $transaction->commit();
        }
        catch(Exception $e)
        {
            echo "Exception: ".$e->getMessage()."\n";
            $transaction->rollBack();
            return false;
        }
    }

    // ...similar code for down()
}
```

Namun, cara lebih gampang untuk mendapatkan dukungan transaction adalah dengan mengimplementasikan `safeUp()` daripada `up()`, dan `safeDown()` daripada `down()`. Misalnya,

```
class m101129_185401_create_news_table extends CDbMigration
{
    public function safeUp()
    {
        $this->createTable('tbl_news', array(
            'id' => 'pk',
            'title' => 'string NOT NULL',
            'content' => 'text',
        ));
    }

    public function safeDown()
    {
        $this->dropTable('tbl_news');
    }
}
```

Ketika Yii melakukan migrasi, maka Yii akan memulai DB transaction dan memanggil `safeUp()` atau `safeDown()`. Jika terdapat error DB yang terjadi di `safeUp()` atau `safeDown()`, maka transaksi akan di-rollback, sehingga memastikan database tetap dalam keadaan baik.

Catatan: Tidak semua DBMS mendukung transaction. Dan beberapa query DB tidak dapat dimasukkan ke transaction. Dalam hal ini, Anda tetap harus mengimplementasikan `up()` dan `down()`. Untuk MySQL, beberapa statement bisa menyebabkan implicit commit.

3. Mengaplikasikan Migrasi

Untuk mengaplikasikan migrasi baru yang tersedia (misalnya membuat database lokal up-to-date), jalankan perintah berikut:

```
yiic migrate
```

Perintah ini akan menampilkan daftar semua migrasi yang baru. Jika Anda konfirmasi untuk mengaplikasikan migrasi, maka akan dijalankan method `up()` di setiap kelas migrasi baru, satu per satu secara berurutan, sesuai dengan urutan nilai timstamp di dalam nama kelas.

Setelah mengaplikasikan migrasi, tool migrasi akan mencatat record di sebuah table database bernama `tbl_migration`. Dengan demikian memungkinkan tool untuk mengidentifikasikan berapa migrasi yang telah diaplikasikan dan yang mana belum. Jika table `tbl_migration` tidak ada, maka tool ini akan secara otomatis membuatkannya, tergantung pada database yang dispesifikasikan komponen aplikasi `db`.

Kadang-kadang, kita hanya ingin mengaplikasikan satu atau beberapa migrasi. Kita dapat menggunakan perintah berikut ini:

```
yiic migrate up 3
```

Perintah ini kan mengaplikasikan 3 migrasi baru. Dengan mengubah nilai 3 akan mengubah jumlah migrasi yang diaplikasikan.

Kita juga dapat migrasikan database ke versi tertentu melalui perintah berikut:

```
yiic migrate to 101129_185401
```

Kita menggunakan bagian timestamp pada nama migrasi untuk menentukan versi yang ingin kita migrasikan databasenya. Jika terdapat beberapa migrasi di antara migrasi terakhir yang di-apply dan migrasi khusus, semua migrasi ini akan diaplikasikan. Jika migrasi yang ditetapkan telah diaplikasikan sebelumnya, maka seluruh migrasi diaplikasikan setelah di-revert (kembalikan ke semula)

4. Me-Revert Migrations

Untuk me-revert migrasi yang paling terakhir atau beberapa migrasi yang sudah teraplikasikan, kita dapat menggunakan command berikut ini:

```
yiic migrate down [step]
```

dengan parameter opsional `step` menentukan berapa banyak migrasi yang ingin di-revert. Nilai default-nya adalah 1, yang artinya me-revert kembali ke migrasi yang sudah teraplikasi paling terakhir.

Seperti yang sudah disebutkan sebelumnya, tidak semua migrasi dapat di-revert. Mencoba me-revert migrasi seperti demikian akan menghasilkan error dan menghentikan seluruh proses revert.

5. Mengulangi Migrasi

Mengulangi migrasi maksudnya melakukan revert kemudian mengaplikasikan migrasi tertentu. Proses ini dapat dilakukan dengan melakukan perintah berikut:

```
yiic migrate redo [step]
```

dengan parameter opsional `step` menunjukkan berapa banyak migrasi yang ingin diulangi. Nilai default-nya 1, yang artinya mengulangi migrasi terakhir.

6. Menampilkan Informasi Migrasi

Selain mengaplikasikan dan me-revert migrasi, perangkat migrasi juga dapat digunakan untuk menampilkan histori migrasi dan migrasi baru yang akan diaplikasikan.

```
yiic migrate history [limit]
yiic migrate new [limit]
```

Parameter opsional `limit` menunjukkan angka migrasi yang akan ditampilkan. Jika `limit` tidak ditentukan, semua migrasi yang tersedia akan ditampilkan.

Perintah pertama yang menampilkan migrasi yang sudah diaplikasikan, sedangkan perintah kedua menampilkan migrasi yang belum diaplikasikan.

7. Mengubah Histori Migrasi

Kadang-kadang, mungkin kita ingin mengubah histori migrasi ke versi migrasi tertentu tanpa sebetulnya mengaplikasikan atau me-revert kembali migrasi yang bersangkutan. Biasanya sering terjadi selama pengembangan migrasi baru. Kita dapat menggunakan perintah berikut.

```
yiic migrate mark 101129_185401
```

Perintah ini sangat mirip dengan perintah `yiic migrate to` kecuali dia hanya memodifikasikan tabel histori migrasi ke versi yang ditentukan tanpa melakukan perubahan ataupun me-revert migrasi.

8. Mengkustomisasi Perintah Migrasi

Terdapat beberapa cara untuk mengkustomisasi perintah migration.

Menggunakan Opsi Command Line

Perintah migrasi datang dari empat opsi yang dpaat ditentukan di command line:

- `interactive`: boolean, menentukan apakah migrasi dilakukan dengan cara interaktif. Secara default bernilai true, yang artinya user akan ditanya terlebih dahulu ketika melakukan migrasi tertentu. Anda dapat set nilainya menjadi false sehingga migrasi dapat dilakukan di belakang layar.
- `migrationPath`: string, menentukan direktori yang menampung seluruh file kelas migrasi. Harus diisi dengan bentuk path alias, dan direktori bersangkutan harus ada. Jika tidak ditentukan, maka akan digunakan sub-direktori `migrations` di dalam base path aplikasi.
- `migrationTable`: string, menentukan nama tabel database yang digunakan untuk menyimpan informasi histori migrasi. Secara default bernilai `tbl_migration`. Struktur tabelnya yakni `version varchar(255) primary key, apply_time integer`.
- `connectionID`: string, menentukan ID pada komponen database aplikasi. Nilai default adala 'db'.
- `templateFile`: string, menentukan jalur ke file yang ditunjukkan sebagai template code untuk men-generate kelas migrasi. Harus diisi dalam format path alias (misalnya `application.migrations.template`). Jika tidak diisi, maka template internal yang akan digunakan. Di dalam template, token `{ClassName}` akan digantikan oleh nama kelas migrasi yang sesungguhnya.

Untuk menentukan opsi-opsi tersebut, jalankan perintah migrate dengan format ini

```
yiic migrate up --option1=value1 --option2=value2 ...
```

Misalnya kita ingin migrasi module forum yang file-file migrasinya terletak di direktori module `migrations`, kita dapat menggunakan perintah berikut :

```
yiic migrate up --migrationPath=ext.forum.migrations
```

Mengatur Perintah Secara Global

Opsi command line memungkinkan kita melakukan konfiguras secara on-the-fly, namun kadangkala kita mungkin ingin mengatur command untuk selamanya. Misalnya, kita ingin menggunakan tabel yang berbeda untuk menyimpan histori migrasi, atau kita ingin menggunakan sebuah template migrasi yang sudah kita desain. Kita dapat melakukannya dengan mengubah konfigurasi console aplikasi dengan begini,

```
return array(
    ......
    'commandMap'=>array(
        'migrate'=>array(
            'class'=>'system.cli.commands.MigrateCommand',
            'migrationPath'=>'application.migrations',
            'migrationTable'=>'tbl_migration',
```

```
            'connectionID'=>'db',
            'templateFile'=>'application.migrations.template',
        ),
        ......
    ),
    ......
);
```

Sekarang jika kita jalankan perintah `migrate`, konfigurasi di atas akan berefek tanpa kita harus menulis perintah opsi setiap saat lagi.