No C-QL

(Or how I learned to stop worrying, and love eventual consistency)

Brian Brazil
Senior Software Engineer
Boxever

Overview

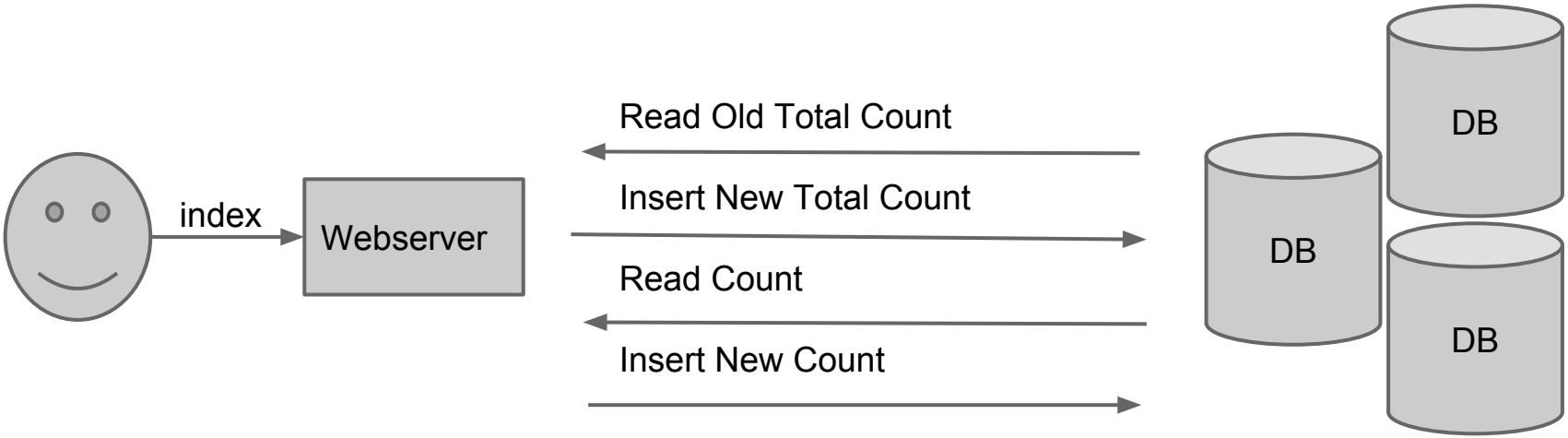
NoSQL and eventual consistency are complementary.

Why would you need it, how do you take advantage of it, and when is it useful in distributed systems?

A Simple Example

Website that counts how many times a user visits the site, and each page.

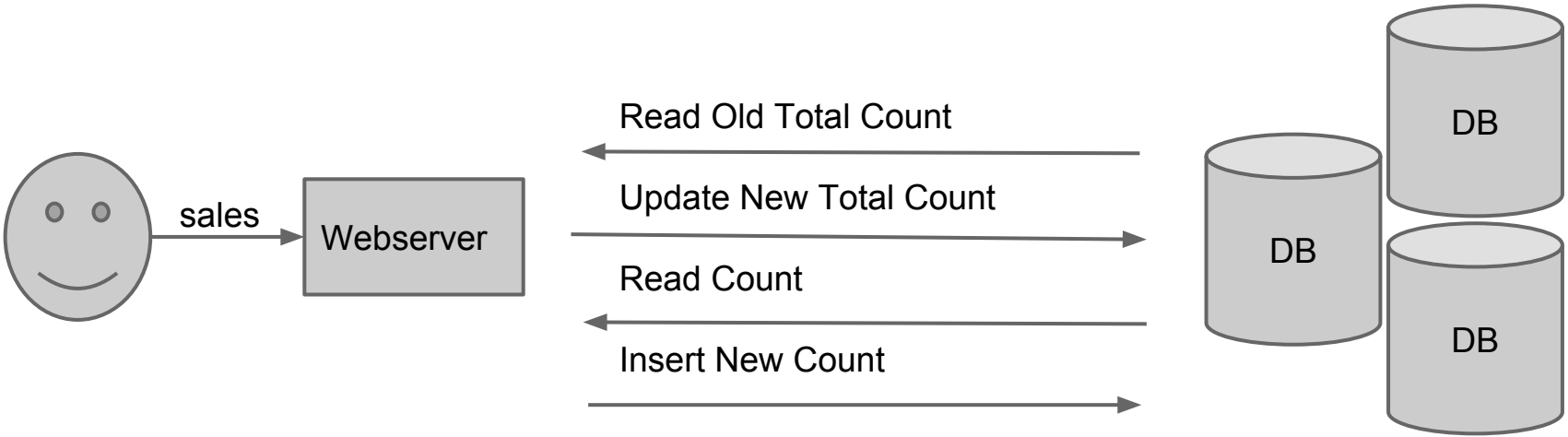
First visit



User	Total Count
Brian	1

User	Page	Count
Brian	index	1

Second visit



User	Total Count
Brian	2

User	Page	Count
Brian	index	1
Brian	sales	1

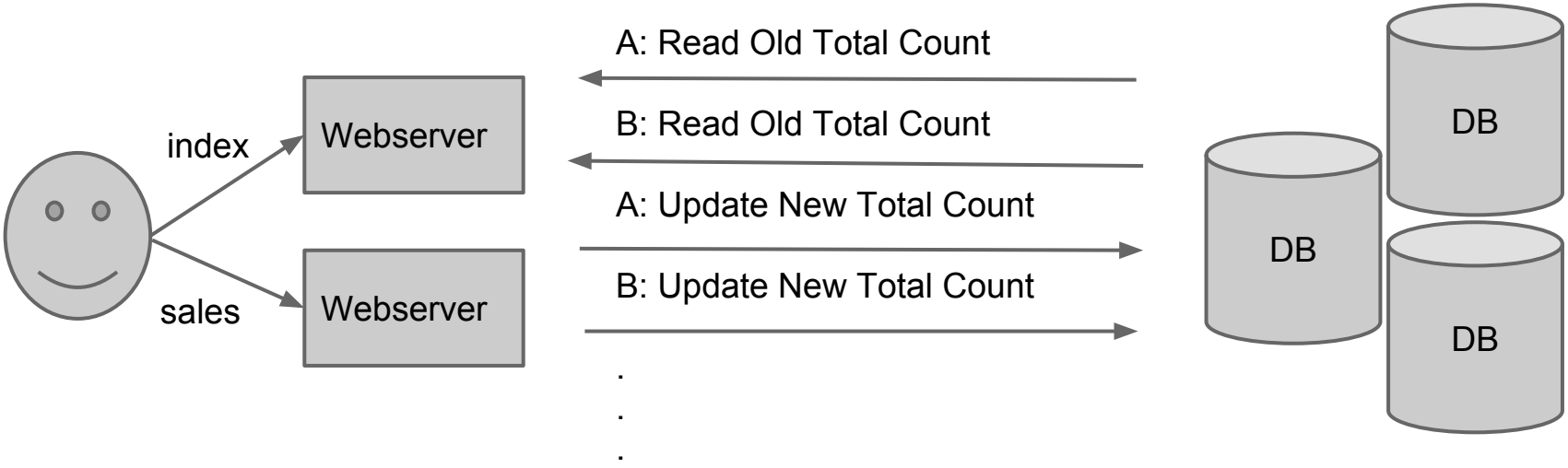
What do you think?

Initial thoughts

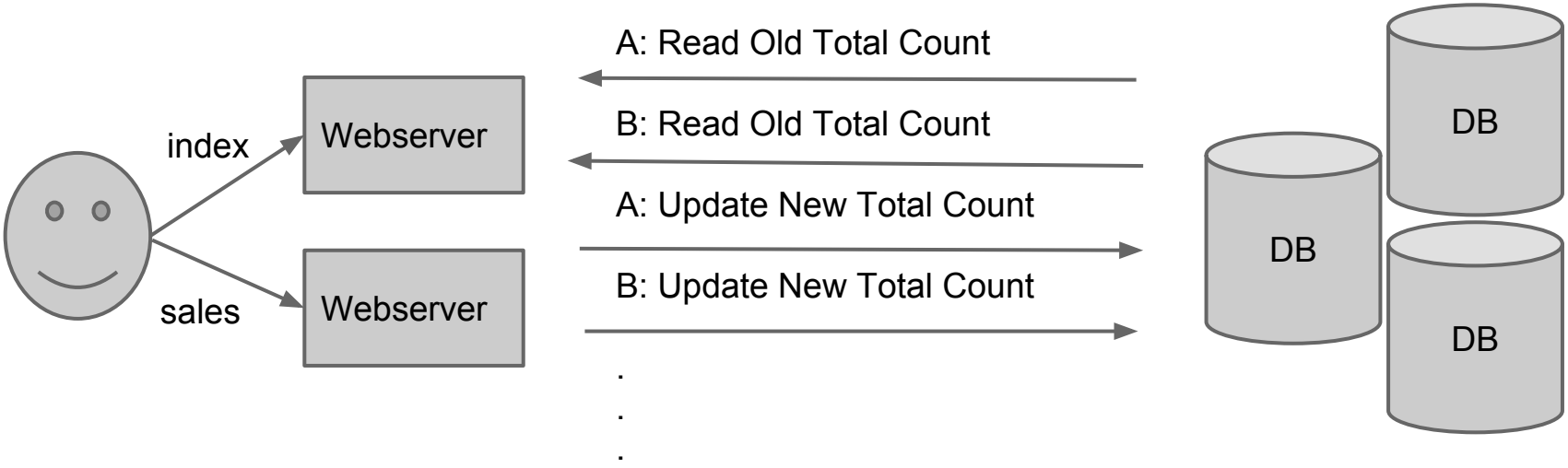
Looks like a simple, obvious solution

Tabbed Browsing

Tabbed Browsing



Tabbed Browsing



User	Total Count
Brian	3

User	Page	Count
Brian	index	2
Brian	sales	2

What happened?

Concurrent updates clashed.

Need to protect against that.

ACID to the rescue!

- Apply each set of updates atomically
- Updates don’t interfere with each other

ACID to the rescue?

- Bottleneck for writes
- Webserver needs to retry
- Databases need to coordinate for quorum

Aside: Cost of Reads

- Hard disk seek: 5 - 10ms

Aside: Cost of Reads

- Hard disk seek: 5 - 10ms
- Cluster quorum read: 1 - 200ms

Aside: Cost of Reads

- Hard disk seek: 5 - 10ms
- Cluster quorum read: 1 - 200ms
- RTT to webserver: 1 - 200ms

Aside: Cost of Reads

- Hard disk seek: 5 - 10ms
- Cluster quorum read: 1 - 200ms
- RTT to webserver: 1 - 200ms

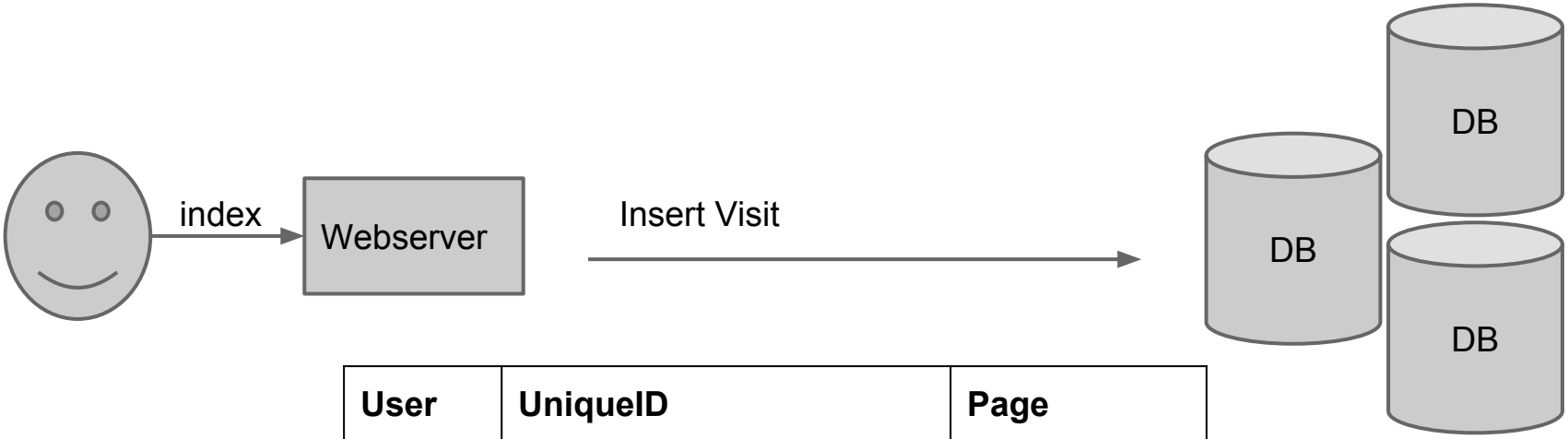
Avoid reads on the write path.

Eventual consistency

Being slightly out of date doesn’t matter for many applications.

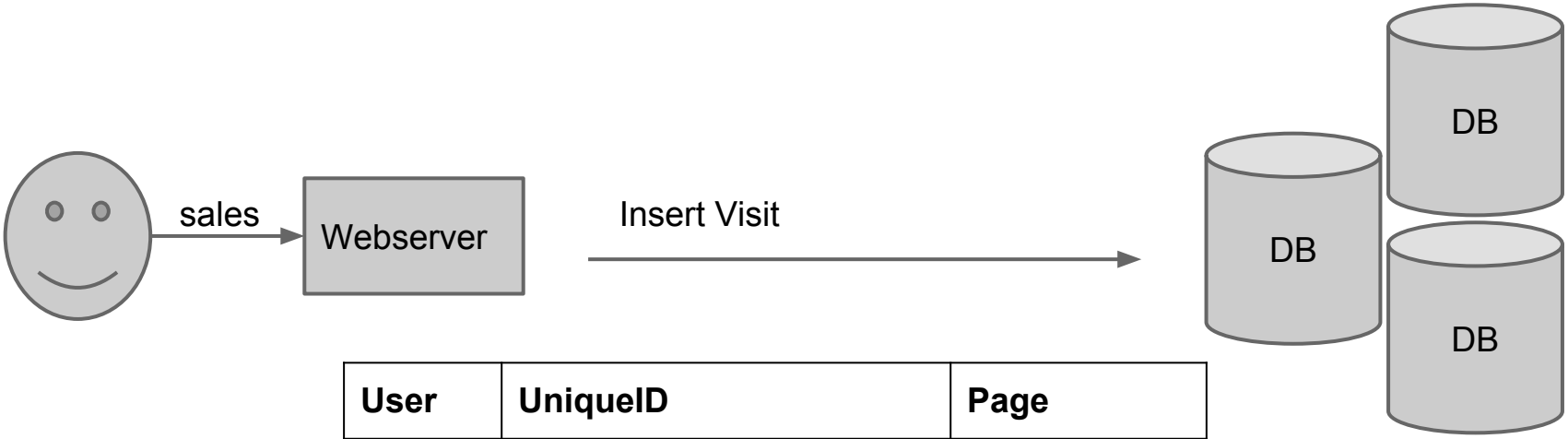
Can we take advantage of this to avoid bottlenecks and latency?

First visit



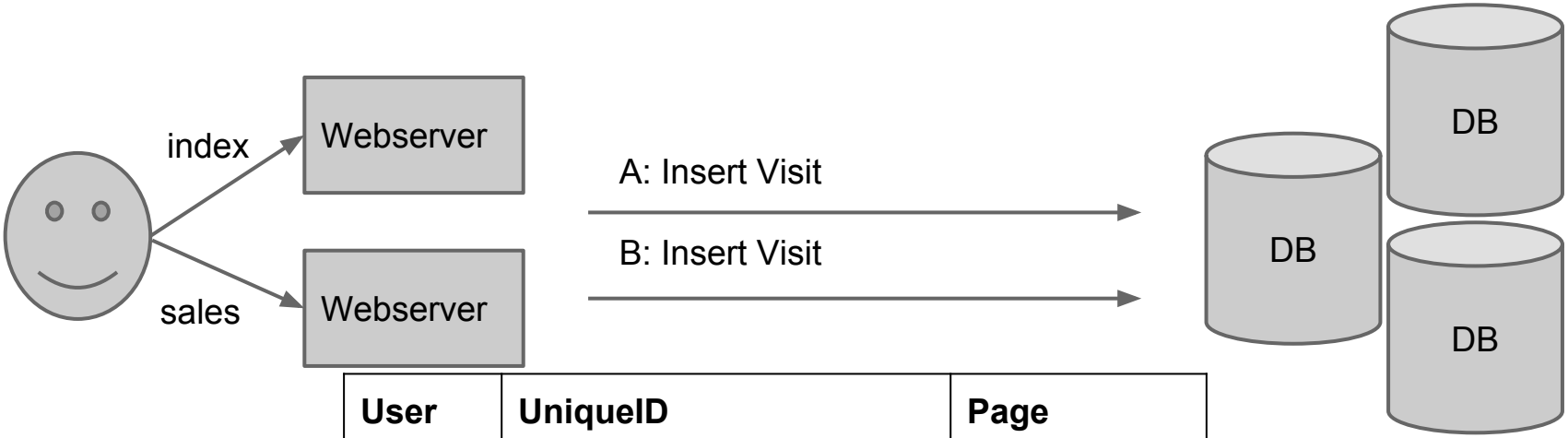
User	UniqueID	Page
Brian	1409778671-Server1-PID	index

Second visit



User	UniqueID	Page
Brian	1409778671-Server1-PID	index
Brian	1409778689-Server1-PID	sales

Tabbed Browsing



User	UniqueID	Page
Brian	1409778671-Server1-PID	index
Brian	1409778689-Server1-PID	sales
Brian	1409778691-Server1-PID	index
Brian	1409778691-Server2-PID	sales

Analysis

- Unique IDs avoid clashes
- No bottlenecks due to consistency
- Data always internally consistent
- Fire & forget
- Only need to talk to nearest database(s)
- Resilient to network partitions

Not all good

- Reads have to look at more data
- Need to make sure data makes it everywhere
- What if something depends on having data up to time X?

NoSQL?

Everything thus far applies equally to relational and non-relational databases.

NoSQL Advantage

- Data locality on disk, multiple entries per row
- Replication options beyond master/slave
- Writes are cheap
- Can also have consistency where needed

When is it good?

- Writes dominate
- Need high throughput and scalability
- Inter/intra continental databases
- 100% correct data in realtime not essential

When is it not so good?

- Writes are relatively rare
- Scale isn’t a concern
- Data must be completely consistent

Idempotency

Each entry is unique, so can be safely replayed again and again.

Gives you options for disaster recovery:

Log requests to local webserver disk, replay if database goes down

Improvements

- Take advantage of entry ordering
- Do rollups as a batch task
  - Reduces data to be processed, cutting latency
- Row per day/week/month
  - Avoids rows getting too big
- Handling updates and deletes
  - Can be done using only puts

Advanced

This is a very simple example, only doing a count.

For more see

“Extreme availability and self-healing data with CRDTs” at 11am with Uwe Friedrichsen in Room 2

Summary

Consistency brings bottlenecks.

Eventual consistency allows for high-throughput, reliable writes.

NoSQL combined with eventual consistency can scale very well.

Questions?