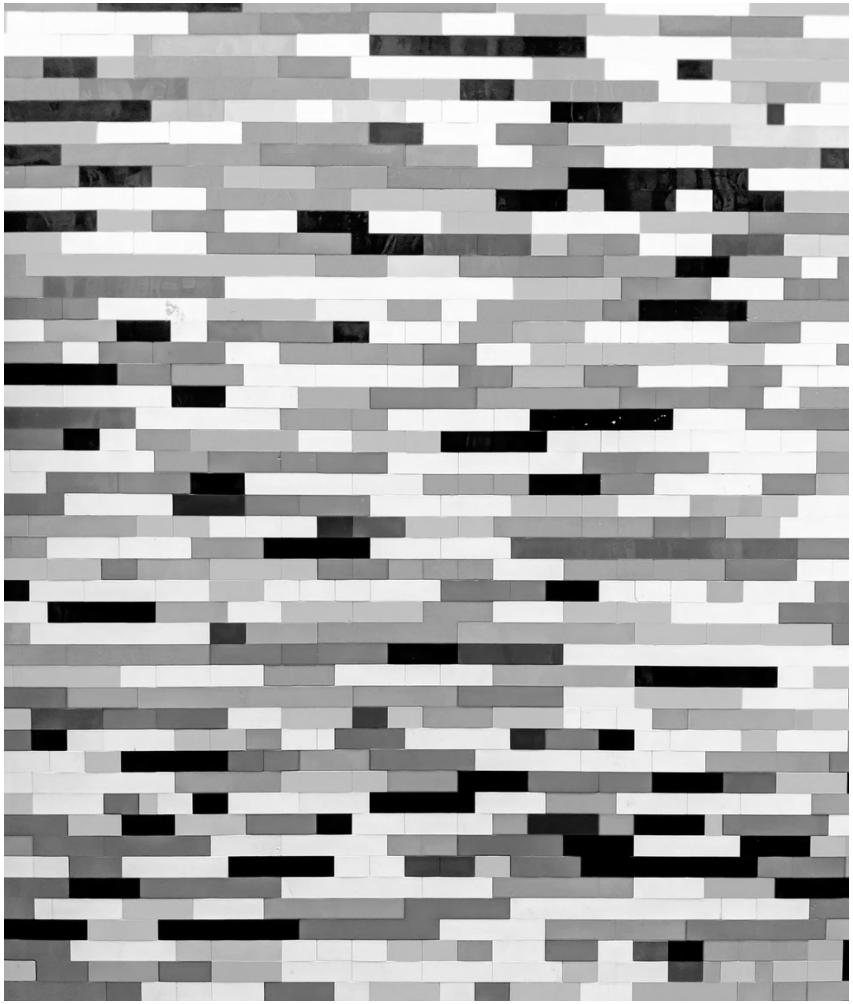
DEEPer

Further Databases
Week 5 Session 2

TODAY'S SESSION



01 Databases Continued

02 PDO++

Committing & Pushing

- It’s important to push work!
- Pushing means we can see your code on BitBucket, and we can track your progress and help when you’re stuck
- Think of it like a backup of your work – we can see a history of changes and easily revert if something breaks

```
git add . && git commit -m "Task complete" && git push
```

New Script!

- Download “W5S2.zip” from Slack
- Open Downloads in Finder and double click the zip file to extract
- In iTerm: `sh ~/Downloads/W5S2/prepare-lecture.sh`
- This will:
  - Prepare a project database which we’ll be adding to during this lecture (complete with a product table and some sample data)
  - Copy all of today’s code samples to a W5S2 folder in your `~/projects` directory
  - Open the new directory in PhpStorm

Today's Code Alongs

- All of today's exercises will be performed in the new project database
- The queries are available in the W5S2 folder

Databases Continued

01

Primary Keys

- An index allows us to find data quickly
- Like the back section of the Argos catalogue!
- All tables should have a `PRIMARY KEY`, usually on the `id` column. This means that:
  - The value **MUST** be unique
  - The value **MUST NOT** contain a `NULL` value
  - The value **CANNOT** be changed afterwards
- A table may only have one primary key
- This is usually done as part of the `CREATE TABLE` statement, but they can be retrofitted

Primary Keys – Code Along


```sql
-- File: ~/projects/W5S2/01-create-table-pk.sql

CREATE TABLE `user` (
  `id` INT AUTO_INCREMENT PRIMARY KEY,
  `name` VARCHAR(50)
);
```

Altering Tables

- Quite often, requirements may change and new columns may need to be added to an existing table
- The structure of a table can be changed with an `ALTER TABLE` statement

Altering Tables – Code Along


```sql
-- File: ~/projects/W5S2/02-alter-table-add-column.sql

ALTER TABLE `user`
  ADD COLUMN `email_address` VARCHAR(50);
```

Unique Constraints

- While our `id` is generally a `PRIMARY KEY`, we may have other values which must be unique
- The most common example of this is an email address on a users table, preventing users having multiple accounts
- If we attempt to `INSERT` duplicate data, MySQL will throw an error
- This saves us from needing to `SELECT` and confirm that 0 rows are returned before attempting to `INSERT`
- Like `PRIMARY KEY`s, unique constraints can be added as part of the `CREATE TABLE` statement or retrofitted using `ALTER TABLE`

Unique Constraints – Example Query


```sql
-- File: ~/projects/W5S2/03-create-table-unique.sql

CREATE TABLE `user` (
  `id` INT AUTO_INCREMENT PRIMARY KEY,
  `name` VARCHAR(50),
  `email_address` VARCHAR(50) UNIQUE
);
```

Unique Constraints – Code Along


```sql
-- File: ~/projects/W5S2/04-alter-table-unique.sql

ALTER TABLE `user`
  ADD CONSTRAINT UNIQUE (email_address);
```

Relational Tables

- Some tables may contain links to data in other tables
- Data can be referenced via the ID in the related table
- We can retrieve data by adding a `JOIN` clause to a `SELECT` statement

Relational Tables – Code Along


```sql
-- File: ~/projects/W5S2/05-create-checkin-table.sql

CREATE TABLE `checkin` (
  `id` INT AUTO_INCREMENT PRIMARY KEY,
  `product_id` INT,
  `name` VARCHAR(50),
  `rating` INT,
  `review` VARCHAR(500),
  `posted` DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Relational Tables – Code Along


```
-- File: ~/projects/W5S2/06-insert-checkins.sql

INSERT INTO `checkin` (`product_id`, `name`, `rating`, `review`)
  VALUES(1, 'John Smith', 2, 'Not keen on this');

INSERT INTO `checkin` (`product_id`, `name`, `rating`, `review`)
  VALUES(1, 'Jane Doe', 5, 'Love it!');
```

Selecting Data

- There are several types of `JOIN`, but today we will be focusing on the two most common:
  - `LEFT JOIN` – selects from A even if there is nothing in B
  - `RIGHT JOIN` – selects from B even if there is nothing in A
- It is possible to JOIN with multiple tables

Anatomy of a `JOIN` Clause

- In our column list, we can prefix the columns we wish to `SELECT` with their respective table names
- We still `SELECT` from a single table
- Then `JOIN` other tables `ON` a specified condition (similar to `WHERE`)
- `JOIN` clauses come before the `WHERE` clause

Selecting Data – Code Along


```sql
-- File: ~/projects/W5S2/07-left-join.sql

SELECT `product`.`id`, `product`.`title`, `checkin`.`rating`
  FROM `product`
  LEFT JOIN `checkin` ON `checkin`.`product_id` = `product`.`id`;
```

id	title	rating
1	Macbook Pro	2
1	Macbook Pro	5
2	Pencil	NULL
3	Ruler	NULL
4	Coat	NULL
5	Milk	NULL

Selecting Data – Code Along


```sql
-- File: ~/projects/W5S2/08-right-join.sql

SELECT `product`.`id`, `product`.`title`, `checkin`.`rating`
  FROM `product`
  RIGHT JOIN `checkin` ON `checkin`.`product_id` = `product`.`id`;
```

id	title	rating
1	Macbook Pro	2
1	Macbook Pro	5

GROUPing Data

- The `GROUP BY` clause comes at the end of the `SELECT` statement
- It allows us to reduce the number of rows we retrieve into something meaningful
- We're currently retrieving multiple rows for a single product
- In the previous examples, we have two check-ins for a Macbook Pro with ratings of 2 and 5
- MySQL offers a number of handy functions – but for now, we'll just be using `AVG` to find the average

GROUPing Data – Code Along


```sql
-- File: ~/projects/W5S2/09-group-by.sql

SELECT `product`.`id`, `product`.`title`,
    AVG(`checkin`.`rating`) AS `average_rating`
  FROM `product`
  LEFT JOIN `checkin` ON `checkin`.`product_id` = `product`.`id`
  GROUP BY `product`.`id`;
```

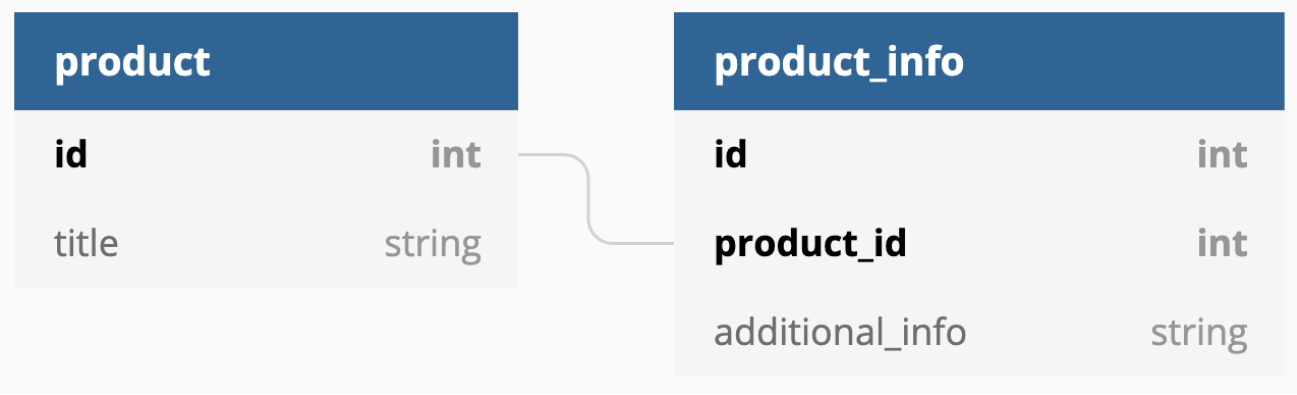
id	title	average_rating
1	Macbook Pro	3.5000
2	Pencil	NULL
3	Ruler	NULL
4	Coat	NULL
5	Milk	NULL

Types of Relationship

- There are three types of relation:
  - One-to-One
  - One-to-Many
  - Many-to-Many

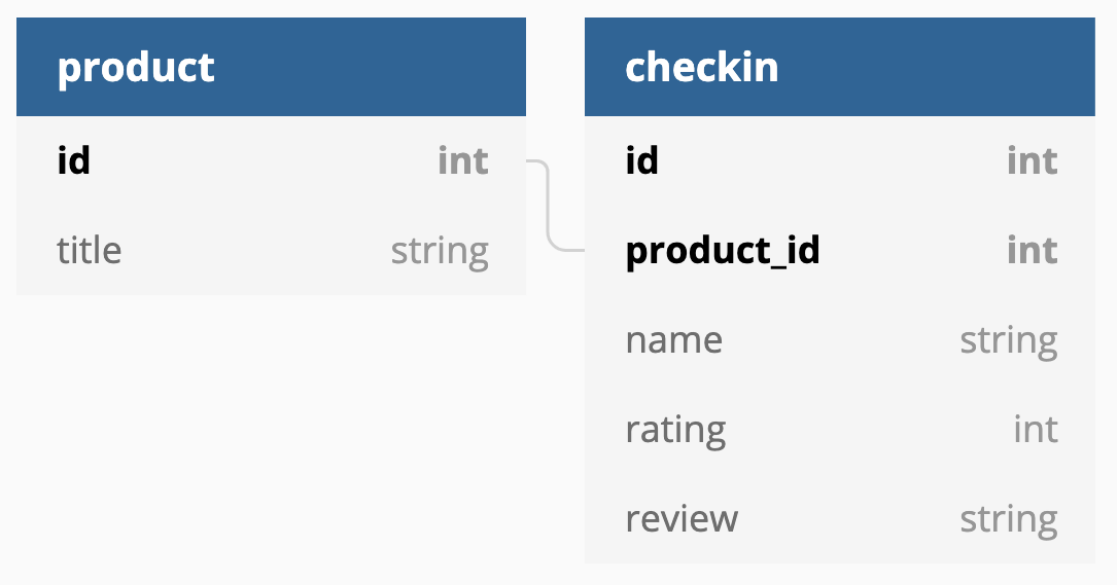
One-to-One Relationships

- The least common type of relation
- Rarely holds much value, but can be used for security purposes or splitting up large tables with lots of columns



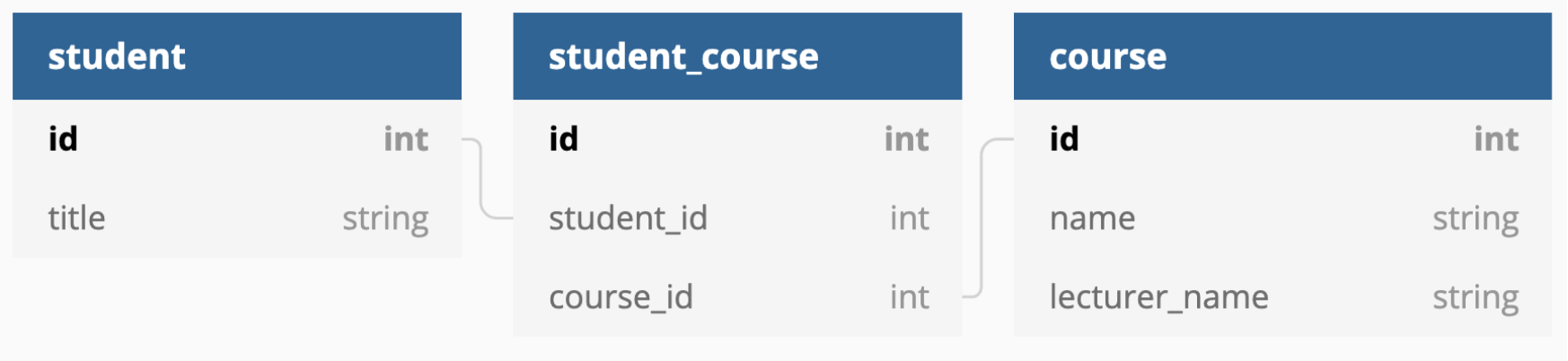
One-to-Many Relationships

- Links a single parent record to many child records
- The child table contains a column which references its parent ID



Many-to-Many Relationships

- Links many parent record to many child records
- A “join” table must be created to store references
- Normally just holds the parent ID and child ID



Searching

- Commonly, we may want to search for records containing a value – although we may not know the *exact* value
- Where we've previously used `WHERE column = 'value'`, we can use `LIKE`
- We can use percentage signs (%) to indicate a *wildcard* value
- e.g.
  - `WHERE column LIKE '%value'` – ending with "value"
  - `WHERE column LIKE 'value%'` – beginning with "value"
  - `WHERE column LIKE '%value%'` – containing "value"

Searching – Code Along


```sql
-- File: ~/projects/W5S2/10-like.sql

SELECT * FROM `product` WHERE `title` LIKE '%p%';
```

id	title
1	Macbook Pro
2	Pencil

02

PDO++

Using Entities

- Previously, we've seen a result set from a database being returned as a key/value array
- We can also *hydrate* our classes, giving us a more meaningful representation of the data
- Limitations of hydrating a class directly from PDO include:
  - Properties must match columns **exactly** so unfortunately we have to snake case if our columns are snake cased!
  - Limited by native types like `string` and `int` – nothing like `DateTime`

Using Entities – Code Along

File: ~/projects/deeper/exercises/week-05/lecture/entities.php
Browser: http://localhost/exercises/week-05/lecture/entities.php?id=1

```php
<?php

if (!isset($_GET['id'])) {
    die('Please specify an id in the URL');
}

class Product
{
    public int $id;
    public string $title;
}

$db = new PDO('mysql:host=mysql;dbname=project', 'root', 'root');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

$stmt = $db->prepare('SELECT * FROM `product` WHERE `id` = :id');
$stmt->execute(['id' => $_GET['id']]);

$product = $stmt->fetchObject(Product::class);

var_dump($product);
```

```
/var/www/sites/deeper.local/exercises/week-05/lecture/entities.php:21:
object(Product)[3]
  public int 'id' => int 1
  public string 'title' => string 'Macbook Pro' (length=11)
```

Hydrating Related Entities

- In order to hydrate related entities, usually we would perform a join and build a custom hydrator to hydrate entities from a single result set
- For today, we will do the simpler option and perform a second query
- This can become problematic for large datasets as you may end up running hundreds, if not thousands, of queries to get the data you need!

Hydrating Related Entities – Code Along


```
File: ~/projects/deeper/exercises/week-05/lecture/entities.php
Browser: http://localhost/exercises/week-05/lecture/entities.php?id=1

<?php

// ...

class Product
{
    // ...
    /** @var Checkin[] **/
    public array $checkIns;
}

// ...
```

```
class CheckIn
{
    public int $id;
    public int $product_id;
    public string $name;
    public int $rating;
    public string $review;
    public string $posted;
}
```

```
$stmt = $db->prepare('SELECT * FROM `checkin` WHERE `product_id` = :product_id');
$stmt->execute(['product_id' => $product->id]);

$product->checkIns = $stmt->fetchAll(PDO::FETCH_CLASS, CheckIn::class);

var_dump($product);
```

Hydrating Related Entities – Code Along


```
/var/www/sites/deeper.local/exercises/week-05/lecture/entities.php:38:
object(Product)[3]
  public int 'id' => int 1
  public string 'title' => string 'Macbook Pro' (length=11)
  public array 'checkIns' =>
    array (size=2)
      0 =>
        object(CheckIn)[2]
          public int 'id' => int 1
          public int 'product_id' => int 1
          public string 'name' => string 'John Smith' (length=10)
          public int 'rating' => int 2
          public string 'review' => string 'Not keen on this' (length=16)
          public string 'posted' => string '2020-09-02 17:08:20' (length=19)
      1 =>
        object(CheckIn)[5]
          public int 'id' => int 2
          public int 'product_id' => int 1
          public string 'name' => string 'Jane Doe' (length=8)
          public int 'rating' => int 5
          public string 'review' => string 'Love it!' (length=8)
          public string 'posted' => string '2020-09-02 17:08:20' (length=19)
```