How to Move Object-Relational Mapping into the Database

David Wheeler
Kineticode
OSCON 2005

Polymorphic Database Design

David Wheeler
Kineticode
OSCON 2005

Executive summary

Executive summary

- Look at simple object-oriented examples

Executive summary

- Look at simple object-oriented examples
- Examine typical database serialization approach

Executive summary

- Look at simple object-oriented examples
- Examine typical database serialization approach
- Make the **database** do the work

Executive summary

- Look at simple object-oriented examples
- Examine typical database serialization approach
- Make the **database** do the work
- Let's give it a try

Start with a class

Start with a class

Start with a class

- Simple UML is good UML

Standard Approach

Standard Approach

- Rules:

Standard Approach

- Rules:
  - Class == Table

Standard Approach

- Rules:
  - Class == Table
  - Object == Row

Standard Approach

- Rules:
  - Class == Table
  - Object == Row
  - Attribute == Column

Standard Approach

- Rules:
  - Class == Table
  - Object == Row
  - Attribute == Column
- Methodology

Standard Approach

- Rules:
  - Class == Table
  - Object == Row
  - Attribute == Column
- Methodology
  - Create Table

Standard Approach

- Rules:
  - Class == Table
  - Object == Row
  - Attribute == Column
- Methodology
  - Create Table
  - Create SELECT query

Create a Table

Create a Table

```
CREATE TABLE album (
    id INTEGER NOT NULL PRIMARY KEY
       AUTOINCREMENT,
    title     TEXT,
    artist    TEXT,
    publisher TEXT,
    isbn      TEXT
);
```

Create a Table

```
CREATE TABLE album (
    id INTEGER NOT NULL PRIMARY KEY
       AUTOINCREMENT,
    title     TEXT,
    artist    TEXT,
    publisher TEXT,
    isbn      TEXT
);
```

- Not much to see here

Create a Table

```
CREATE TABLE album (
    id INTEGER NOT NULL PRIMARY KEY
       AUTOINCREMENT,
    title     TEXT,
    artist    TEXT,
    publisher TEXT,
    isbn      TEXT
);
```

- Not much to see here
- **Use primary keys!**

Write SELECT query

Write SELECT query

```
SELECT id, title, artist, publisher,
       isbn
FROM   album;
```

Write SELECT query

```
SELECT id, title, artist, publisher,
       isbn
FROM   album
WHERE  id = ?;
```

Write SELECT query

```
SELECT id, title, artist, publisher,
       isbn
FROM   album
WHERE  id = ?;
```

- Gee, that was easy

Write SELECT query

```
SELECT id, title, artist, publisher,
       isbn
FROM   album
WHERE  id = ?;
```

- Gee, that was easy
- Inflate objects from each row

Write SELECT query

```
SELECT id, title, artist, publisher,
       isbn
FROM   album
WHERE  id = ?;
```

- Gee, that was easy
- Inflate objects from each row
- Let's see if it works...

Examine the output

id	title	artist	publisher	isbn
1	Rage Against the Machine	Rage Against the Machine	Atlantic	012848939
2	OK Computer	Radiohead	Atlantic	934293249
3	Elephant	The White Stripes	BMG	000864P55
4	Get Behind Me Satan	The White Stripes	BMG	949394595
5	Purple Rain	Prince and the Revolution	Warner Bro	638594823
6	Roots of a Revolution	James Brown	Polydor	496758395
7	Blood Sugar Sex Magik	Red Hot Chili Peppers	Warnter Br	957483845
8	Frizzle Fry	Primus	Atlantic	685968345

Examine the output

id	title	artist	publisher	isbn
1	Rage Against the Machine	Rage Against the Machine	Atlantic	012848939
2	OK Computer	Radiohead	Atlantic	934293249
3	Elephant	The White Stripes	BMG	000864P55
4	Get Behind Me Satan	The White Stripes	BMG	949394595
5	Purple Rain	Prince and the Revolution	Warner Bro	638594823
6	Roots of a Revolution	James Brown	Polydor	496758395
7	Blood Sugar Sex Magik	Red Hot Chili Peppers	Warnter Br	957483845
8	Frizzle Fry	Primus	Atlantic	685968345

- Hey cool!

Examine the output

id	title	artist	publisher	isbn
1	Rage Against the Machine	Rage Against the Machine	Atlantic	012848939
2	OK Computer	Radiohead	Atlantic	934293249
3	Elephant	The White Stripes	BMG	000864P55
4	Get Behind Me Satan	The White Stripes	BMG	949394595
5	Purple Rain	Prince and the Revolution	Warner Bro	638594823
6	Roots of a Revolution	James Brown	Polydor	496758395
7	Blood Sugar Sex Magik	Red Hot Chili Peppers	Warnter Br	957483845
8	Frizzle Fry	Primus	Atlantic	685968345

- Hey cool!
- Single row == single object

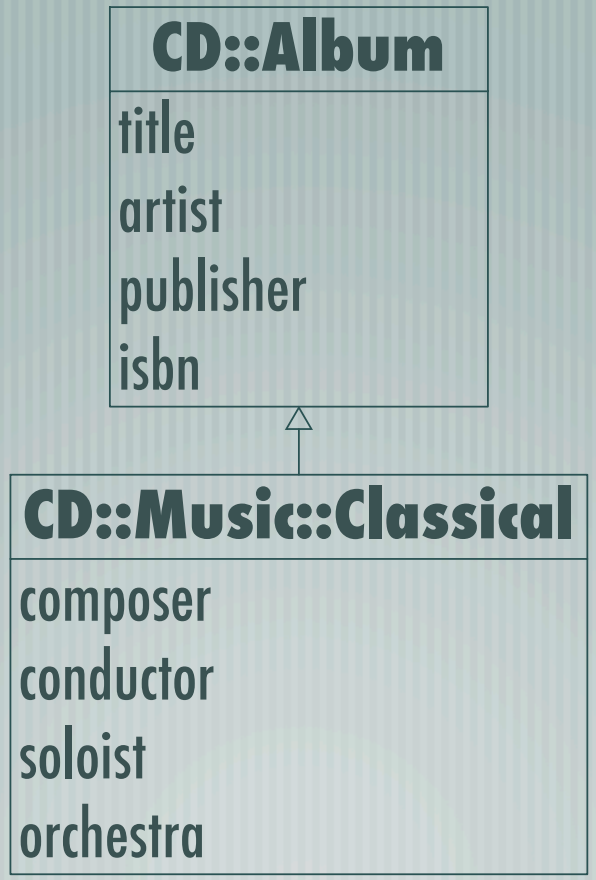
Examine the output

id	title	artist	publisher	isbn
1	Rage Against the Machine	Rage Against the Machine	Atlantic	012848939
2	OK Computer	Radiohead	Atlantic	934293249
3	Elephant	The White Stripes	BMG	000864P55
4	Get Behind Me Satan	The White Stripes	BMG	949394595
5	Purple Rain	Prince and the Revolution	Warner Bro	638594823
6	Roots of a Revolution	James Brown	Polydor	496758395
7	Blood Sugar Sex Magik	Red Hot Chili Peppers	Warnter Br	957483845
8	Frizzle Fry	Primus	Atlantic	685968345

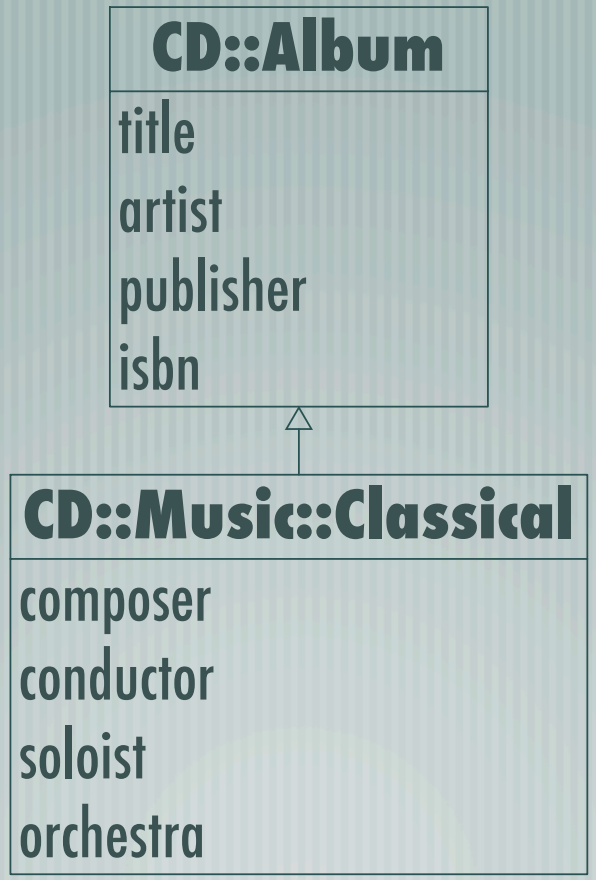
- Hey cool!
- Single row == single object
- Single column == single attribute

Let's add a subclass

Let's add a subclass



Let's add a subclass



- Piece of cake, right?

Typical Subclassing Approach

Typical Subclassing Approach

- Create another table

Typical Subclassing Approach

- Create another table
- Reference the parent class' table

Typical Subclassing Approach

- Create another table
- Reference the parent class' table
- Write a two queries for each table

Typical Subclassing Approach

- Create another table
- Reference the parent class' table
- Write a two queries for each table
- Write a JOIN query for both tables

Create the new table

Create the new table

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
               AUTOINCREMENT,
    album_id   INTEGER NOT NULL
               REFERENCES album(id),
    composer   TEXT,
    conductor  TEXT,
    orchestra  TEXT
);
```

Create the new table

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
               AUTOINCREMENT,
    album_id   INTEGER NOT NULL
               REFERENCES album(id),
    composer   TEXT,
    conductor  TEXT,
    orchestra  TEXT
);
```

- Hrm...not bad

Create the new table

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
               AUTOINCREMENT,
    album_id   INTEGER NOT NULL
               REFERENCES album(id),
    composer   TEXT,
    conductor  TEXT,
    orchestra  TEXT
);
```

- Hrm...not bad
- You use foreign keys, **right?**

Create the new table

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
               AUTOINCREMENT,
    album_id   INTEGER NOT NULL
               REFERENCES album(id),
    composer   TEXT,
    conductor  TEXT,
    orchestra  TEXT
);
```

- Hrm...not bad
- You use foreign keys, **right?**
- What about the SELECT statement?

Write SELECT query

Write SELECT query

```
SELECT m.id, title, artist, publisher,
       isbn, c.id composer, conductor,
       orchestra
FROM   album m JOIN classical c
       ON m.id = c.album_id;
```

Write SELECT query

```
SELECT m.id, title, artist, publisher,
       isbn, c.id composer, conductor,
       orchestra
FROM   album m JOIN classical c
       ON m.id = c.album_id;
```

- Also not too bad

Write SELECT query

```
SELECT m.id, title, artist, publisher,
       isbn, c.id composer, conductor,
       orchestra
FROM   album m JOIN classical c
       ON m.id = c.album_id;
```

- Also not too bad
- Let's see how it works...

What's wrong here?

id	title	id	composer
9	Emperor Concerto	1	Beethoven
10	Eroica Symphony	2	Beethoven
11	Amadeus Soundtrack	3	Mozart

What's wrong here?

id	title	id	composer
9	Emperor Concerto	1	Beethoven
10	Eroica Symphony	2	Beethoven
11	Amadeus Soundtrack	3	Mozart

- Two IDs? That's annoying

What's wrong here?

id	title	id	composer
9	Emperor Concerto	1	Beethoven
10	Eroica Symphony	2	Beethoven
11	Amadeus Soundtrack	3	Mozart

- Two IDs? That's annoying
- Different ID for the same object

What's wrong here?

id	title	id	composer
9	Emperor Concerto	1	Beethoven
10	Eroica Symphony	2	Beethoven
11	Amadeus Soundtrack	3	Mozart

- Two IDs? That's annoying
- Different ID for the same object
- Which to use?

What's wrong here?

id	title	id	composer
9	Emperor Concerto	1	Beethoven
10	Eroica Symphony	2	Beethoven
11	Amadeus Soundtrack	3	Mozart

- Two IDs? That's annoying
- Different ID for the same object
- Which to use?
- **Stick to one and avoid the problem.**

Create the table again

Create the table again

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
       REFERENCES album (id),
    composer  TEXT,
    conductor TEXT,
    orchestra TEXT
);
```

Create the table again

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
       REFERENCES album (id),
    composer  TEXT,
    conductor TEXT,
    orchestra TEXT
);
```

- Gee, that's simpler

Create the table again

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
       REFERENCES album (id),
    composer  TEXT,
    conductor TEXT,
    orchestra TEXT
);
```

- Gee, that's simpler
- The primary key is also a foreign key

Create the table again

```
CREATE TABLE classical (
    id INTEGER NOT NULL PRIMARY KEY
       REFERENCES album (id),
    composer  TEXT,
    conductor TEXT,
    orchestra TEXT
);
```

- Gee, that's simpler
- The primary key is also a foreign key
- You still use the foreign key constraint, **right?**

Write SELECT query again

Write SELECT query again

```
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN classical c
       ON m.id = c.id;
```

Write SELECT query again

```
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN classical c
       ON m.id = c.id;
```

- That got a bit simpler, too

Write SELECT query again

```
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN classical c
       ON m.id = c.id;
```

- That got a bit simpler, too
- Disambiguate the ID column

Write SELECT query again

```
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN classical c
       ON m.id = c.id;
```

- That got a bit simpler, too
- Disambiguate the ID column
- Let's see if it works...

Now we're talkin'!

id	title	composer	conductor
9	Emperor Concerto	Beethoven	Ozawa
10	Eroica Symphony	Beethoven	Bernstein
11	Amadeus Soundtrack	Mozart	Neville Ma

Now we're talkin'!

id	title	composer	conductor
9	Emperor Concerto	Beethoven	Ozawa
10	Eroica Symphony	Beethoven	Bernstein
11	Amadeus Soundtrack	Mozart	Neville Ma

- Just one ID

Now we're talkin'!

id	title	composer	conductor
9	Emperor Concerto	Beethoven	Ozawa
10	Eroica Symphony	Beethoven	Bernstein
11	Amadeus Soundtrack	Mozart	Neville Ma

- Just one ID
- But why must we write a JOIN query?

Now we're talkin'!

```
id          title                composer      conductor
----------  -------------------  ------------  ----------
9           Emperor Concerto     Beethoven     Ozawa
10          Eroica Symphony      Beethoven     Bernstein
11          Amadeus Soundtrack   Mozart        Neville Ma
```

- Just one ID
- But why must we write a JOIN query?
- Couldn't it be **even simpler**?

Now we're talkin'!

```
id          title                composer      conductor
----------  -------------------  ------------  ----------
9           Emperor Concerto     Beethoven     Ozawa
10          Eroica Symphony      Beethoven     Bernstein
11          Amadeus Soundtrack   Mozart        Neville Ma
```

- Just one ID
- But why must we write a JOIN query?
- Couldn't it be **even simpler**?
- Can't we truly have one class == one table?

Yes, it can!

```
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN classical c
       ON m.id = c.id;
```

Yes, it can!

```
CREATE VIEW classical AS
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN album_classical c
       ON m.id = c.id;
```

Yes, it can!

```
CREATE VIEW classical AS
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN album_classical c
       ON m.id = c.id;
```

- Now we have a single "table"

Yes, it can!

```
CREATE VIEW classical AS
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN album_classical c
       ON m.id = c.id;
```

- Now we have a single "table"
- Rename subclass table to represent inheritance

Yes, it can!

```
CREATE VIEW classical AS
SELECT m.id AS id, title, artist,
       publisher, isbn, composer,
       conductor, orchestra
FROM   album m JOIN album_classical c
       ON m.id = c.id;
```

- Now we have a single "table"
- Rename subclass table to represent inheritance
- Supported by PostgreSQL, SQLite, and MySQL

Query the VIEW

Query the VIEW

```
SELECT id, title, artist, publisher,
       isbn, composer, conductor,
       orchestra
FROM   classical;
```

Query the VIEW

```
SELECT id, title, artist, publisher,
       isbn, composer, conductor,
       orchestra
FROM   classical;
```

- Also a bit simpler

Query the VIEW

```
SELECT id, title, artist, publisher,
       isbn, composer, conductor,
       orchestra
FROM   classical;
```

- Also a bit simpler
- The database handles inheritance transparently

Query the VIEW

```
SELECT id, title, artist, publisher,
       isbn, composer, conductor,
       orchestra
FROM   classical;
```

- Also a bit simpler
- The database handles inheritance transparently
- The VIEW is compiled, thus faster

Query the VIEW

```
SELECT id, title, artist, publisher,
       isbn, composer, conductor,
       orchestra
FROM   classical;
```

- Also a bit simpler
- The database handles inheritance transparently
- The VIEW is compiled, thus faster
- Let's see if it works...

Get the same output

```
id          title               composer      conductor
----------  ------------------  ------------  ----------
9           Emperor Concerto    Beethoven     Ozawa
10          Eroica Symphony     Beethoven     Bernstein
11          Amadeus Soundtrack  Mozart        Neville Ma
```

Get the same output

```
id          title               composer      conductor
----------  ------------------  ------------  ----------
9           Emperor Concerto    Beethoven     Ozawa
10          Eroica Symphony     Beethoven     Bernstein
11          Amadeus Soundtrack  Mozart        Neville Ma
```

- Hey, awesome!

Get the same output

```
id          title               composer      conductor
----------  ------------------  ------------  ----------
9           Emperor Concerto    Beethoven     Ozawa
10          Eroica Symphony     Beethoven     Bernstein
11          Amadeus Soundtrack  Mozart        Neville Ma
```

- Hey, awesome!
- Returns the same records as before

Get the same output

```
id          title               composer      conductor
----------  ------------------  ------------  ----------
9           Emperor Concerto    Beethoven     Ozawa
10          Eroica Symphony     Beethoven     Bernstein
11          Amadeus Soundtrack  Mozart        Neville Ma
```

- Hey, awesome!
- Returns the same records as before
- It's faster, too

Get the same output

```
id          title               composer      conductor
----------  ------------------  ------------  ----------
9           Emperor Concerto    Beethoven     Ozawa
10          Eroica Symphony     Beethoven     Bernstein
11          Amadeus Soundtrack  Mozart        Neville Ma
```

- Hey, awesome!
- Returns the same records as before
- It's faster, too
- But what about INSERTs, UPDATEs, and DELETEs?

Back to the base class

Back to the base class

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Kid B', 'Radiohed', 'Polygramm', '0000');
```

Back to the base class

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Kid B', 'Radiohed', 'Polygramm', '0000');

UPDATE album
SET    title     = 'Kid A',
       artist    = 'Radiohead',
       publisher = 'Polygram',
       isbn      = '4959'
WHERE  id        = 12;
```

Back to the base class

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Kid B', 'Radiohed', 'Polygramm', '0000');

UPDATE album
SET    title     = 'Kid A',
       artist    = 'Radiohead',
       publisher = 'Polygram',
       isbn      = '4959'
WHERE  id        = 12;

DELETE FROM album
WHERE  id        = 12;
```

Back to the base class

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Kid B', 'Radiohed', 'Polygramm', '0000');

UPDATE album
SET    title     = 'Kid A',
       artist    = 'Radiohead',
       publisher = 'Polygram',
       isbn      = '4959'
WHERE  id        = 12;

DELETE FROM album
WHERE  id        = 12;
```

- Seems pretty straight-forward

Back to the base class

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Kid B', 'Radiohed', 'Polygramm', '0000');

UPDATE album
SET    title     = 'Kid A',
       artist    = 'Radiohead',
       publisher = 'Polygram',
       isbn      = '4959'
WHERE  id        = 12;

DELETE FROM album
WHERE  id        = 12;
```

- Seems pretty straight-forward
- Let's try it...

And now the subclass

And now the subclass

```
INSERT INTO classical (title, artist, publisher, isbn,
                       composer, conductor, orchestra)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323',
        'Verious', 'Jon Williams', 'Pittsburg Symphony');
```

And now the subclass

```
INSERT INTO classical (title, artist, publisher, isbn,
                       composer, conductor, orchestra)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323',
        'Verious', 'Jon Williams', 'Pittsburg Symphony');

UPDATE classical
SET    title     = 'Cinema Serenade',
       artist    = 'Itzak Perlman',
       publisher = 'BMG',
       isbn      = '2764',
       composer  = 'Various',
       conductor = 'John Williams',
       orchestra = 'Pittsburgh Symphony'
WHERE  id        = 13;
```

And now the subclass

```
INSERT INTO classical (title, artist, publisher, isbn,
                       composer, conductor, orchestra)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323',
        'Verious', 'Jon Williams', 'Pittsburg Symphony');

UPDATE classical
SET    title     = 'Cinema Serenade',
       artist    = 'Itzak Perlman',
       publisher = 'BMG',
       isbn      = '2764',
       composer  = 'Various',
       conductor = 'John Williams',
       orchestra = 'Pittsburgh Symphony'
WHERE  id        = 13;

DELETE FROM classical
WHERE  id = 13;
```

And now the subclass

```
INSERT INTO classical (title, artist, publisher, isbn,
                       composer, conductor, orchestra)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323',
        'Verious', 'Jon Williams', 'Pittsburg Symphony');

UPDATE classical
SET    title     = 'Cinema Serenade',
       artist    = 'Itzak Perlman',
       publisher = 'BMG',
       isbn      = '2764',
       composer  = 'Various',
       conductor = 'John Williams',
       orchestra = 'Pittsburgh Symphony'
WHERE  id        = 13;

DELETE FROM classical
WHERE  id = 13;
```

Let's see if these work...

D'oh! What now?

```
INSERT INTO classical (title, artist, publisher, isbn,
                       composer, conductor, orchestra)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323',
        'Verious', 'Jon Williams', 'Pittsburg Symphony');

SQL error: cannot modify classical because it is a view
```

D'oh! What now?

```
INSERT INTO classical (title, artist, publisher, isbn,
                       composer, conductor, orchestra)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323',
        'Verious', 'Jon Williams', 'Pittsburg Symphony');

SQL error: cannot modify classical because it is a view
```

- VIEWs are not updatable

D'oh! What now?

```
INSERT INTO classical (title, artist, publisher, isbn,
                       composer, conductor, orchestra)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323',
        'Verious', 'Jon Williams', 'Pittsburg Symphony');

SQL error: cannot modify classical because it is a view
```

- VIEWs are not updatable
- Must modify the two tables separately

INSERT classical object

INSERT classical object

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323');

INSERT INTO album_classical
       (id, composer, conductor, orchestra)
VALUES (last_insert_rowid(), 'Verious', 'Jon Williams',
        'Pittsburg Symphony');
```

INSERT classical object

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323');

INSERT INTO album_classical
       (id, composer, conductor, orchestra)
VALUES (last_insert_rowid(), 'Verious', 'Jon Williams',
        'Pittsburg Symphony');
```

- We have to modify the two table separately

INSERT classical object

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323');

INSERT INTO album_classical
       (id, composer, conductor, orchestra)
VALUES (last_insert_rowid(), 'Verious', 'Jon Williams',
        'Pittsburg Symphony');
```

- We have to modify the two table separately
- Use function to populate subclass ID

INSERT classical object

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323');

INSERT INTO album_classical
       (id, composer, conductor, orchestra)
VALUES (last_insert_rowid(), 'Verious', 'Jon Williams',
        'Pittsburg Symphony');
```

- We have to modify the two table separately
- Use function to populate subclass ID
  - SQLite: `last_insert_rowid()`

INSERT classical object

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323');

INSERT INTO album_classical
       (id, composer, conductor, orchestra)
VALUES (last_insert_rowid(), 'Verious', 'Jon Williams',
        'Pittsburg Symphony');
```

- We have to modify the two table separately
- Use function to populate subclass ID
  - SQLite: `last_insert_rowid()`
  - MySQL: `last_insert_id()`

INSERT classical object

```
INSERT INTO album (title, artist, publisher, isbn)
VALUES ('Cinema Serenad', 'Itzack Perlman', 'BMC', '2323');

INSERT INTO album_classical
       (id, composer, conductor, orchestra)
VALUES (last_insert_rowid(), 'Verious', 'Jon Williams',
        'Pittsburg Symphony');
```

- We have to modify the two table separately
- Use function to populate subclass ID
  - SQLite: `last_insert_rowid()`
  - MySQL: `last_insert_id()`
  - PostgreSQL: `CURRVAL('album_id_seq');`

UPDATE classical object

UPDATE classical object

```
UPDATE album
SET    title     = 'Cinema Serenade',
       artist    = 'Itzak Perlman',
       publisher = 'BMG',
       isbn      = '2764'
WHERE  id = 13;

UPDATE album_classical
SET    composer  = 'Various',
       conductor = 'John Williams',
       orchestra = 'Pittsburgh Symphony'
WHERE  id = 13;
```

UPDATE classical object

```
UPDATE album
SET    title     = 'Cinema Serenade',
       artist    = 'Itzak Perlman',
       publisher = 'BMG',
       isbn      = '2764'
WHERE  id = 13;

UPDATE album_classical
SET    composer  = 'Various',
       conductor = 'John Williams',
       orchestra = 'Pittsburgh Symphony'
WHERE  id = 13;
```

- Again, modify the two table separately

DELETE classical object

DELETE classical object

```
DELETE FROM album_classical
WHERE  id = 13;

DELETE FROM album
WHERE  id = 13;
```

DELETE classical object

```
DELETE FROM album_classical
WHERE  id = 13;

DELETE FROM album
WHERE  id = 13;
```

- And again, modify the two table separately

DELETE classical object

```
DELETE FROM album_classical
WHERE  id = 13;

DELETE FROM album
WHERE  id = 13;
```

- And again, modify the two table separately
- Unless your foreign key is ON DELETE CASCADE

DELETE classical object

```
DELETE FROM album_classical
WHERE  id = 13;

DELETE FROM album
WHERE  id = 13;
```

- And again, modify the two table separately
- Unless your foreign key is ON DELETE CASCADE
- Let's see how these queries work...

Is there no other way?

Is there no other way?

- The more complex your relations, the more queries

Is there no other way?

- The more complex your relations, the more queries
- All because you can't update VIEWs

Is there no other way?

- The more complex your relations, the more queries
- All because you can't update VIEWs
- Or **can** you?

Updatable views

Updatable views

- SQLite supports triggers on VIEWs

Updatable views

- SQLite supports triggers on VIEWs
- PostgreSQL supports rules on VIEWs

Updatable views

- SQLite supports triggers on VIEWs
- PostgreSQL supports rules on VIEWs
- With work, you **can** INSERT, UPDATE, and DELETE on VIEWs

SQLite INSERT trigger

SQLite INSERT trigger

```
CREATE TRIGGER insert_classical
INSTEAD OF INSERT ON classical
FOR EACH ROW BEGIN
  INSERT INTO album (title, artist, publisher, isbn)
  VALUES (NEW.title, NEW.artist, NEW.publisher, NEW.isbn);

  INSERT INTO album_classical
         (id, composer, conductor, orchestra)
  VALUES (last_insert_rowid(), NEW.composer, NEW.conductor,
          NEW.orchestra);
END;
```

SQLite INSERT trigger

```
CREATE TRIGGER insert_classical
INSTEAD OF INSERT ON classical
FOR EACH ROW BEGIN
  INSERT INTO album (title, artist, publisher, isbn)
  VALUES (NEW.title, NEW.artist, NEW.publisher, NEW.isbn);

  INSERT INTO album_classical
         (id, composer, conductor, orchestra)
  VALUES (last_insert_rowid(), NEW.composer, NEW.conductor,
          NEW.orchestra);
END;
```

- Use NEW variable to populate values

SQLite INSERT trigger

```
CREATE TRIGGER insert_classical
INSTEAD OF INSERT ON classical
FOR EACH ROW BEGIN
  INSERT INTO album (title, artist, publisher, isbn)
  VALUES (NEW.title, NEW.artist, NEW.publisher, NEW.isbn);

  INSERT INTO album_classical
         (id, composer, conductor, orchestra)
  VALUES (last_insert_rowid(), NEW.composer, NEW.conductor,
          NEW.orchestra);
END;
```

- Use NEW variable to populate values
- Use last_insert_rowid() for classical ID

PostgreSQL INSERT rule

PostgreSQL INSERT rule

```
CREATE RULE insert_classical AS
ON INSERT TO classical DO INSTEAD (
  INSERT INTO album (title, artist, publisher, isbn)
  VALUES (NEW.title, NEW.artist, NEW.publisher, NEW.isbn);

  INSERT INTO album_classical
         (id, composer, conductor, orchestra)
  VALUES (CURRVAL('seq_album_id_seq'), NEW.composer,
          NEW.conductor, NEW.orchestra);
);
```

PostgreSQL INSERT rule

```
CREATE RULE insert_classical AS
ON INSERT TO classical DO INSTEAD (
  INSERT INTO album (title, artist, publisher, isbn)
  VALUES (NEW.title, NEW.artist, NEW.publisher, NEW.isbn);

  INSERT INTO album_classical
         (id, composer, conductor, orchestra)
  VALUES (CURRVAL('seq_album_id_seq'), NEW.composer,
          NEW.conductor, NEW.orchestra);
);
```

- Use NEW variable to populate values

PostgreSQL INSERT rule

```
CREATE RULE insert_classical AS
ON INSERT TO classical DO INSTEAD (
  INSERT INTO album (title, artist, publisher, isbn)
  VALUES (NEW.title, NEW.artist, NEW.publisher, NEW.isbn);

  INSERT INTO album_classical
         (id, composer, conductor, orchestra)
  VALUES (CURRVAL('seq_album_id_seq'), NEW.composer,
          NEW.conductor, NEW.orchestra);
);
```

- Use NEW variable to populate values
- Use CURRVAL() to populate classical ID

SQLite UPDATE trigger

SQLite UPDATE trigger

```
CREATE TRIGGER update_classical
INSTEAD OF UPDATE ON classical
FOR EACH ROW BEGIN
  UPDATE album
  SET    title     = NEW.title,
         artist    = NEW.artist,
         publisher = NEW.publisher,
         isbn      = NEW.isbn
  WHERE  id        = OLD.id;

  UPDATE album_classical
  SET    composer  = NEW.composer,
         conductor = NEW.conductor,
         orchestra = NEW.orchestra
  WHERE  id        = OLD.id;
END;
```

SQLite UPDATE trigger

```
CREATE TRIGGER update_classical
INSTEAD OF UPDATE ON classical
FOR EACH ROW BEGIN
  UPDATE album
  SET    title     = NEW.title,
         artist    = NEW.artist,
         publisher = NEW.publisher,
         isbn      = NEW.isbn
  WHERE  id        = OLD.id;

  UPDATE album_classical
  SET    composer  = NEW.composer,
         conductor = NEW.conductor,
         orchestra = NEW.orchestra
  WHERE  id        = OLD.id;
END;
```

- Use OLD.id variable to reference existing rows

PostgreSQL UPDATE rule

PostgreSQL UPDATE rule

```
CREATE RULE update_classical AS
ON UPDATE TO classical DO INSTEAD (
  UPDATE album
  SET    title     = NEW.title,
         artist    = NEW.artist,
         publisher = NEW.publisher,
         isbn      = NEW.isbn
  WHERE  id        = OLD.id;

  UPDATE album_classical
  SET    composer  = NEW.composer,
         conductor = NEW.conductor,
         orchestra = NEW.orchestra
  WHERE  id        = OLD.id;
);
```

PostgreSQL UPDATE rule

```
CREATE RULE update_classical AS
ON UPDATE TO classical DO INSTEAD (
  UPDATE album
  SET    title     = NEW.title,
         artist    = NEW.artist,
         publisher = NEW.publisher,
         isbn      = NEW.isbn
  WHERE  id        = OLD.id;

  UPDATE album_classical
  SET    composer  = NEW.composer,
         conductor = NEW.conductor,
         orchestra = NEW.orchestra
  WHERE  id        = OLD.id;
);
```

- Use OLD.id variable to reference existing rows

SQLite DELETE trigger

SQLite DELETE trigger

```
CREATE TRIGGER delete_classical
INSTEAD OF DELETE ON classical
FOR EACH ROW BEGIN
  DELETE FROM album_classical
  WHERE  id = OLD.id;

  DELETE FROM album
  WHERE  id = OLD.id;
END;
```

SQLite DELETE trigger

```
CREATE TRIGGER delete_classical
INSTEAD OF DELETE ON classical
FOR EACH ROW BEGIN
  DELETE FROM album_classical
  WHERE  id = OLD.id;

  DELETE FROM album
  WHERE  id = OLD.id;
END;
```

- Use OLD.id to delete the proper row

PostgreSQL DELETE rule

PostgreSQL DELETE rule

```
CREATE RULE delete_classical AS
ON DELETE TO classical DO INSTEAD (
  DELETE FROM album_classical
  WHERE  id = OLD.id;

  DELETE FROM album
  WHERE  id = OLD.id;
);
```

PostgreSQL DELETE rule

```
CREATE RULE delete_classical AS
ON DELETE TO classical DO INSTEAD (
  DELETE FROM album_classical
  WHERE  id = OLD.id;

  DELETE FROM album
  WHERE  id = OLD.id;
);
```

- Use OLD.id to delete the proper row

PostgreSQL DELETE rule

```
CREATE RULE delete_classical AS
ON DELETE TO classical DO INSTEAD (
  DELETE FROM album_classical
  WHERE  id = OLD.id;

  DELETE FROM album
  WHERE  id = OLD.id;
);
```

- Use OLD.id to delete the proper row
- Let's see how they work...

Trigger/Rule Advantages

Trigger/Rule Advantages

- Queries are pre-compiled

Trigger/Rule Advantages

- Queries are pre-compiled
- Much simpler client-side code

Trigger/Rule Advantages

- Queries are pre-compiled
- Much simpler client-side code
- Fewer queries sent to the database

Trigger/Rule Advantages

- Queries are pre-compiled
- Much simpler client-side code
- Fewer queries sent to the database
- Reduced network overhead

Trigger/Rule Advantages

- Queries are pre-compiled
- Much simpler client-side code
- Fewer queries sent to the database
- Reduced network overhead
- Maintains normalization

One-to-many relationships

One-to-many relationships

- Often have one-to-many relationships

One-to-many relationships

- Often have one-to-many relationships
- Let's we add track objects

One-to-many relationships

- Often have one-to-many relationships
- Let's we add track objects
- Each refers to a single album object

One-to-many relationships

- Often have one-to-many relationships
- Let's we add track objects
- Each refers to a single album object
- Often need to know album information for the track

One-to-many relationships

- Often have one-to-many relationships
- Let's we add track objects
- Each refers to a single album object
- Often need to know album information for the track
- Generally requires two queries, one for each object

One-to-many relationships

- Often have one-to-many relationships
- Let's we add track objects
- Each refers to a single album object
- Often need to know album information for the track
- Generally requires two queries, one for each object

```
SELECT id, title, numb, album_id
FROM   track;

SELECT id, title, artist, publisher, isbn
FROM   album
WHERE  id = $album_id;
```

JOIN composite objects

```
SELECT track.id AS id, track.title AS title, numb,
       album.id    AS album__id,
       album.title AS album__title,
       artist      AS album__artist,
       publisher   AS album__publisher,
       isbn        AS album__isbn
FROM   track JOIN album
       ON track.album_id = album.id;
```

JOIN composite objects

```
SELECT track.id AS id, track.title AS title, numb,
       album.id    AS album__id,
       album.title AS album__title,
       artist      AS album__artist,
       publisher   AS album__publisher,
       isbn        AS album__isbn
FROM   track JOIN album
       ON track.album_id = album.id;
```

- Or, do a JOIN, instead

JOIN composite objects

```
SELECT track.id AS id, track.title AS title, numb,
       album.id    AS album__id,
       album.title AS album__title,
       artist      AS album__artist,
       publisher   AS album__publisher,
       isbn        AS album__isbn
FROM   track JOIN album
       ON track.album_id = album.id;
```

- Or, do a JOIN, instead
- Saves network overhead

JOIN composite objects

```
SELECT track.id AS id, track.title AS title, numb,
       album.id    AS album__id,
       album.title AS album__title,
       artist      AS album__artist,
       publisher   AS album__publisher,
       isbn        AS album__isbn
FROM   track JOIN album
       ON track.album_id = album.id;
```

- Or, do a JOIN, instead
- Saves network overhead
- But might as well make a view for it...

JOIN composite objects

```
CREATE VIEW track AS
SELECT _track.id AS id, _track.title AS title, numb,
       album.id    AS album__id,
       album.title AS album__title,
       artist      AS album__artist,
       publisher   AS album__publisher,
       isbn        AS album__isbn
FROM   _track JOIN album
       ON _track.album_id = album.id;
```

- Or, do a JOIN, instead
- Saves network overhead
- But might as well make a view for it...

What about MySQL?

What about MySQL?

- MySQL 5 supports views

What about MySQL?

- MySQL 5 supports views
- Can INSERT, UPDATE, & DELETE on single table views

What about MySQL?

- MySQL 5 supports views
- Can INSERT, UPDATE, & DELETE on single table views
- Cannot on multi-table (JOIN) views

What about MySQL?

- MySQL 5 supports views
- Can INSERT, UPDATE, & DELETE on single table views
- Cannot on multi-table (JOIN) views
- MySQL supports triggers

What about MySQL?

- MySQL 5 supports views
- Can INSERT, UPDATE, & DELETE on single table views
- Cannot on multi-table (JOIN) views
- MySQL supports triggers
- Cannot assign triggers to views

What about MySQL?

- MySQL 5 supports views
- Can INSERT, UPDATE, & DELETE on single table views
- Cannot on multi-table (JOIN) views
- MySQL supports triggers
- Cannot assign triggers to views
- No rules

Resources

Resources

- “PostgreSQL: Introduction and Concepts” by Bruce Momjian
  http://xrl.us/gy3a

Resources

- "PostgreSQL: Introduction and Concepts" by Bruce Momjian http://xrl.us/gy3a
- SQLite: http://www.sqlite.org/

Resources

- "PostgreSQL: Introduction and Concepts" by Bruce Momjian http://xrl.us/gy3a
- SQLite: http://www.sqlite.org/
- MySQL: http://dev.mysql.com/doc/mysql/en/views.html

Thank you

David Wheeler
Kineticode
OSCON 2005