CS 290B

Scalable Internet Services

Andrew Mutz

October 23, 2014

Today's Agenda

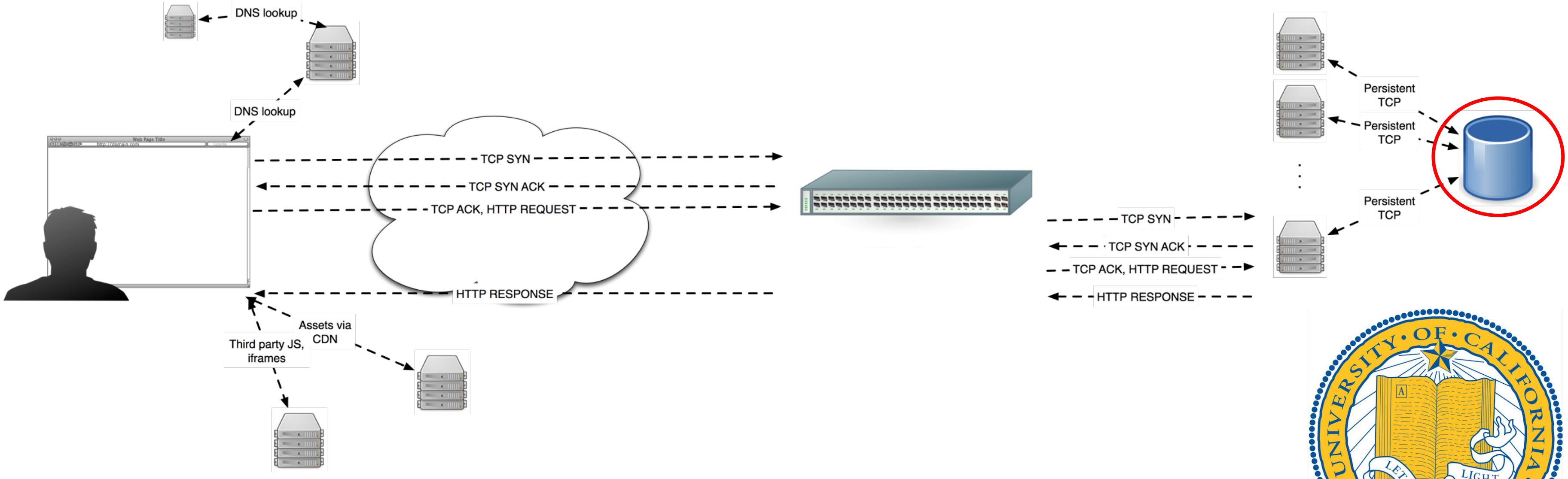
A Stable Data Layer: Motivation

Database Concurrency Control

User Authentication with Devise

For Next Time...

Title



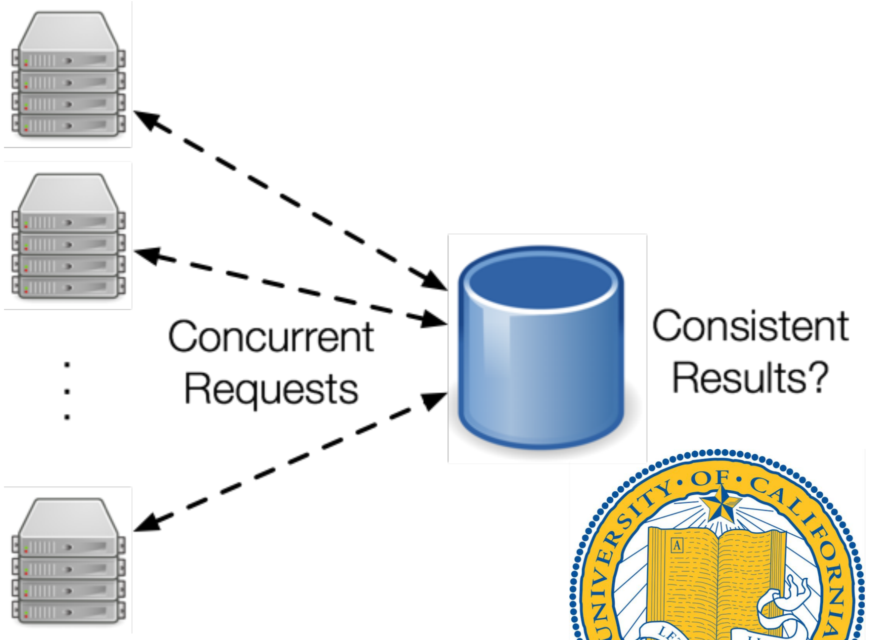
DNS lookup
DNS lookup
TCP SYN
TCP SYN ACK
TCP ACK, HTTP REQUEST
HTTP RESPONSE
Assets via CDN
Third party JS, iframes
TCP SYN
TCP SYN ACK
TCP ACK, HTTP REQUEST
HTTP RESPONSE
Persistent TCP
Persistent TCP
Persistent TCP

A Stable Data Layer - Transactions

We have many application servers running in parallel.

Each needs to persist data that persists between requests.

The prevailing way to do this today is with Relational Databases

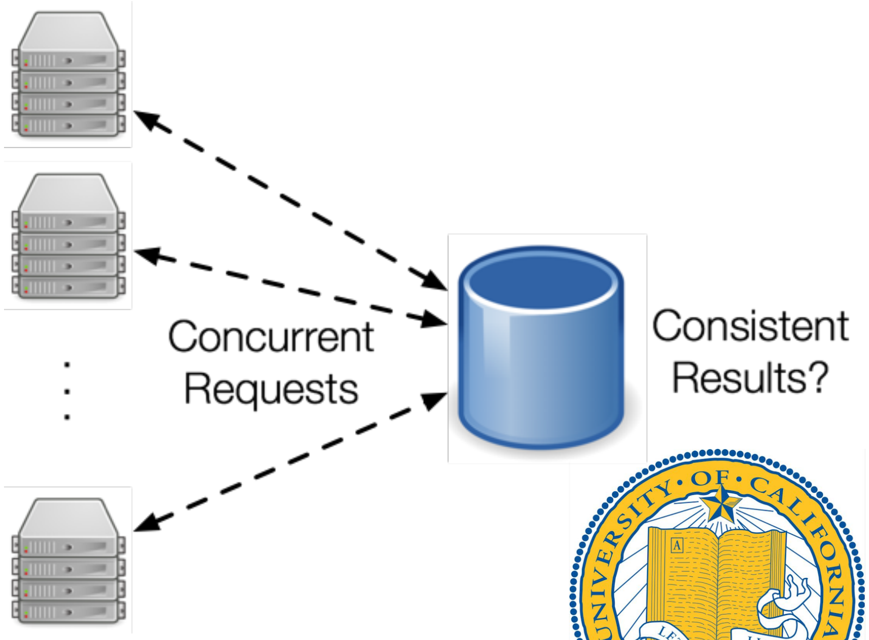


A Stable Data Layer - Transactions

These application servers have needs

- Data needs to be seen by other requests/servers
- Access shouldn't be slow
- Data layer should make sense:

```
david.withdrawal(100)
mary.deposit(100)
```



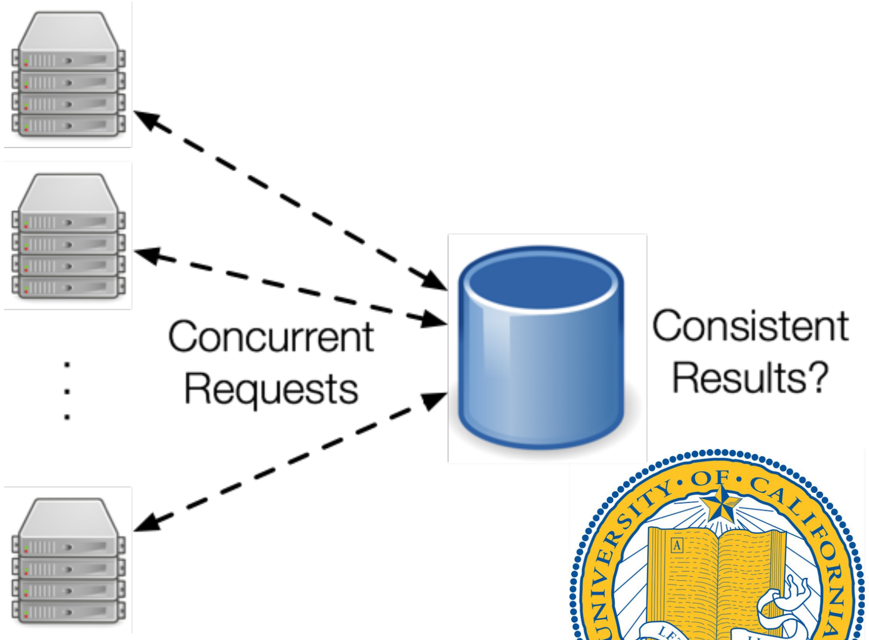
A Stable Data Layer - Transactions

Using transactions in Rails is easy:

```
ActiveRecord::Base.transaction do
  david.withdrawal(100)
  mary.deposit(100)
end
```

Today we will learn more about how these transactions are implemented.

Next time we will learn about transactions in practice



A Stable Data Layer

Database Transactions

- Background
  - Concept that allows a system to guarantee certain semantic properties. Gives control over concurrency.
  - Rigorously defined guarantees mean we can build correct systems on top of them.

A Stable Data Layer - Transactions

History of Database Transactions

- Mid 1970’s, IBM System R’s RSS
  - First system to implement SQL
  - Introduced formal notions of transactions and serializability
  - Led by Jim Gray
    - Result: 1998 Turing Award.

A Stable Data Layer - Transactions

ACID properties in a database:

- Atomicity
  - All or nothing.
  - No partial application of a transaction.

- Consistency
  - At the beginning and at the end of the transaction, the database should be consistent.
  - Consistency is defined by the integrity constraints

A Stable Data Layer - Transactions

ACID properties in a database:

- Isolation
  - A transaction should not see the effects of other uncommitted transactions.
- Durability
  - Once committed, the transaction's effects should not disappear. (being overwritten by later transactions is fine)

A Stable Data Layer - Transactions

These have overlapping concerns

- Atomicity and Durability are related and are generally provided by journalling
- Consistency and Isolation are provided by concurrency control (usually implemented via locking)

No help with side-effects

- Actions that are visible outside of the system
- Transfer money, communicate with web service, etc.

A Stable Data Layer - Transactions

Schedule (or “history”):

- Abstract model used to describe execution of transactions running in the system.

T1	R(X), W(X), Com.		
T2		R(Y), W(Y), Com.	
T3			R(Z), W(Z), Com.

A Stable Data Layer - Transactions

Conflicting Actions:

- Two actions are said to be in conflict if
  - The actions belong to different transactions
  - At least one of the actions is a write operation
  - The actions access the same object (read or write)
- Example of conflicting actions:
  - `T1: R(X), T2: W(X), T3: W(X)`
- And these are not conflicting:
  - `T1: R(X), T2: R(X), T3: R(X)`
  - `T1: R(X), T2: W(Y), T3: R(X)`

Conflict => we can't blindly execute them in parallel.

A Stable Data Layer - Transactions

Why can't we blindly execute them in parallel? Example:

Lost Update Problem

2nd transaction writes a value on top of a 1st transaction and the value is lost to other transactions running concurrently with the 1st transaction. Concurrent read transactions will have incorrect results.

T1	R(X)	R(X)	R(X)...
T2		W(X)	Com.

A Stable Data Layer - Transactions

Why can't we blindly execute them in parallel? Example:

Dirty Read Problem

Transactions read a value written by a transaction that is later aborted and removed from the database. Reading transactions will have incorrect results.

T1		R(X)	W(Y), Com.
T2	W(X)		Abort

A Stable Data Layer - Transactions

Why can't we blindly execute them in parallel? Example:

Incorrect Summary Problem

1st transaction takes a summary over the values of all the instances of a repeated data item. While a 2nd transaction updates some instances of the data item. Resulting summary will not reflect a correct result for any deterministic order of the transactions. Result will be random depending on the timing of the updates.

T1	R(All X), AVG	W(Y)	Com.
T2	W(Some X), Com.		

A Stable Data Layer - Transactions

A schedule is **serial** if

- The transactions are executed non-interleaved

Two schedules are **conflict equivalent** if

- They involve the same actions of the same transactions
- Every pair of conflicting actions is ordered in the same way

Schedule S is **conflict serializable** if

- S is **conflict equivalent** to some serial schedule

A schedule is **recoverable** if

- Transactions commit only after all transactions whose changes they read, commit.

A Stable Data Layer - Transactions

Example of a schedule that is not **conflict serializable**:

T1	R(A), W(A)		R(B), W(B)
T2		R(A), W(A), R(B), W(B)	

Because it is not **conflict equivalent** to this:

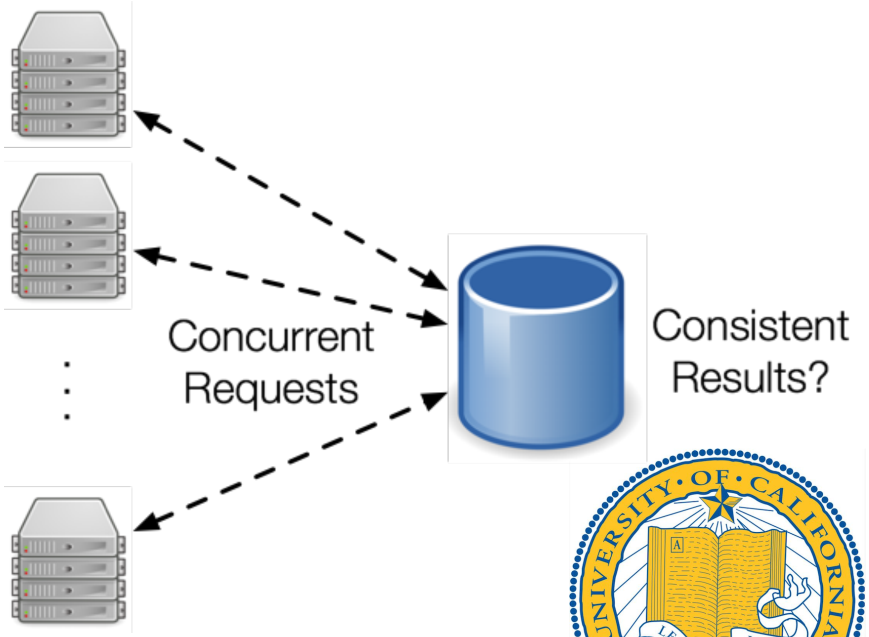
T1	R(A), W(A), R(B), W(B)	
T2		R(A), W(A), R(B), W(B)

or this:

T1		R(A), W(A), R(B), W(B)
T2	R(A), W(A), R(B), W(B)	

A Stable Data Layer - Transactions

Why is it important that we get a serializable schedule?



A Stable Data Layer - Transactions

Otherwise you can get inconsistent results - not good when you are keeping track of your bank balance in the database

A serial execution of transactions is safe but slow

Most general purpose relational databases default to employing conflict-serializable and recoverable schedules

If you don't want to do a serial execution, what else can you do?

A Stable Data Layer - Transactions

How do we implement a database that schedules that are conflict serializable and recoverable?

Locks

- A lock is a system object associated with a shared resource such as a data item, a row, or a page in memory
- A database lock may need to be acquired by a transaction before accessing the object
- Prevent undesired, incorrect, or inconsistent operations on shared resources by concurrent transactions

A Stable Data Layer - Transactions

Two types of database locks:

- Write-lock
  - Blocks writes and reads
  - Also called “exclusive lock”
- Read-lock
  - Blocks writes
  - Also called “shared lock”

A Stable Data Layer - Transactions

Two-Phase Locking

- 2PL is a concurrency control method that guarantees serializability
- Two-Phase Locking Protocol
  - Each Transaction must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing
    - If a Transaction holds an X lock on an object, no other Transaction can get a lock (S or X) on that object
  - A transaction cannot request additional locks once it releases any locks
  - Two phases: acquire locks, release locks
- Issue: can result in “cascading aborts”

```
T1: R(A) W(A) unlock(A) ......... abort
T2:                 R(A) .......... abort
```

A Stable Data Layer - Transactions

Strong Strict Two-Phase Locking

- SS2PL allows only conflict serializable schedules
- SS2PL Protocol
  - Each Transaction must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing
    - If a Transaction holds an X lock on an object, no other Transaction can get a lock (S or X) on that object
  - All locks held by a transaction are released when the transaction completes
- Avoids cascading aborts

Today's Agenda

Up Next: User Authentication with Devise

For Next Time…

- Code, code, code!