Advanced Object Oriented Database access using PDO

Marcus Börger

ApacheCon EU 2005

Intro

☑ PHP and Databases

☑ PHP 5 and PDO

PHP 4 and Databases

- ☑ PHP can connect to all important RDBMS

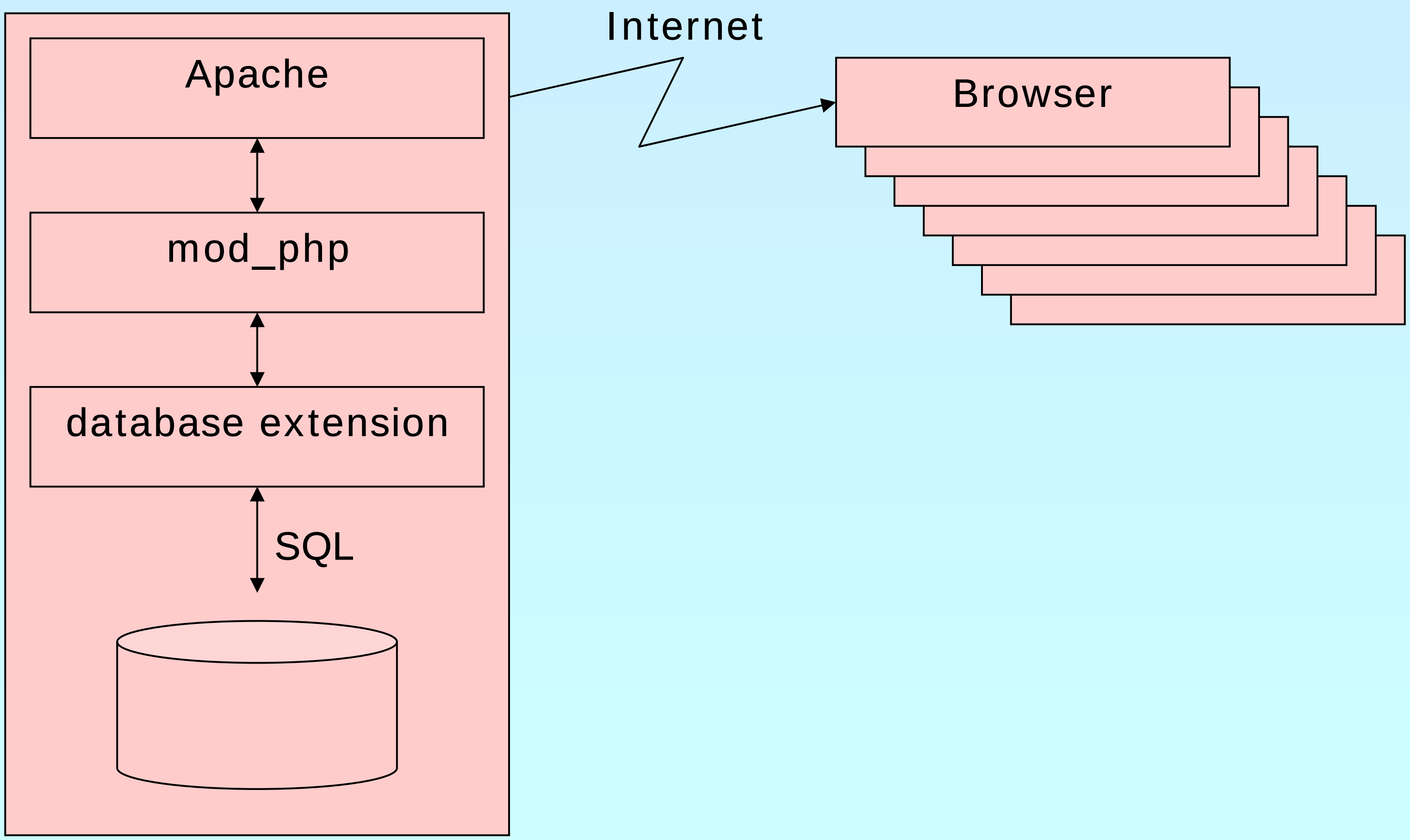
- ☒ Each RDBMS needs a separate extension
- ☒ Each extension has a different interface
  - ☒ ext/dbx is an inefficient abstraction
- ☒ Multiple PEAR solutions
  - ☑ Abstraction layers
  - ☑ Query builders
  - ☑ Data Access Objects . . . Nested Set support

  - ☒ But there is ‘no’ OO in PHP 4

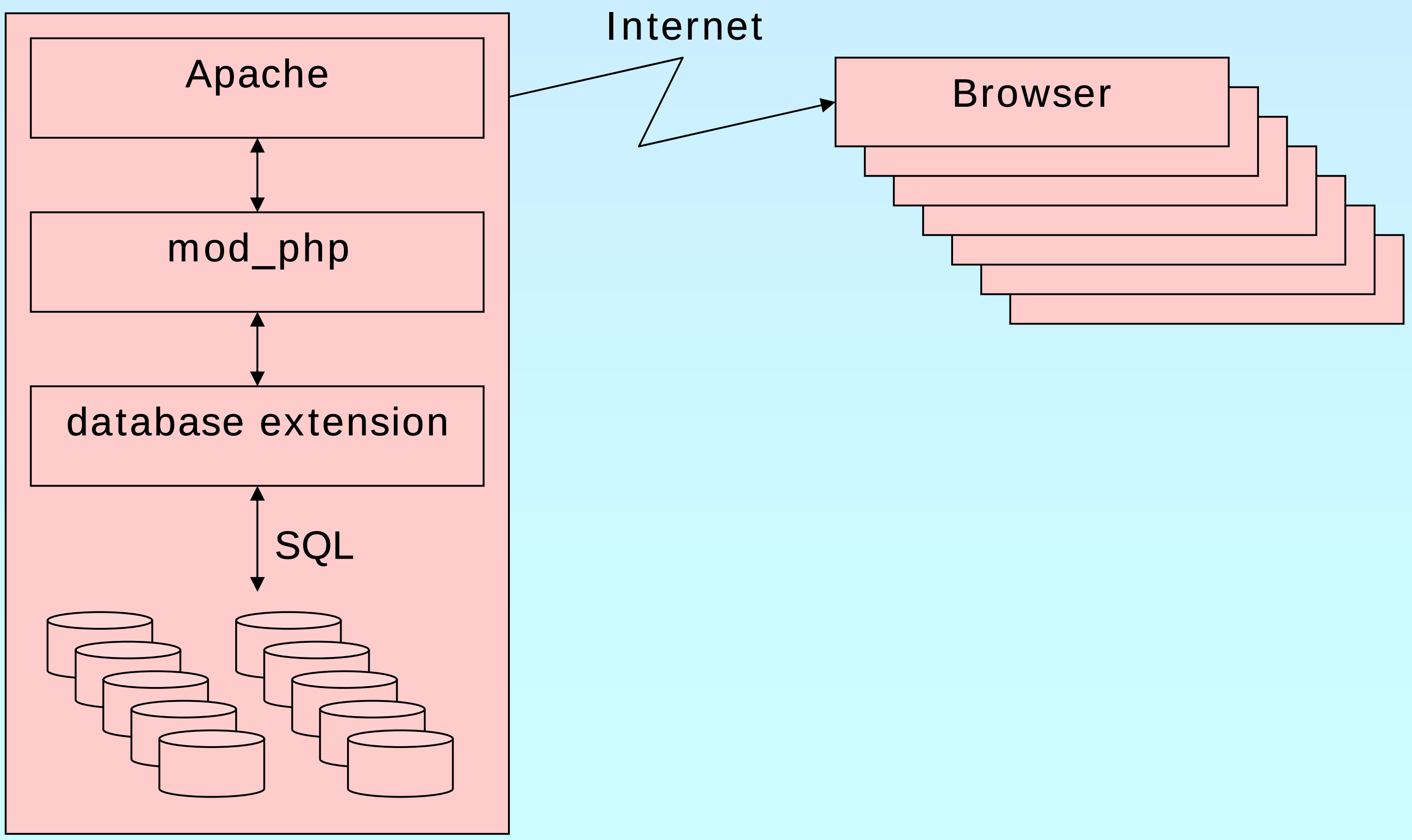
PHP 5 and Databases

- ☑ PHP can connect to all important RDBMS
- ☑ PDO provides a unified efficient abstraction
- ☑ PHP is ready for UML
- ☑ Specialized extensions allow detailed control
- ☑ Multiple PEAR solutions
  - ☑ More sophisticated abstraction layers
  - ☑ Query builders
  - ☑ Data Access Objects . . . Nested Set support
- ☑ Multiple ways of using databases with PHP
  - ☑ File based as ext/dba or ext/sqlite or embedded MySQL
  - ☑ Talking SQL with embedded RDBMS
  - ☑ Talking SQL with external RDBMS
  - ☑ Using ODBC

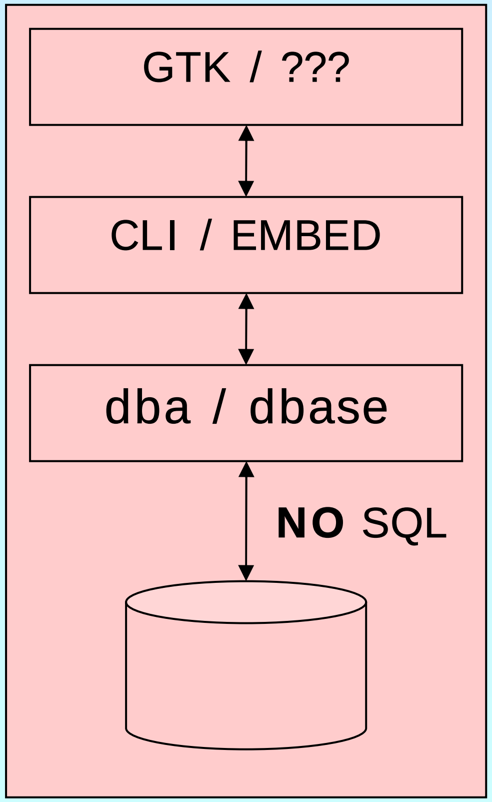
Dedicated Host



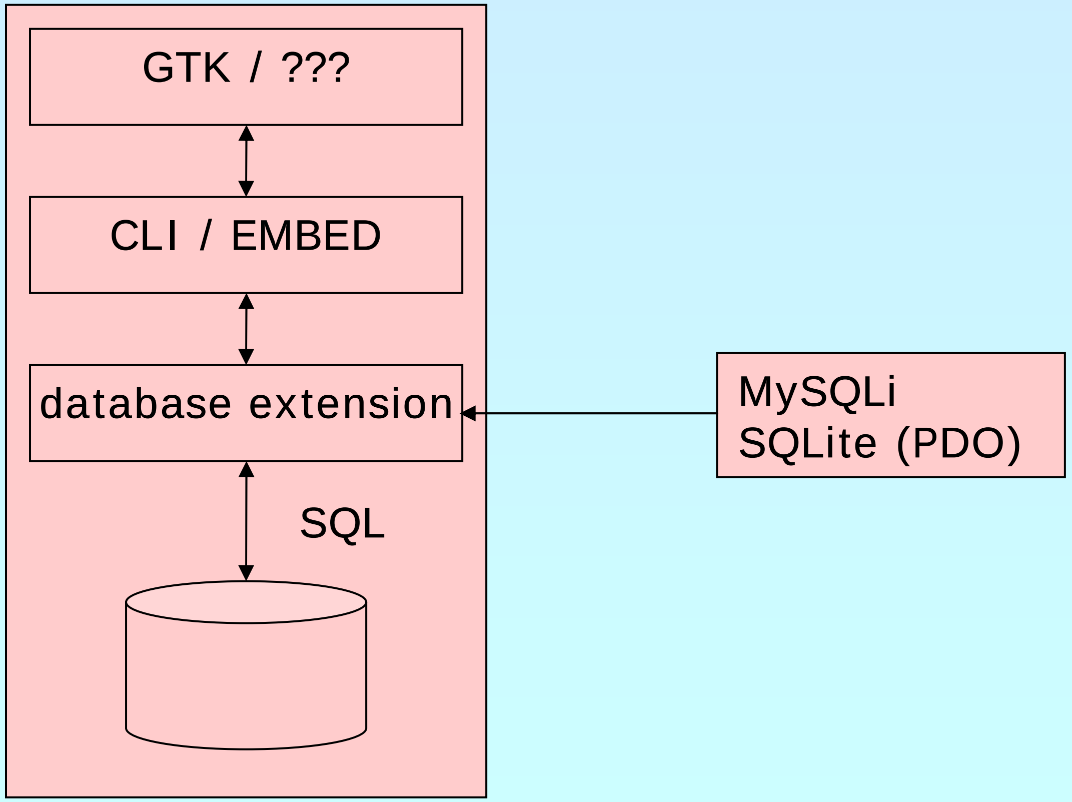
ISP/Shared Host



Embedded



Embedded



PHP and Databases

☑ PHP can connect to all important RDBMs

- ☑ Oracle
- ☑ PostgreSQL
- ☑ MySQL
- ☑ Interbase/Firebird
- ☑ ODBC
- ☑ SQLite
- ☑ MS-SQL
- ☑ mSQL

All talk some SQL dialect

and

All using different API

- ☑ DBM-style databases

- ☑ Support for native XML database available using XQL

APACHECON EU 2005

PHP and Databases

☑ PHP can connect to all important RDBMs

☑ Oracle	PDO & native (pc)
☑ PostgreSQL	PDO & native (pc)
☑ MySQL	PDO & native (pc/oo)
☑ Interbase/Firebird	PDO & native (pc)
☑ ODBC	PDO & native (pc)
☑ SQLite	PDO & native (pc/oo)
☑ MS-SQL	PDO & native (pc)
☑ mSQL	native (pc)

☑ DBM-style databases

☑ Support for native XML database available using XQL

PDO at a glance

- ☑ Data access abstraction (API unification)
- ☑ Multiple database plug-in extensions
- ☑ Object oriented
- ☑ Iterator support
- ☑ Destructive read support
- ☑ All written in a tiny c layer
- ☑ Will be used us base layer of upcoming MDB2
- ☑ Available through PECL
  - ☑ Buildable for PHP 5.0
  - ☑ Built-in starting from 5.1
  - ☑ Windows DLLs available
  - ☑ Already used in a few production servers
  - ☑ ATM still marked experimental

PDO at a glance

- ☑ Prepared statements (unified, name and index)
- ☑ SQL state error code
- ☑ Portability attributes
- ☑ Transaction supprt
- ☑ Scrollable cursors
- ☑ Uses normal PHP error facilities or Exceptions

Plans:

- ☑ LOB support

Connecting to the database

☑ PDO uses DSNs to connect

    <handler-name> ':' <native-DSN>

```
try {
   $dbh = new PDO($dsn, $user, $password, $options);
   //
   // Use the database
   //
   // and close it
   $dbh = NULL;
} catch (PDOException $e) {
   echo "Failed to connect:" . $e->getMessage();
}
```

PDO DSN format

- ☑ odbc:odbc_dsn
- ☑ mysql:host=*name*;dbname=*dbname*
- ☑ sqlite:/path/to/db/file
- ☑ sqlite::memory:
- ☑ sqlite2:/path/to/sqlite2/file
- ☑ pgsql:host=*localhost* port=*5432* dbname=*test*
- ☑ oci:dbname=*dbname*;charset=*charset*
- ☑ firebird:dbname=*db*;charset=*charset*;role=*role*

Direct SQL execution

- ☑ PDO::exec() allows to avoid PDOStatement object
  - ☑ Most usefull for DDL (i.e. CREATE) and INSETR, UPDATE

```
$dbh = new PDO($dsn);
$cnt = $dbh->exec($sql);
if ($cnt !== false) {
   echo "Rows affected: " . $cnt;
   echo "Last inserted id: " . $dbh->lastInsertId();
} else {
   echo "Error";
}
```

Fetching data with prepare

- ☑ The default fetch methodology is unbuffered
- ☑ Uses methods **prepare**() and execute()
- ☑ Forward only
- ☑ Row count unknown

```
$dbh = new PDO($dsn);
$stmt = $dbh->prepare("SELECT * FROM FOO");
$stmt->execute();
while ($row = $stmt->fetch()) {
    // use data in $row
}
$stmt = null;
```

Fetching data w/o prepare

- ☑ Uses method query()
- ☑ Forward only
- ☑ Row count unknown

```
$dbh = new PDO($dsn);
$stmt = $dbh->query("SELECT * FROM FOO");
$stmt->execute();
while ($row = $stmt->fetch()) {
    // use data in $row
}
$stmt = null;
```

Fetching data from iterator

- ☑ Faster data access
- ☑ Works with and without preparation
- ☑ Forward only
- ☑ Row count not available

```
$dbh = new PDO($dsn);
$stmt = $dbh->prepare("SELECT * FROM FOO");
$stmt->execute();
foreach ($stmt as $row) {
    // use data in $row
}
$stmt = null;

foreach($dbh->query("SELECT * FROM bar") as $row) {
   // use data in $row
}
```

Fetching data into array

- ☑ Data is fully buffered
- ☑ Works with and without preparation
- ☑ Randam access
- ☑ Row count available
- ☑ Usefull if database doesn't support parallel queries

```
$dbh = new PDO($dsn);
$stmt = $dbh->prepare("SELECT * FROM FOO");
$stmt->execute();
$data = $stmt->fetchAll();
foreach ($data as $row) {
    // use data in $row
}
$stmt = null;
```

How to retrieve data

☑ Fetch single dataset in default way

```
   mixed PDOStatement::fetch(
        int     $mode               = PDO_FETCH_BOTH,
        int     $orientation        = PDO_FETCH_ORI_NEXT,
        int     $offset             = 0)
```

also controlled by

```
   void PDOStatement::setFetchMode(
        int     $mode,              // PDO_FETCH_*
        [mixed*$params])            // mode specific params
```

☑ Fetch single column value

```
   mixed PDOStatement::fetchColumn(
        int     $column_number = 0)  // zero based index
```

How to retrieve data

☑ Fetch all rows at once

```
array PDOStatement::fetchAll(
     int    $mode            = PDO_FETCH_BOTH,
     string $class_name      = NULL,
     array  $ctor_args       = NULL)
```

☑ Fetch single row as object

```
mixed PDOStatement::fetchObject(
     string $class_name      = NULL,
     array  $ctor_args       = NULL)
```

Fetch modes and flags

☑ Modes

PDO_FETCH_ASSOC	associative array
PDO_FETCH_NUM	numeric array
PDO_FETCH_BOTH	**default** (assoc/numeric)
PDO_FETCH_OBJ	into `stdClass` object
PDO_FETCH_BOUND	into bound variables
PDO_FETCH_COLUMN	single column
PDO_FETCH_CLASS	into new instance
PDO_FETCH_INTO	into existing object
PDO_FETCH_FUNC	through function call

☑ Flags

PDO_FETCH_GROUP	group by first col
PDO_FETCH_UNIQUE	group unique by first col
PDO_FETCH_CLASSTYPE	use class name in row
PDO_FETCH_SERIALIZE	use serialization

PDO_FETCH_BOUND

- ☑ Fetching returns true until there is no more data
  - ☑ Binding parameters by "?" in sql (1 based index)
  - ☑ Binding parameters by ":name" in sql
  - ☑ Binding columns by name and index

```
$dbh = new PDO($dsn);
$stmt = $dbh->prepare(
   'SELECT url FROM urls WHERE key=:urlkey');
$stmt->bindParam(':urlkey', $urlkey);
$stmt->bindColumn('url', $href);

$urlkey = ...; // get url key to translate
$stmt->execute(); // execute the query

// fetch data
$stmt->fetch(PDO_FETCH_BOUND);
// use data
echo '<a href="' . $href . '">' . $urlkey . '</a>';
```

PDO_FETCH_BOUND

☑ Fetching returns true until there is no more data

- ☑ Binding parameters by "?" in sql 1 based index
- ☑ Binding parameters by ":name" in sql
- ☑ Binding columns by name and index
- ☑ Binding can be done on execute()

```
$dbh = new PDO($dsn);
$stmt = $dbh->prepare(
   'SELECT url FROM urls WHERE key=:urlkey');


$urlkey = ...; // get url key to translate
$stmt->execute(array(':urlkey' => $urlkey),
               array('url' => $href));
// fetch data
$stmt->fetch(PDO_FETCH_BOUND);
// use data
echo '<a href="' . $href . '">' . $urlkey . '</a>';
```

PDO_FETCH_CLASS

- ☑ Lets you specify the class to instantiate
  - ☑ PDO_FETCH_OBJ always uses stdClass
  - ☑ Writes data before calling __construct
    - ☑ Can write private/protected members
- ☑ Lets you call the constructor with parameters

```php
class Person {
   protected $dbh, $fname, $lname;
   function __construct($dbh) {
      $this->dbh = $dbh;
   }
   function __toString() {
      return $this->fname . " " . $this->lname;
   }
}
$stmt = $dbh->prepare('SELECT fname, lname FROM persons');
$stmt->setFetchMode(PDO_FETCH_CLASS, 'Person', array($dbh));
$stmt->execute();
foreach($stmt as $person) {
   echo $person;
}
```

PDO_FETCH_CLASSTYPE

- ☑ Lets you fetch the class to instantiate from rows
  - ☑ Must be used with PDO_FETCH_CLASS
  - ☑ The class name specified in fetch mode is a fallback

```
class Person { /* ... */ }
class Employee extends Person { /* ... */ }
class Manager extends Employee { /* ... */ }

$stmt = $dbh->prepare(
   'SELECT class, fname, lname FROM persons LEFT JOIN
      classes ON persons.kind = classes.id');
$stmt->setFetchMode(PDO_FETCH_CLASS|PDO_FETCH_CLASSTYPE,
   'Person', array($dbh));
$stmt->execute();
foreach($stmt as $person) {
   echo $person;
}
```

PDO_FETCH_INTO

- ☑ Lets you reuse an already instantiated object
- ☑ Does not allow to read into protected or private
  - ☑ Because the constructor was already executed

```
class Person {
   public $dbh, $fname, $lname;
   function __construct($dbh) {
      $this->dbh = $dbh;
   }
   function __toString() {
      return $this->fname . " " . $this->lname;
   }
}
$stmt = $dbh->prepare('SELECT fname, lname FROM persons');
$stmt->setFetchMode(PDO_FETCH_INTO, new Person($dbh));
$stmt->execute();
foreach($stmt as $person) {
   echo $person;
}
```

PDO_FETCH_FUNC

☑ Lets you specify a function to execute on each row

```
class Person {
   protected $fname, $lname;
   static function Factory($fname, $lname) {
      $obj = new Person;
      $obj->fname = $fname;
      $obj->lname = $lname;
   }
   function __toString() {
      return $this->fname . " " . $this->lname;
   }
}
$stmt = $dbh->prepare('SELECT fname, lname FROM persons');
$stmt->setFetchMode(PDO_FETCH_FUNC,
                          array('Person', 'Factory'));
$stmt->execute();
foreach($stmt as $person) {
   echo $person;
}
```

PDOStatement as real iterator

- ☑ PDOStatement only implements Traversable
- ☑ Wrapper IteratorIterator takes a Traverable

```
$it = new IteratorIterator($stmt);
```

- ☑ Now the fun begins
  - ☑ Just plug this into any other iterator
  - ☑ Recursion, SQL external unions, Filters, Limit, …

```
foreach(new LimitIterator($it, 10) as $data) {
   var_dump($data);
}
```

Deriving PDOStatement

☑ prepare() allows to specify fetch attributes

```
     PDOStatement PDO::prepare(
          string    $sql,
          array(PDO_ATTR_STATEMENT_CLASS =>
            array(string classname,
              array(mixed * ctor_args))));


class MyPDOStatement extends PDOStatement {
  protected $dbh;
  function __construct($dbh) {
    $this->dbh = $dbh;
  }
}
$dbh->prepare($sql,
          array(PDO_ATTR_STATEMENT_CLASS =>
               array('MyPDOStatement', array($dbh))));
```

Deriving PDOStatement

☑ Deriving allows to convert to real iterator

```
class PDOStatementAggregate extends PDOStatement
   implements IteratorAggregate
{
   private function __construct($dbh, $classtype) {
      $this->dbh = $dbh;
      $this->setFetchMode(PDO_FETCH_CLASS,
         $classtype, array($this));
   }
   function getIterator()   {
      $this->execute();
      return new IteratorIterator($this,
         'PDOStatement'); /* Need to be base class */
   }
}
$stmt = $dbh->prepare('SELECT * FROM Persons',
   array(PDO_ATTR_STATEMENT_CLASS =>
      array('PDOStatementAggregate',
         array($dbh, 'Person'))));
foreach($stmt as $person){
   echo $person;
}
```

PDO error modes

☑ PDO offers 3 different error modes

```
$dbh->setAttribute(PDO_ATTR_ERRMODE, $mode);
```

- ☑ PDO_ERRMODE_SILENT
  Simply ignore any errors
- ☑ PDO_ERRMODE_WARNING
  Issue errors as standard php warnings
- ☑ PDO_ERRMODE_EXCEPTION
  Throw exception on errors

- ☑ Map native codes to SQLSTATE standard codes
- ☑ Aditionally offers native info

Performance

10 times Querying 10 rows

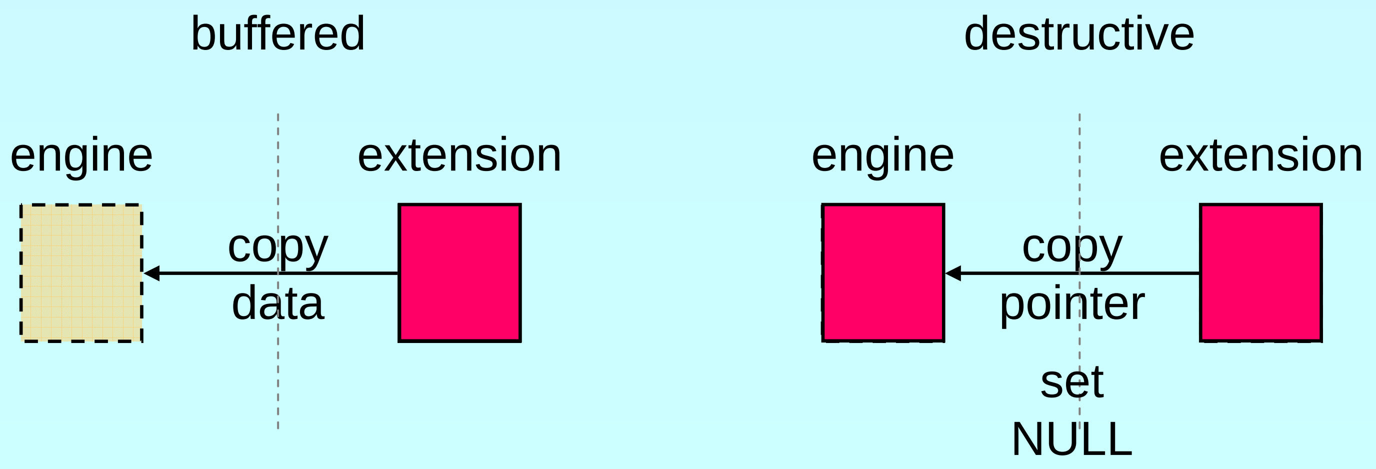
- ☑ Iterators vs. Arrays
  - ☑ Implemented as engine feature: 56%
  - ⌧ Building an Array is expensive
- ☑ queryArray vs. query and fetchArray: 89%
  - ⌧ Function calls are expensive

Performance

☑ Buffered vs. Unbuffered: up to 60%

- ☑ Buffered queries need to build a hash table
- ☑ Buffered queries must copy data
- ☑ Unbuffered queries can use **destructive reads**

⌧ Copying data is expensive



Performance

☑ Comparing OO vs. Procedural code

☑ PC is easy to program?	
☑ PC uses resources:	O(n*log(n))
☑ PC uses a single function table:	2000 ... 4000
☑ OO code is little bit more to learn	
☑ OO code is easy to maintain	
☑ OO code uses object storage:	O(n+c)
☑ OO uses small method tables:	10 ... 100

Performance?

Don't get overexcited

using PDO your RDBMS is your bottlneck

Links

☑ This presenatation
  http://talks.somabo.de

☑ Documenation on PDO
  http://docs.php.net/pdo

☑ The PDO Extension
  http://pecl.php.net/package/PDO

☑ The Windows DLLs
  http://snaps.php.net