
MONTANA
STATE UNIVERSITY

Optimistic Concurrency

...

The Real World

Transactions

- In the last lecture we discussed *Transactions* as a way to ensure data consistency
- As a motivating example, we considered transferring money between accounts

Transactions

- Today let’s consider a similar problem: booking seats on an airplane flight
- Here we have two users, each considering the seats that they want

An Online Transaction

- SERALIZED logic:
  - Person A reads (views) the available seats
  - Person B reads (views the available seats)
  - Person A attempts to book seats
    - FAIL: Person B has a read lock

An Online Transaction

- What if Person B…
  - Leaves the browser window open in a tab and forgets about it?
  - Closes the window?
  - Refreshes the window?

An Online Transaction

- READ UNCOMMITTED logic:
  - Person A reads (views) the available seats
  - Person A writes (reserves) a seat
  - Person B reads (views the available seats) before COMMIT
  - Person B sees seats as available that are not actually available

An Online Transaction

- The reality here is that we have a transaction that is spanning multiple web requests
- Transactions are expensive and, with the web, we don’t know if users will ever complete their action!

Pessimistic Concurrency

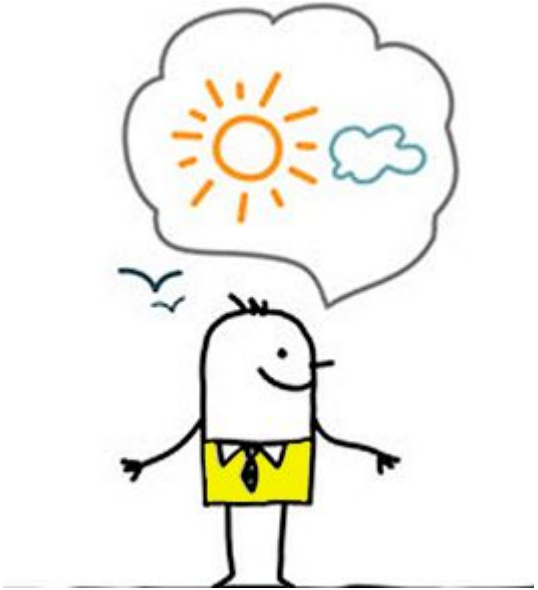
- Thus far we have been using *pessimistic concurrency*
- We assume things will go badly and use a locking/serialization strategy to ensure that they don’t

Pessimistic Concurrency

- Pessimistic concurrency
  - Complex implementation
  - Locks
  - Deadlocks
  - Etc.
- Is it necessary?
- Is it web friendly?
- DBAs tend to be pessimists

Optimistic Concurrency

- There is another option available, however: *Optimistic Concurrency*
- Allow concurrency issues to happen, but....
  - React to them when they do
  - Assume that, for the most part, things will work out
- Developers tend to be optimists
  - Or “ignorant” according to DBAs



Implementing O/C

- There are many ways to implement optimistic concurrency
- Here is one example
- Update the name of the first artist to “DC/AC”, including *all the current values* in the WHERE clause

```
UPDATE artists
SET Name="DC/AC"
WHERE Name="AC/DC" AND ArtistId=1;
```

Implementing O/C

- Check the number of rows updated
  - If 1 row was updated, success
  - If 0 rows were updated, failure
- On success, our optimism paid off!
- On failure, oh well, let the user know and maybe they will try again...

```
UPDATE artists
SET Name="DC/AC"
WHERE Name="AC/DC" AND ArtistId=1;
```

Implementing

- Note that this approach is very web friendly
- If a user is looking at tickets and wanders away, no locks are being held
- If a user accidentally picks a seat already taken (in the meantime), no worse than when using pessimistic concurrency

```
UPDATE artists
SET Name="DC/AC"
WHERE Name="AC/DC" AND ArtistId=1;
```

Implementing

- Another approach: use a version column
- Here we bump the version on every update
- Has the advantage that we don’t need to send every column value in the WHERE clause

```
UPDATE artists
 SET Name="DC/AC", Version=2
WHERE Version=1 AND ArtistId=1;
```

Implementing

- Using a version column is a very common approach and is supported in many database frameworks
- Java standardized this in the JPA - *the Java Persistence API*

```
UPDATE artists
 SET Name="DC/AC", Version=2
WHERE Version=1 AND ArtistId=1;
```

Implementing

- Spring Data is an Object Relational Mapping system
  - We will discuss Spring in more detail in a later lecture
- Here is an optimistically locked data object
- The @Version annotation tells spring to use the version field for optimistic locking

```
@Entity
public class Employee{
  private @Id
  @GeneratedValue
  Long id;
  private String name;
  private String dept;
  private int salary;
  @Version
  private long version;
     ..............
}
```

Implementing

- On a failure, Spring will throw an OptimisticLockingException
  - Gotta love java exception names
- It is up to the application developer to anticipate these exceptions and handle them properly

```
@Entity
public class Employee{
  private @Id
  @GeneratedValue
  Long id;
  private String name;
  private String dept;
  private int salary;
  @Version
  private long version;
     ..............
}
```

Implementing

- Other potential optimistic concurrency implementations
  - A timestamp column (updated at)
  - A random number
  - A cryptographic hash of the entire row

```
@Entity
public class Employee{
  private @Id
  @GeneratedValue
  Long id;
  private String name;
  private String dept;
  private int salary;
  @Version
  private long version;
     .............
}
```

Implementing

- Note that while optimistic concurrency removes the need for locking between application requests, we still need transactions *within* a request
  - What if we dirty-read a version column?

```
@Entity
public class Employee{
  private @Id
  @GeneratedValue
  Long id;
  private String name;
  private String dept;
  private int salary;
  @Version
  private long version;
     ..............
}
```

Academics

- Optimistic concurrency is not discussed much in the database literature
- Probably due to its pragmatic nature
- Good example of the real world/academic distinction

```
@Entity
public class Employee{
  private @Id
  @GeneratedValue
  Long id;
  private String name;
  private String dept;
  private int salary;
  @Version
  private long version;
     ..............
}
```

A Third Option

- There is a third concurrency option: Valhallocking*
  - Just ignore the problem
  - *I live, I die, I live again*
- A surprising number of web applications have taken this approach
  - Advantages: very fast
  - Disadvantages: none

* Made up term, do not use

The Real World

- In the real world, most online systems are either optimistically or valhallocked
  - Shoot for optimistic locking when concurrency is a real issue
  - Don’t feel bad about valhallocking if concurrency isn’t an issue

Optimistic Concurrency

- Transactions are useful for keeping data consistent, but have issues with long-lived and uncertain workflows
- Optimistic Concurrency fits better with the web (and mobile) model with disconnected clients
    - Assumes things will probably work out
    - Column values are used to detect a bad update
    - Transactions are still used within a given request, but not across requests
- Valhallocking isn’t the end of the world for MVPs and many smaller applications

Implementing Optimistic Concurrency

- You are going to implement optimistic concurrency in the project for the Artists table
    - And only for the artists table
    - *Code Review*

> my DBAs fw valhallocking


MONTANA
STATE UNIVERSITY