How to survive in a BASE world

A practical guide for the dauntless developer

Uwe Friedrichsen (codecentric AG) – NoSQL matters, Barcelona – 30. November 2013

@ufried

Uwe Friedrichsen | uwe.friedrichsen@codecentric.de | http://slideshare.net/ufried | http://ufried.tumblr.com

ACID vs. BASE

A C I D

Atomicity Consistency Isolation Durability

BASE

Basically Available Soft State Eventually Consistent

ACID vs. BASE

ACID

- Strong consistency
- Isolation
- Focus on "commit"
- Nested transactions
- Availability?
- Conservative (pessimistic)
- Difficult evolution (e.g. schema)

BASE

- Weak consistency (stale data OK)
- Availability first
- Best effort
- Approximate answers OK
- Aggressive (optimistic)
- Simpler!
- Faster
- Easier evolution

← But I think it's a spectrum →

Source: Eric Brewer, Towards Robust Distributed Systems, PODC Keynote, July-19-2000

Consequences

- No ACID properties across entities
- No ACID properties across nodes
- ACID properties for single entity on single node

A (very) simple example

Data model

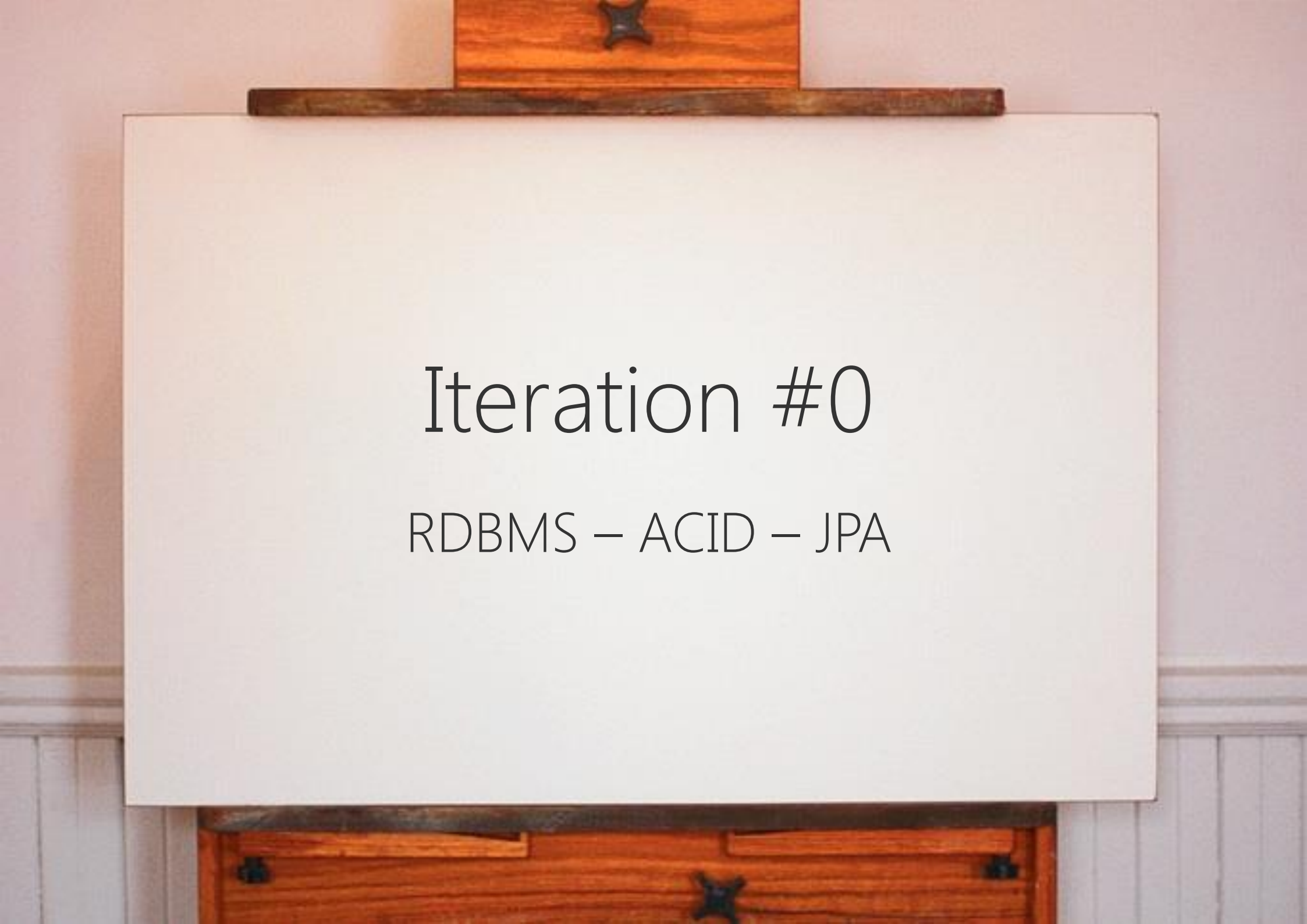
Actions

loan(customer, book) : void

return(customer, book) : void

booksLoaned(customer) : list of book

isBookLoaned(book) : boolean


Iteration #0
RDBMS – ACID – JPA

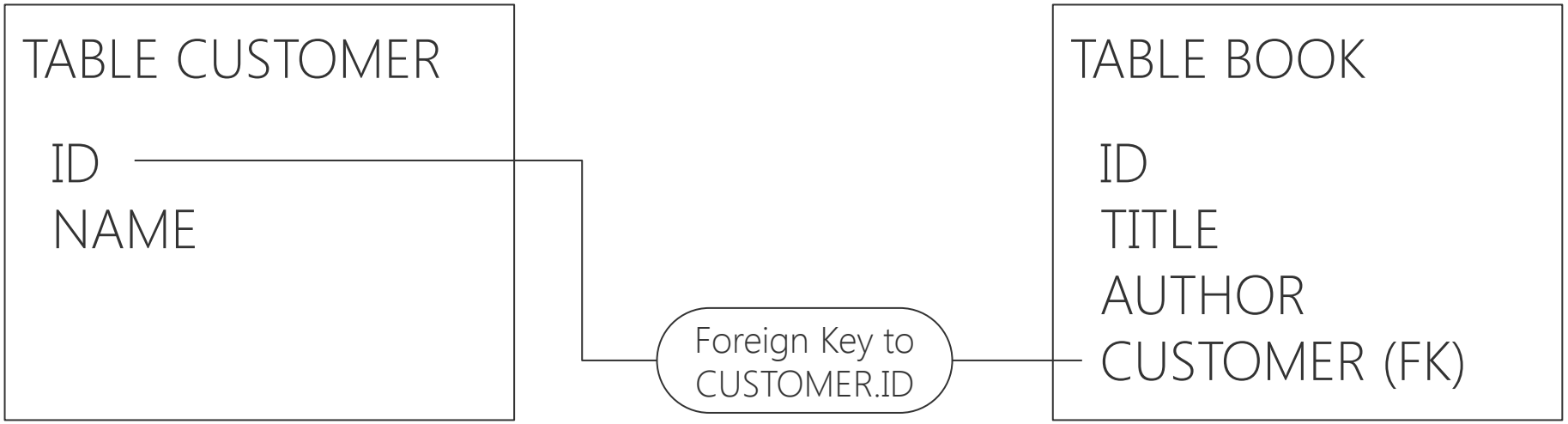
JPA Entity Class Customer

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    @OneToMany(mappedBy = "customer")
    private final List<Book> loanedBooks;

    <Methods>
}
```

JPA Entity Class Book

```
@Entity
public class Book {
    @Id
    @GeneratedValue
    private long id;
    private String title;
    private String author;
    private Customer customer;

    <Methods>
}
```

Database model



Action „loan“ using JPA

```
public void loan(long customerId, long bookId) {
    <Get EntityManager>
    em.getTransaction().begin();

    <Read customer and book from database>

    customer.getLoanedBooks().add(book);
    book.setCustomer(customer);

    em.persist(customer);
    em.persist(book);
    em.getTransaction().commit();

    <Cleanup>
}
```

Same for action „return“

Action „isBookLoaned“ using JPA

```
public boolean isBookLoaned(long bookId) {
    <Get EntityManager>

    Query q = em.createQuery("SELECT b FROM Book b WHERE b.id = :id");
    q.setParameter("id", bookId);
    Book book = (Book) q.getSingleResult();

    return book.getCustomer() != null;
}
```

Action „booksLoaned“ using JPA

```
public List<Book> booksLoaned(long customerId) {
    <Get EntityManager>

    Query q =
        em.createQuery("SELECT c FROM Customer c WHERE c.id = :id");
    q.setParameter("id", customerId);
    Customer customer = (Customer) q.getSingleResult();

    return customer.getLoanedBooks();
}
```

Boils down to an implicitly executed SQL query that looks like

```
SELECT … FROM BOOK WHERE CUSTOMERID = <customerId>
```

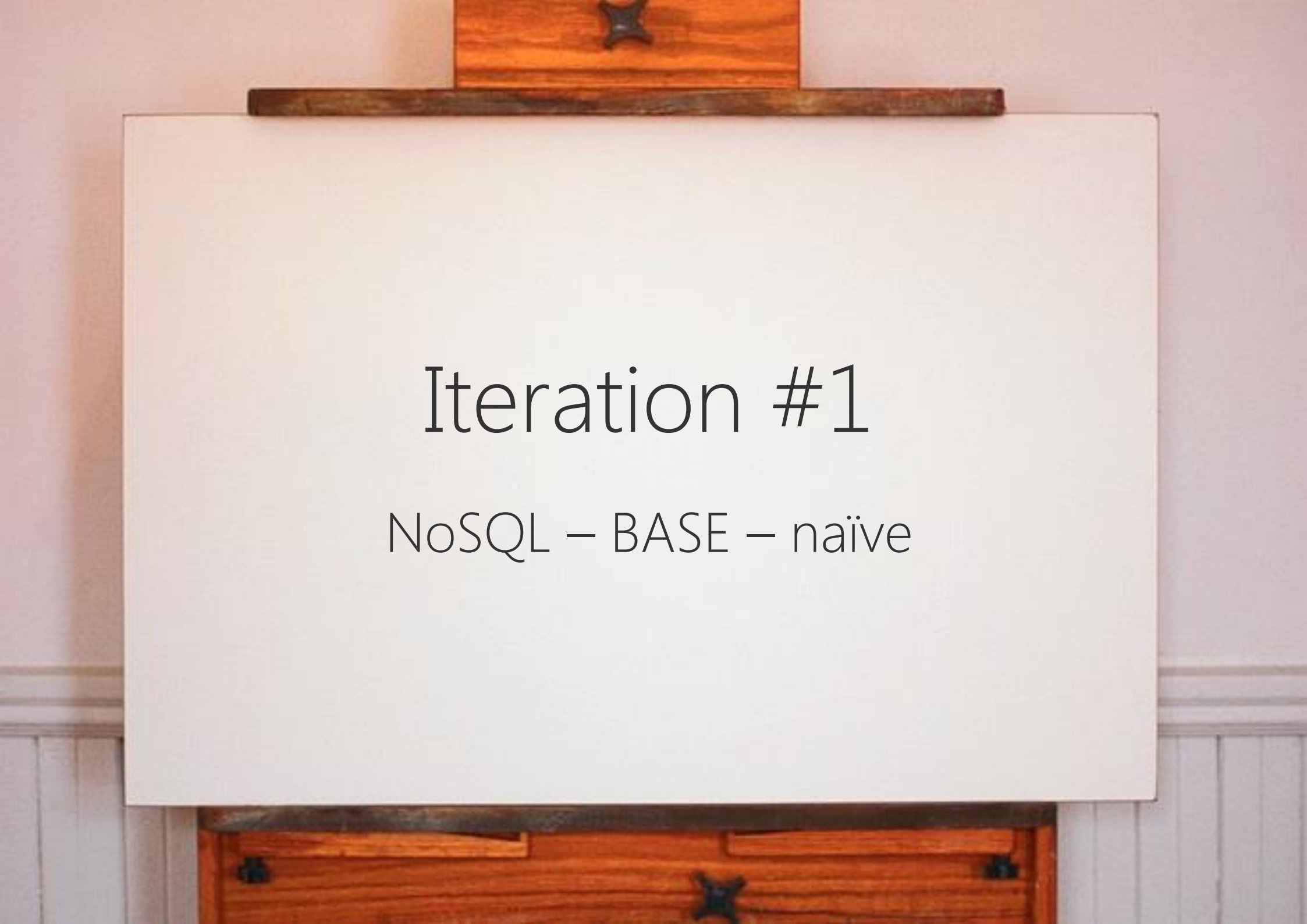
Wrapup ACID model

Mutating action

- begin transaction
- do required changes
- commit

Non-mutating action

- create query
- execute query
- navigate in result set
  (implicitly executing additional queries if required)


Iteration #1
NoSQL – BASE – naïve

Naïve BASE model

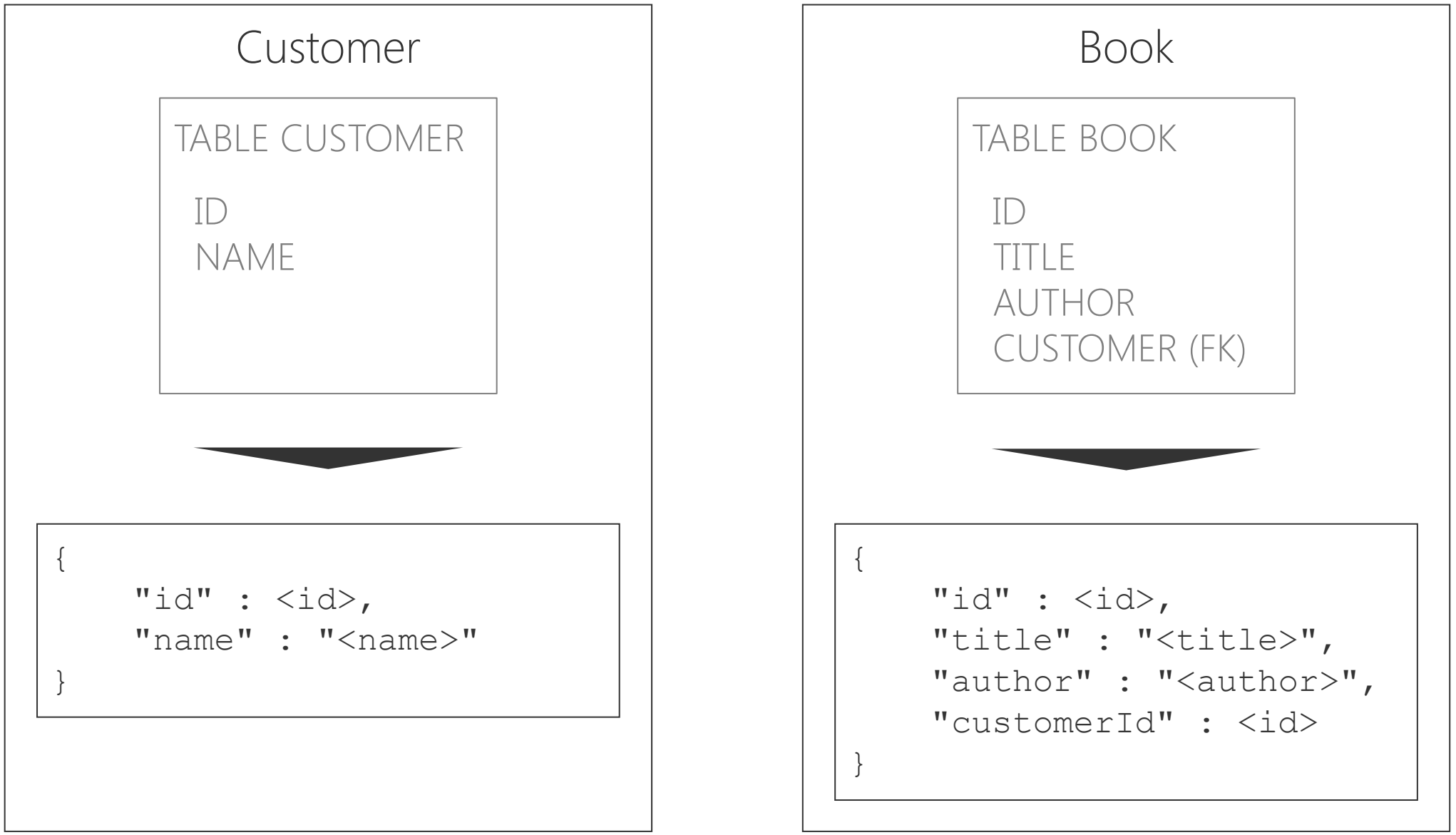
Mutating action

- ~~begin transaction~~
- do required changes
- ~~commit~~

Non-mutating action

- create query
- execute query
- ~~navigate in result set~~ ???
  (implicitly executing additional queries if required)

Data model


Customer
TABLE CUSTOMER
ID
NAME
{
"id" : <id>,
"name" : "<name>"
}
Book
TABLE BOOK
ID
TITLE
AUTHOR
CUSTOMER (FK)
{
"id" : <id>,
"title" : "<title>",
"author" : "<author>",
"customerId" : <id>
}

JPA Entity Class Customer

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
     private long id;
     private String name;
    @OneToMany(mappedBy = "customer")
    private final List<Book> loanedBooks;

     <Methods>
}
```

Need to take care of ID generation ourselves

Gone for the moment (naïve approach)

Class Customer

```
public class Customer {
    private long id;
    private String name;

    <Methods>
}
```

JPA Entity Class Book

```
@Entity
public class Book {
    @Id
    @GeneratedValue
     private long id;
     private String title;
     private String author;
     private Customer customer;

     <Methods>
}
```

Need to take care of ID generation ourselves

Gone for the moment (naïve approach)

Class Book

```
public class Book {
    private long id;
    private String title;
    private String author;
    private long customerId;

    <Methods>
}
```

Only the ID

Action „loan“, „return“ & „isBookLoaned“

```
public void loan(long customerId, long bookId) {
    Book book = readBook(bookId);
    book.setCustomerId(customerId);
    writeBook(book);
}

public void return(long customerId, long bookId) {
    Book book = readBook(bookId);
    book.setCustomerId(0L);
    writeBook(book);
}

public boolean isBookLoaned(long bookId) {
    Book book = readBook(bookId);
    return book.getCustomerId() > 0L;
}
```

readBook

```
private Book readBook(long id) {
    String json = readBookFromDb(id);
    return toBook(json);
}

private Book toBook(String json) {
    JSONObject jo = new JSONObject(json);
    Book b = new Book(jo.getLong("id"), jo.getString("title"),
        jo.getString("author"), jo.getLong("customerId"));
    return b;
}
```

Same for „readCustomer“ and „toCustomer“

writeBook

```
private void writeBook(Book b) {
    long key = b.getId();
    String json = fromBook(b);
    writeBookToDb(key, json);
}

private String fromBook(Book b) {
    JSONObject jo = new JSONObject();
    jo.put("id", b.getId());
    jo.put("title", c.getTitle());
    jo.put("author", c.getAuthor());
    jo.put("customerId", b.getCustomerId());
    return jo.toString();
}
```

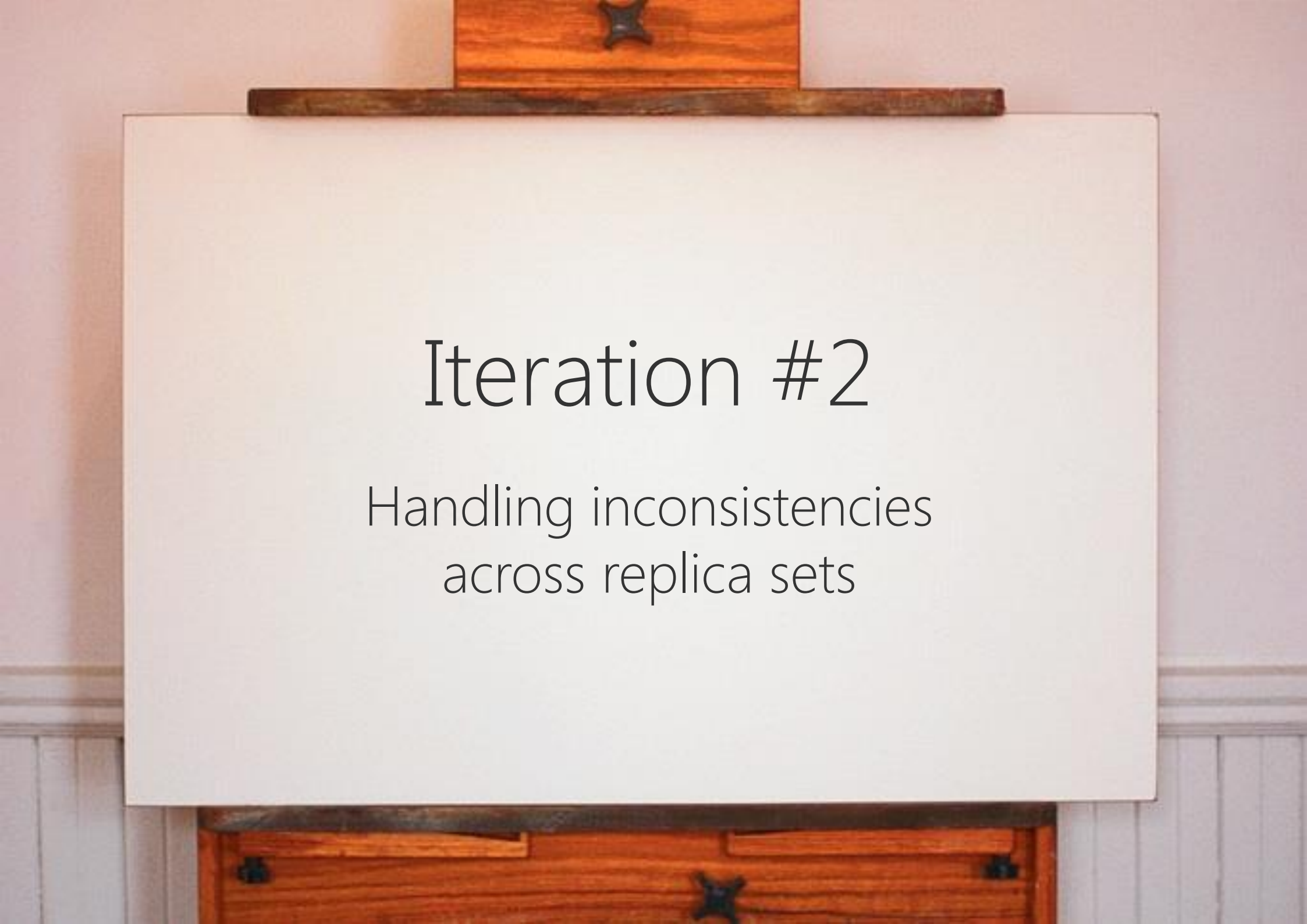
Same for „writeCustomer“ and „fromCustomer“

Action „booksLoaned“

```
public List<Book> booksLoaned(long customerId) {
    List<Book> loanedBooks = new ArrayList<Book>();
    Iterator<Book> it = readAllBooks();
    for (Book b : it) {
        if (book.getCustomerId() == customerId) {
            loanedBooks.add(b);
        }
    }
    return loanedBooks;
}

private Iterator<Book> readAllBooks() {
    DBIterator dbi = readAllBooksFromDb();  // Iterator<String> (json)
    return new BookIterator(dbi);
}
```

BookIterator

```
public class BookIterator implements Iterator<Book> {
    private final DBIterator dbi;

    public BookIterator(DBIterator dbi) {
        this.dbi = dbi;
    }

    @Override public boolean hasNext() {
        return dbi.hasNext();
    }

    @Override public Book next() {
        return toBook(dbi.next());
    }

    @Override public void remove() {
        throw new UnsupportedOperationException();
    }
}
```


Iteration #2
Handling inconsistencies
across replica sets

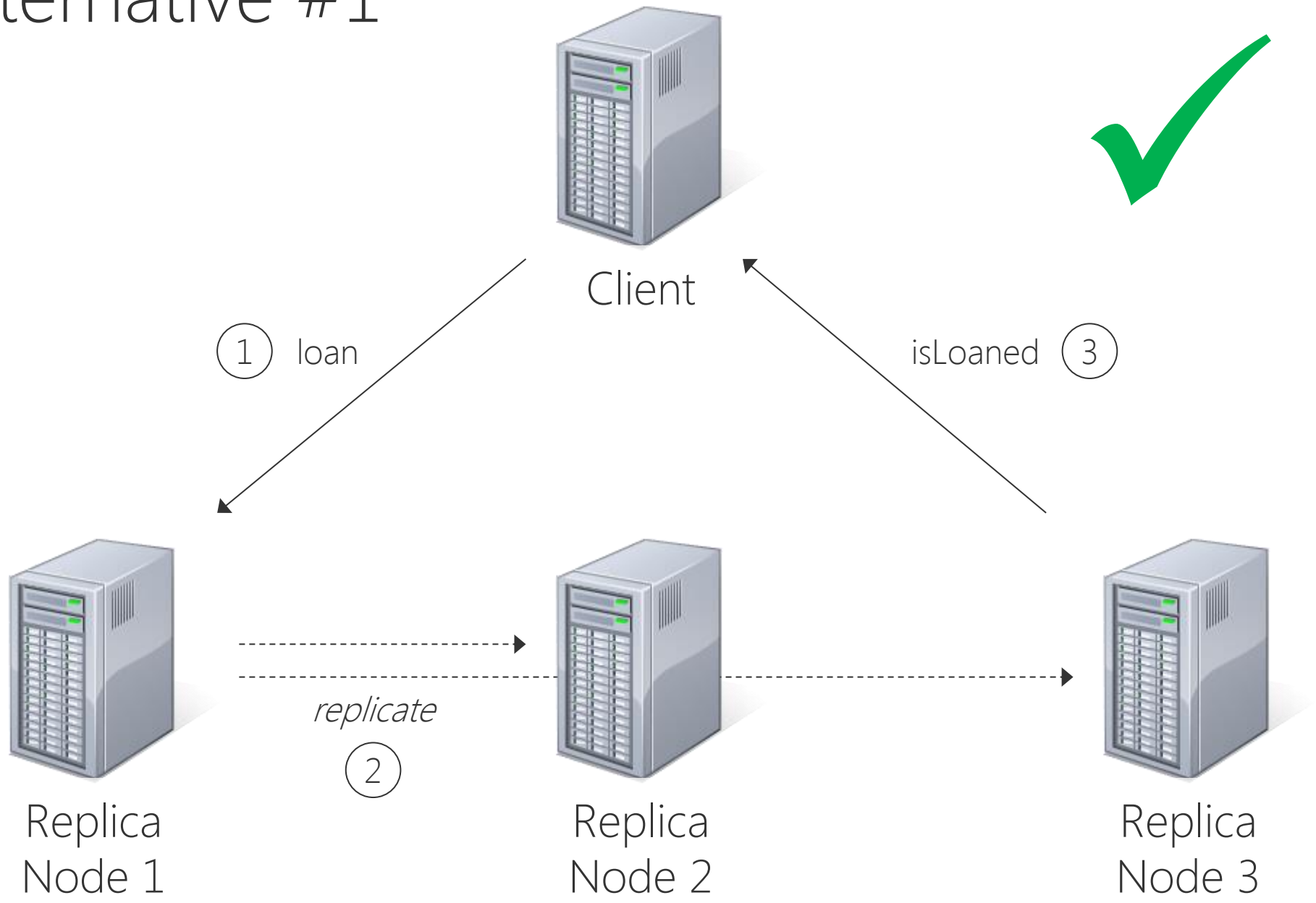
Quick exercise

Assume the following code sequence

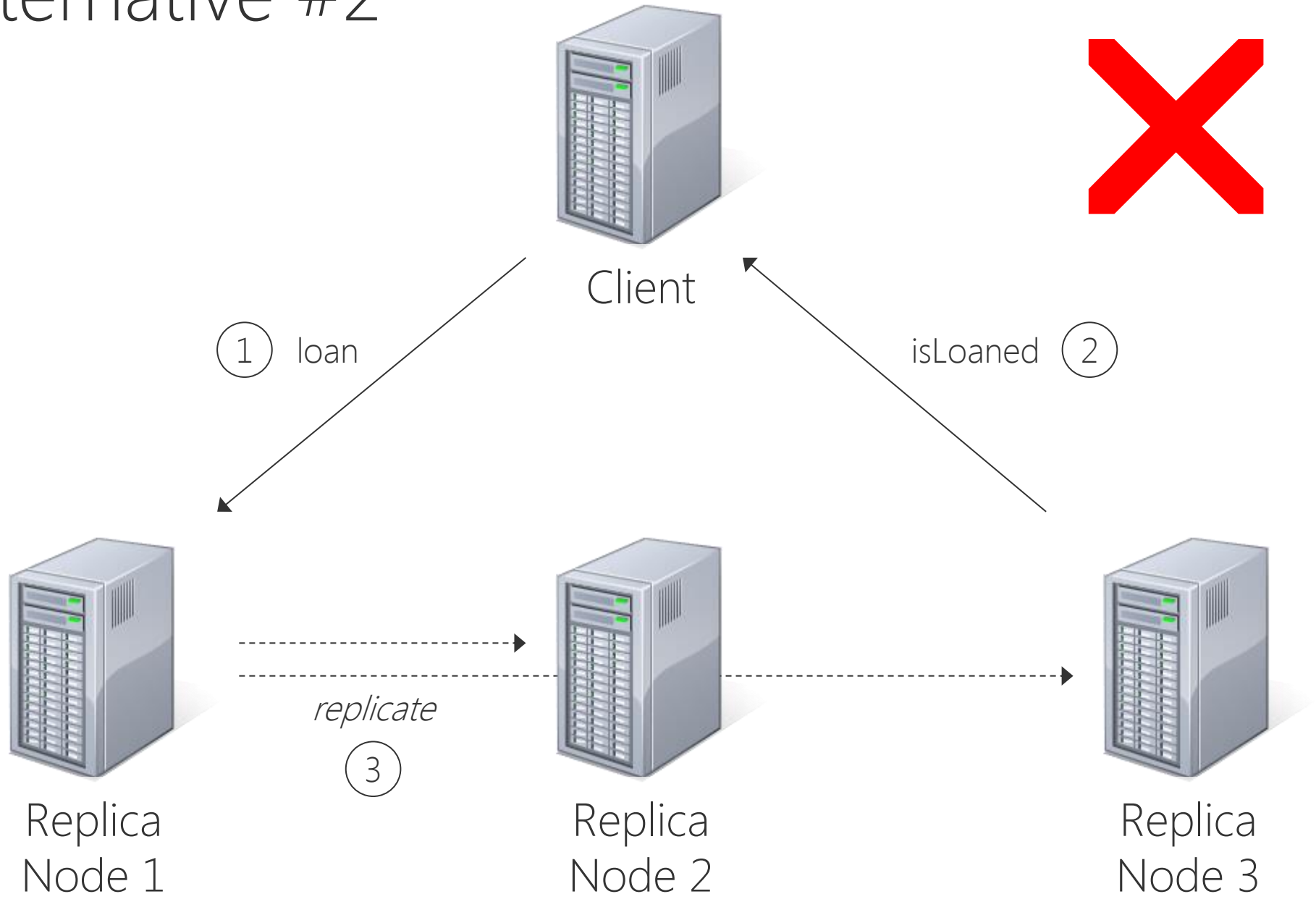
```
loan(2312L, 4711L);
boolean isLoaned = isBookLoaned(4711L);
```

Question: Which value has `isLoaned`?

Alternative #1

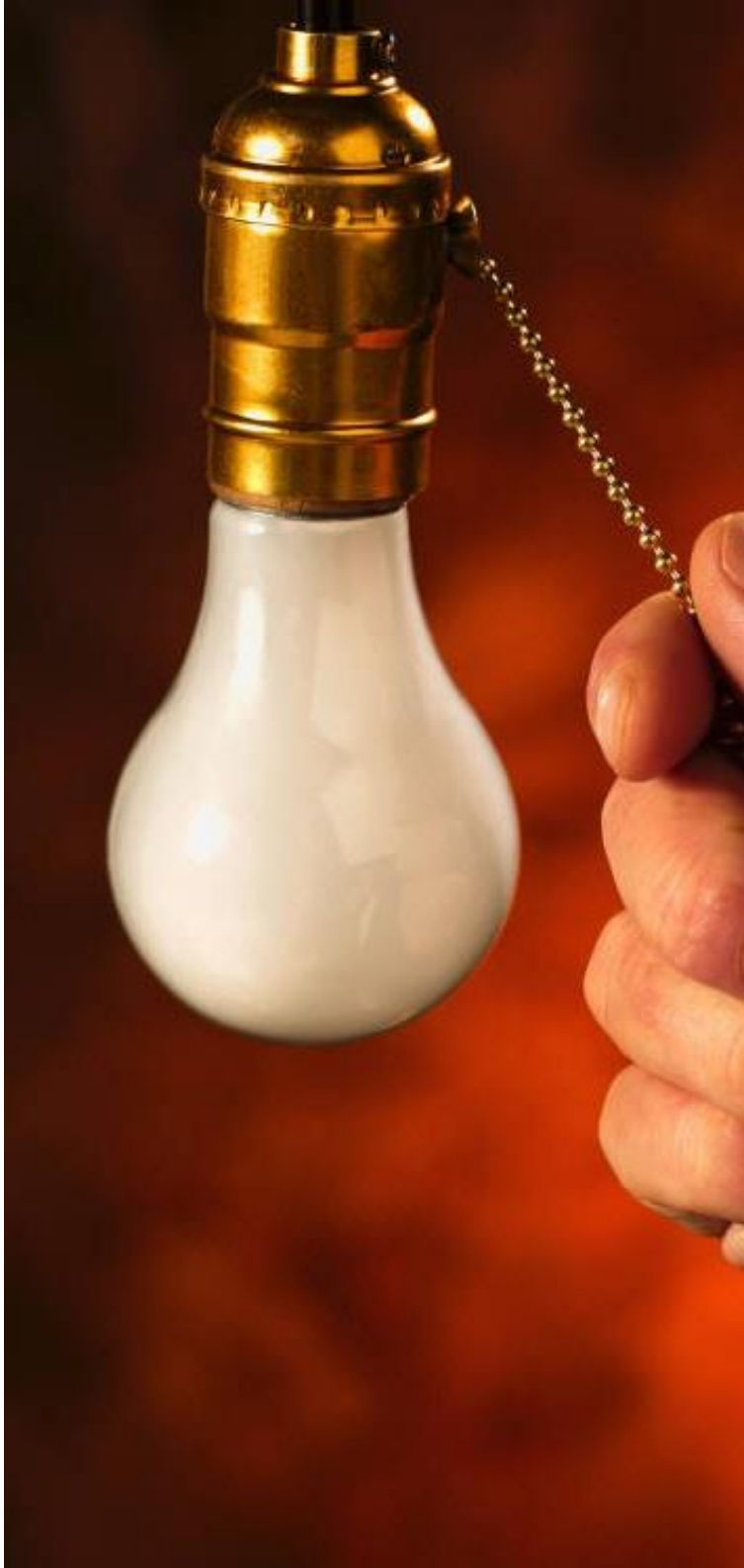


Alternative #2

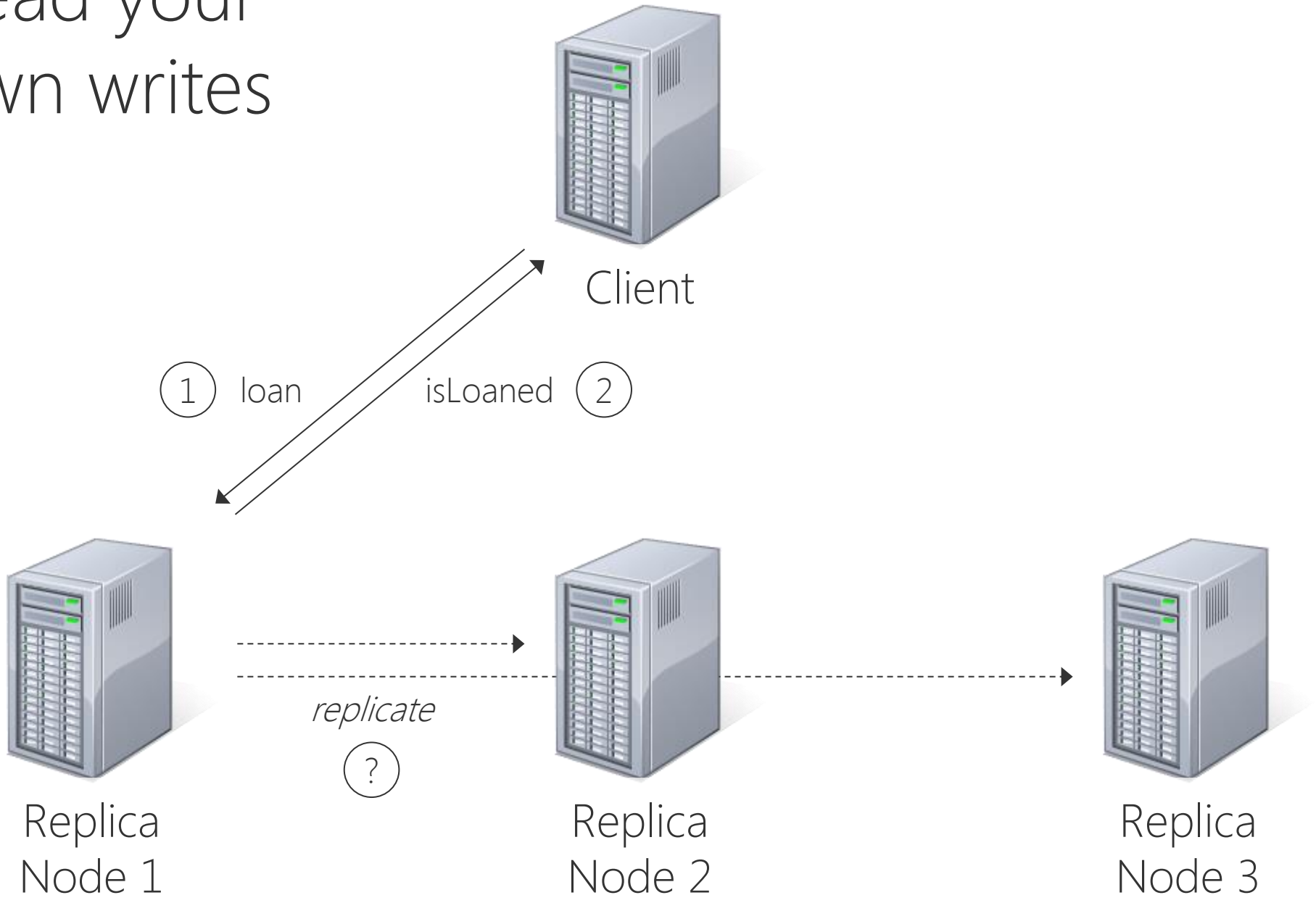


Solution #1

Read your own writes

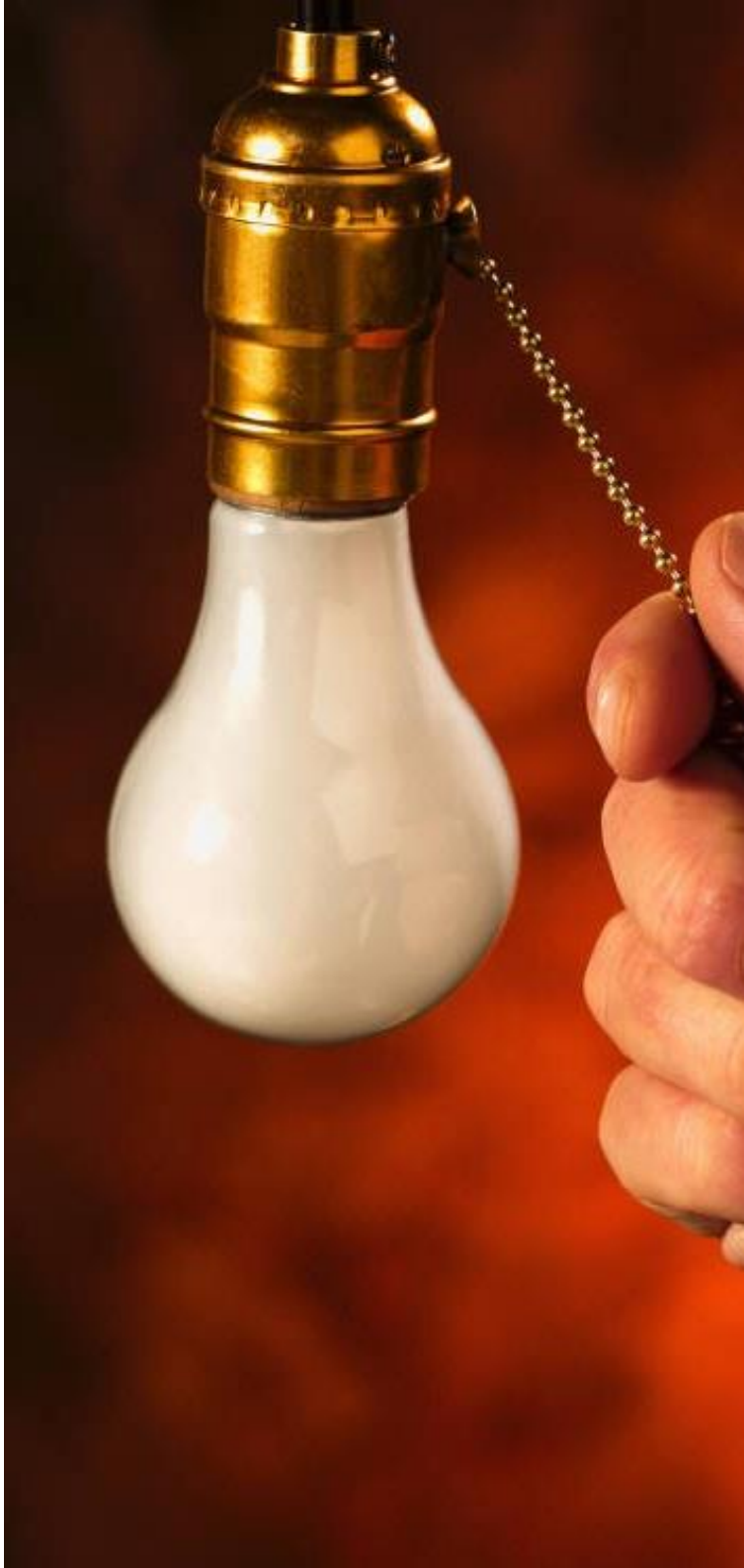


Read your own writes

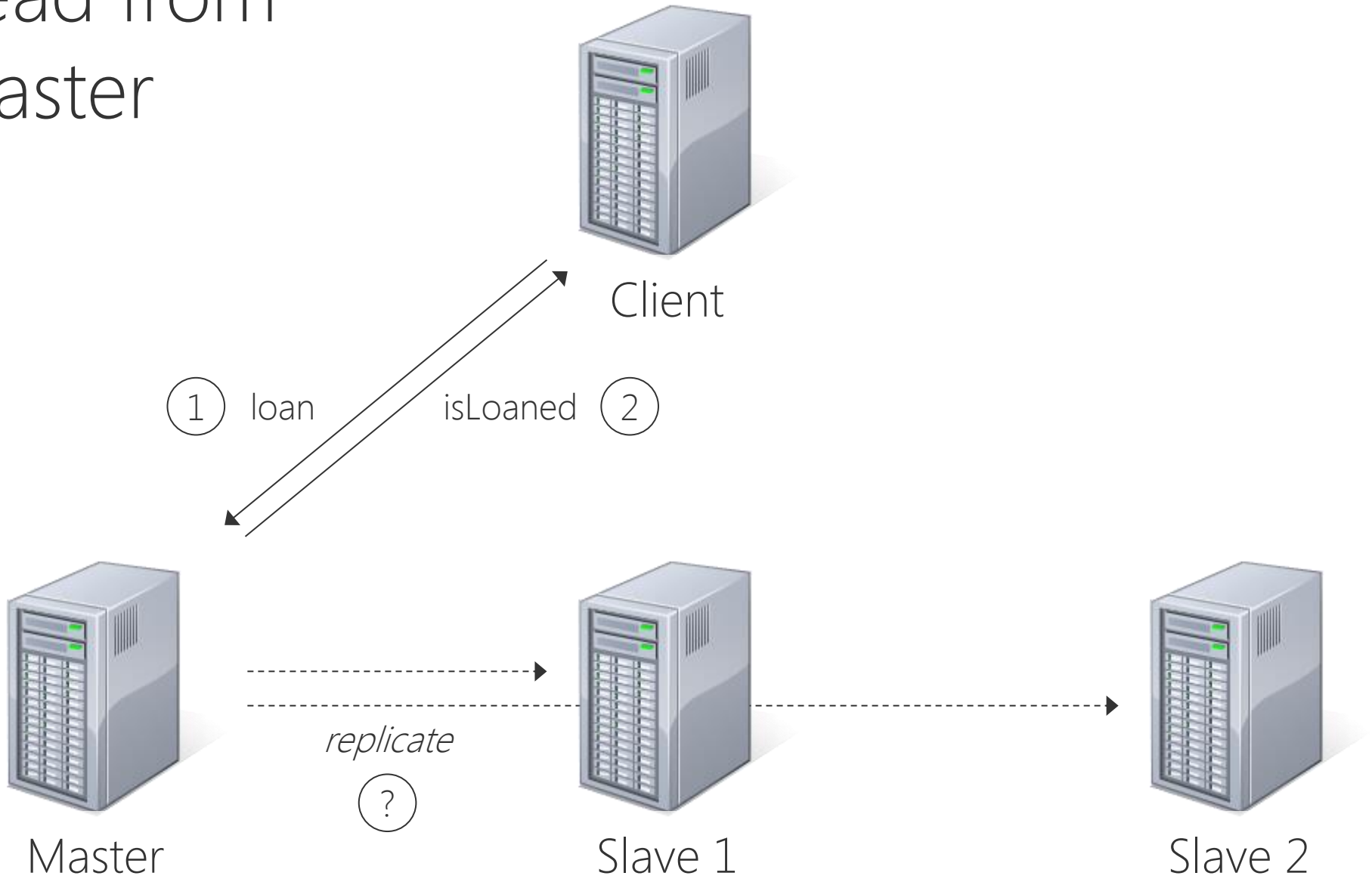


Solution #2

Read from master



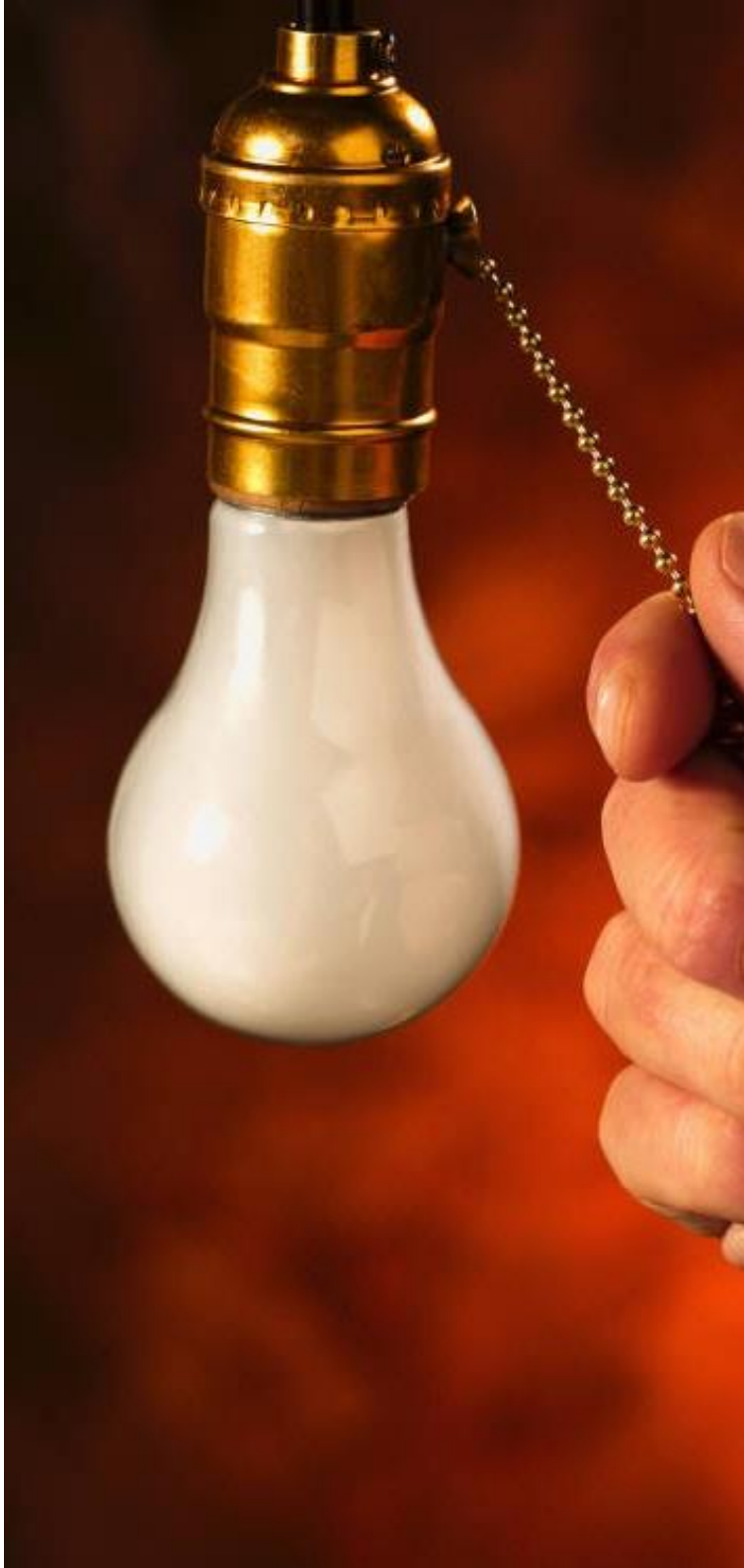
Read from master



Solution #3

Quorum

- W > N/2
- R + W > N



readBook & writeBook

```
private Book readBook(long id) {
    String json = readBookFromDb(id, READ_QUORUM);
    return toBook(json);
}

private void writeBook(Book b) {
    long key = b.getId();
    String json = fromBook(b);
    writeBookToDb(key, json, WRITE_QUORUM);
}
```

Same for „readCustomer“ and „writeCustomer“


Iteration #3
Handling siblings
due to partitioning

Quick exercise

Assume the following code sequence

```
boolean isLoaned = isBookLoaned(4711L);
```

Question: What do you get?

```
isBookLoaned(4711L)
```

```
public boolean isBookLoaned(4711L) {
    Book book = readBook(4711L);
    return book.getCustomerId() > 0L;
}
```

```
private Book readBook(4711L) {
    String json = readBookFromDb(4711L);
    return toBook(json);
}
```

Expects a "normal" book JSON

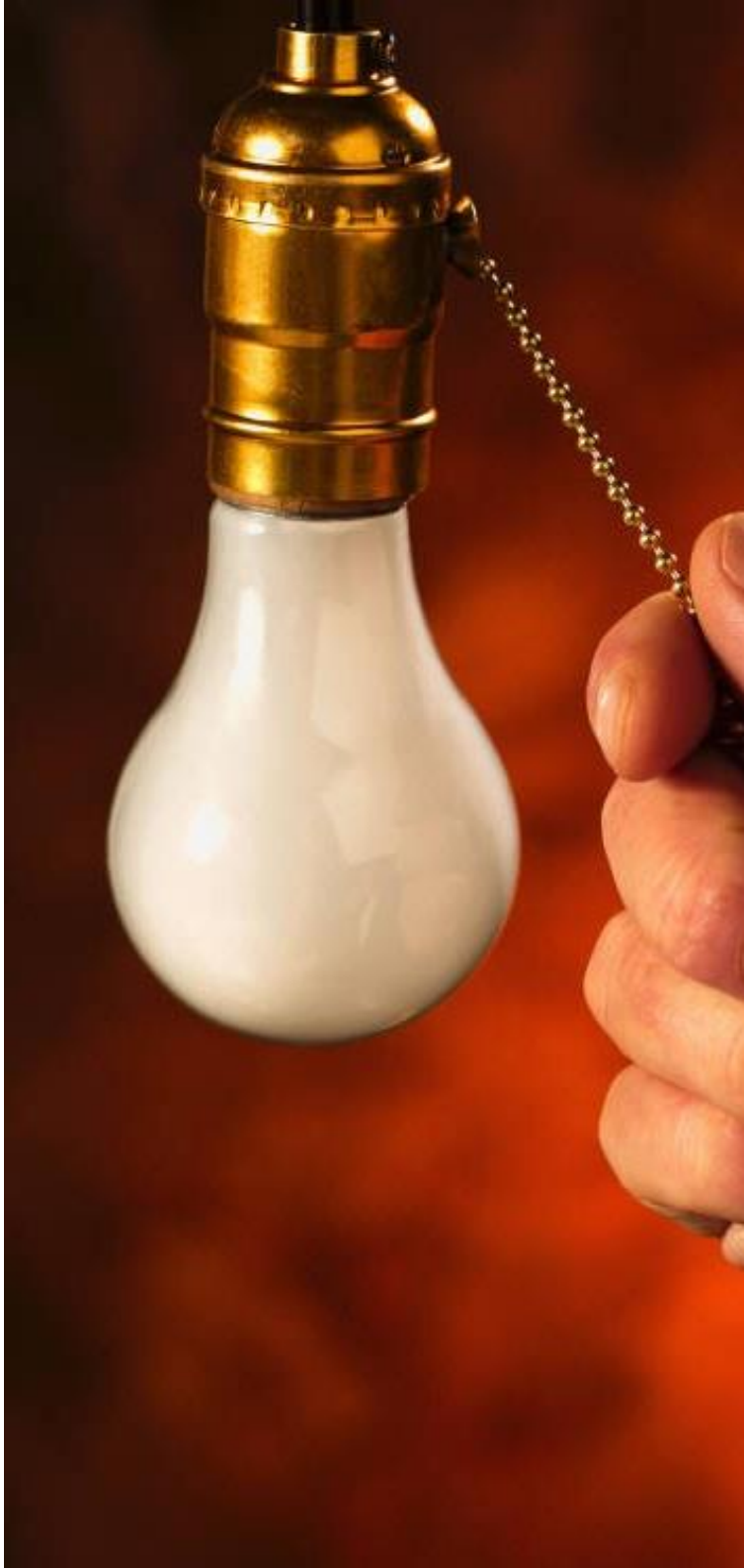
```
private Book toBook(String json) {
    JSONObject jo = new JSONObject(json);
    Book b = new Book(jo.getLong("id"), jo.getString("title"),
        jo.getString("author"), jo.getLong("customerId"));
    return b;
}
```

… but your JSON looks like this

```
{ "siblings" : [
    {
        "id" : 4711L,
        "title" : "The Catcher in the Rye",
        "author" : "J. D. Salinger",
        "customerId" : 2312L
    },
    {
        "id" : 4711L,
        "title" : "The Catcher in the Rye",
        "author" : "J. D. Salinger",
        "customerId" : 0L
    }
    ]
}
```

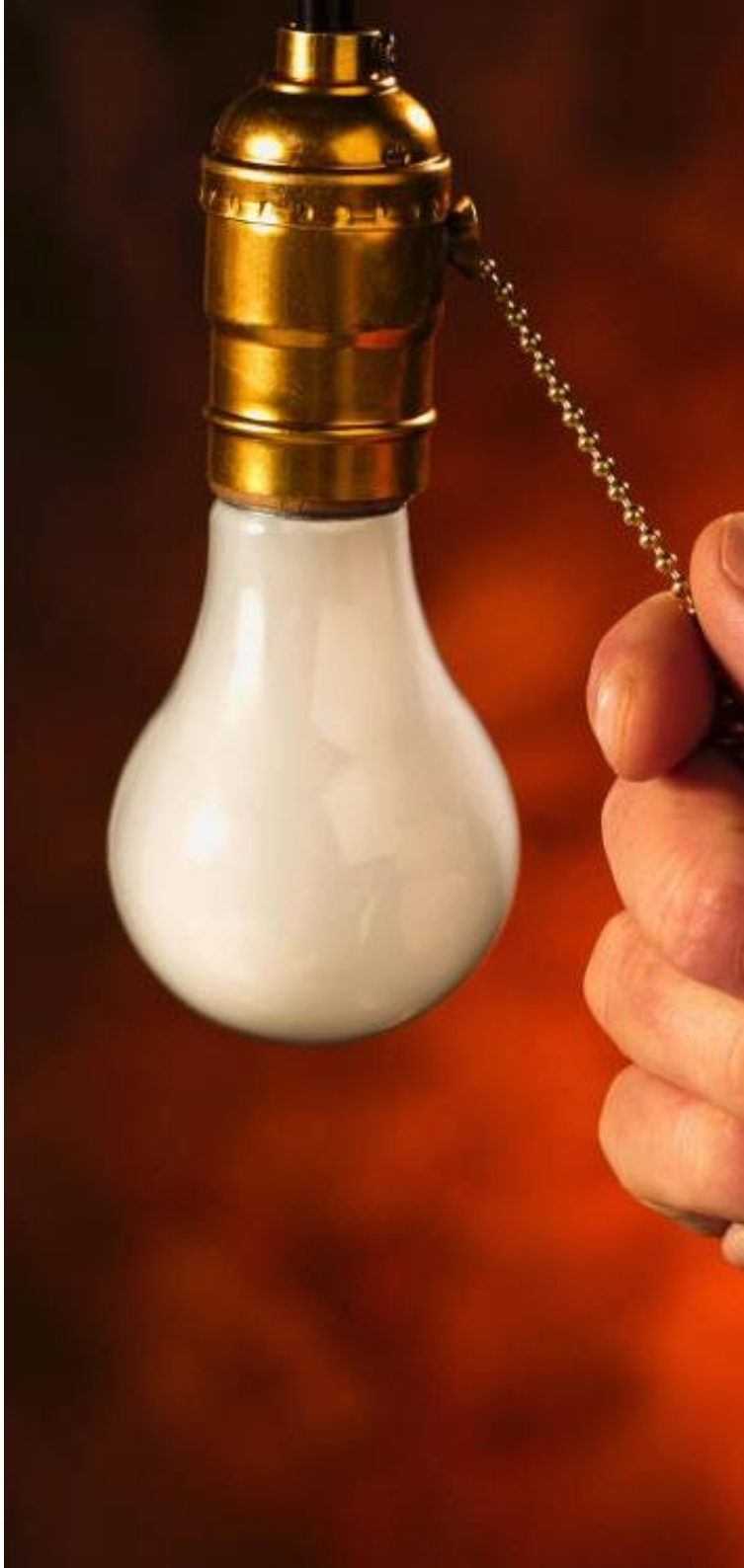
Solution #1

Leave the decision to the caller



Solution #2

Build a resolver



Extended JSON data model

```
Customer:
{
    "id" : <id>,
    "name" : "<name>",
    "timestamp" : <timestamp>
}

Book:
{
    "id" : <id>,
    "title" : "<title>",
    "author" : "<author>",
    "customerId" : <id>,
    "timestamp" : <timestamp>
}
```

Domain Classes

```
public class Customer {
    private Long id;
    private String name;

    <Methods>
}

public class Book {
    private long id;
    private String title;
    private String author;
    private long customerId;

    <Methods>
}
```

Timestamp not required in the domain classes

fromBook

```
private String fromBook(Book b) {
    JSONObject jo = new JSONObject();
    jo.put("id", b.getId());
    jo.put("title", c.getTitle());
    jo.put("author", c.getAuthor());
    jo.put("customerId", b.getCustomerId());
    jo.put("timestamp", System.currentTimeMillis());
    return jo.toString();
}
```

Timestamp added in mapping

Same for „fromCustomer“

toBook

```
private Book toBook(String json) {
    JSONObject jo = new JSONObject(resolveBook(json));
    Book b = new Book(jo.getLong("id"), jo.getString("title"),
        jo.getString("author"), jo.getLong("customerId"));
    return b;
}
```

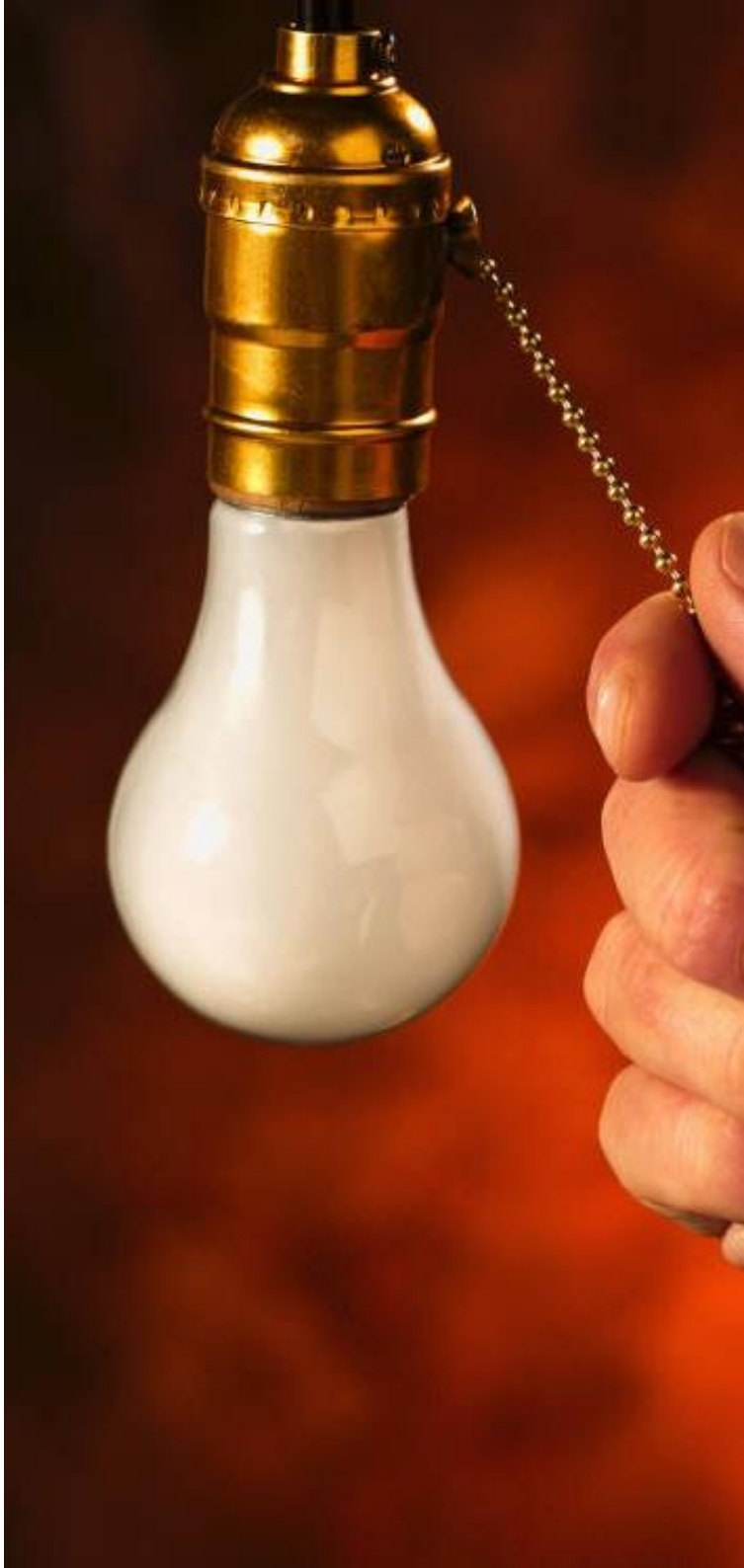
Same for „toCustomer“

resolveBook

```
private String resolveBook(String json) {
    JSONObject jo = new JSONObject(json);
    JSONArray siblings = jo.optJSONArray("siblings");
    if (siblings == null) {
        return json;
    }

    int index = 0;
    long timestamp = 0L
    for (int i = 0; i < siblings.length(); i++) {
        long t = siblings.getJSONObject(i).getLong("timestamp");
        if (t > timestamp) {
            index = i;
            timestamp = t;
        }
    }
    return siblings.getJSONObject(index).toString();
}
```

Solution #3

Conflict-free replicated data types




Iteration #4
Read optimizations

Quick exercise

Assume the following method to retrieve the number of books a customer has loaned

```
public int numberOfBooksLoaned(long cId) {
    return booksLoaned(cId).size();
}
```

Question: How is the performance?

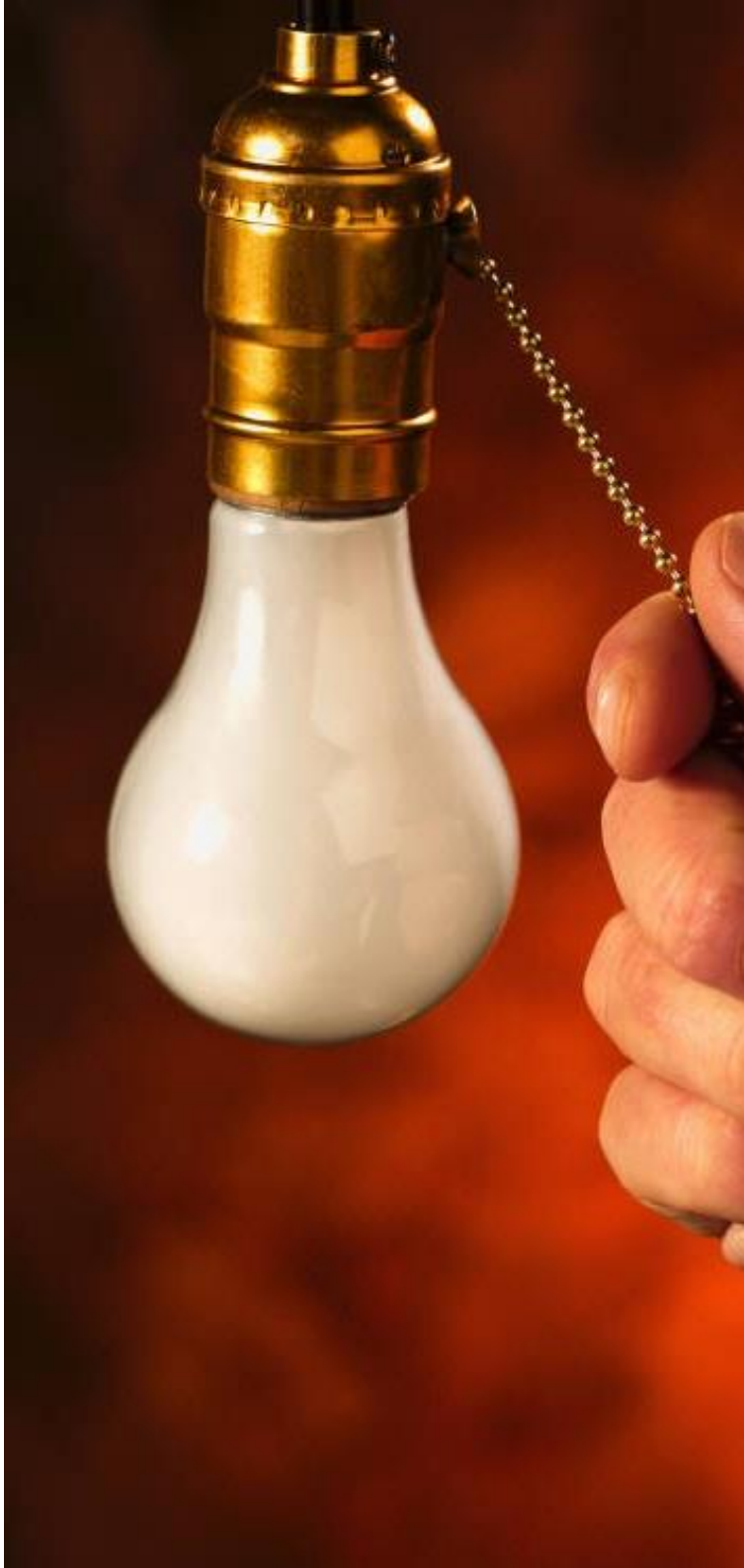
Remember the implementation

```
public List<Book> booksLoaned(long customerId) {
    List<Book> loanedBooks = new ArrayList<Book>();
    Iterator<Book> it = readAllBooks();
    for (Book b : it) {
        if (book.getCustomerId() == customerId) {
            loanedBooks.add(b);
        }
    }
    return loanedBooks;
}

private Iterator<Book> readAllBooks() {
    DBIterator dbi = readAllBooksFromDb();  // Iterator<String> (json)
    return new BookIterator(dbi);
}
```

Reads all book entities distributed across all nodes

Solution #1

Use secondary indices



Re-implementation using indices

```
public List<Book> booksLoaned(long customerId) {
    List<Book> loanedBooks = new ArrayList<Book>();
    Iterator<Book> it = readAllBooksLoanedByACustomer(customerId);
    for (Book b : it) {
        loanedBooks.add(b);
    }
    return loanedBooks;
}

private Iterator<Book> readAllBooksLoanedByACustomer(long customerId) {
    DBIterator dbi =
        readAllBooksUsingSecondaryIndexFromDb(long customerId);
    return new BookIterator(dbi);
}
```

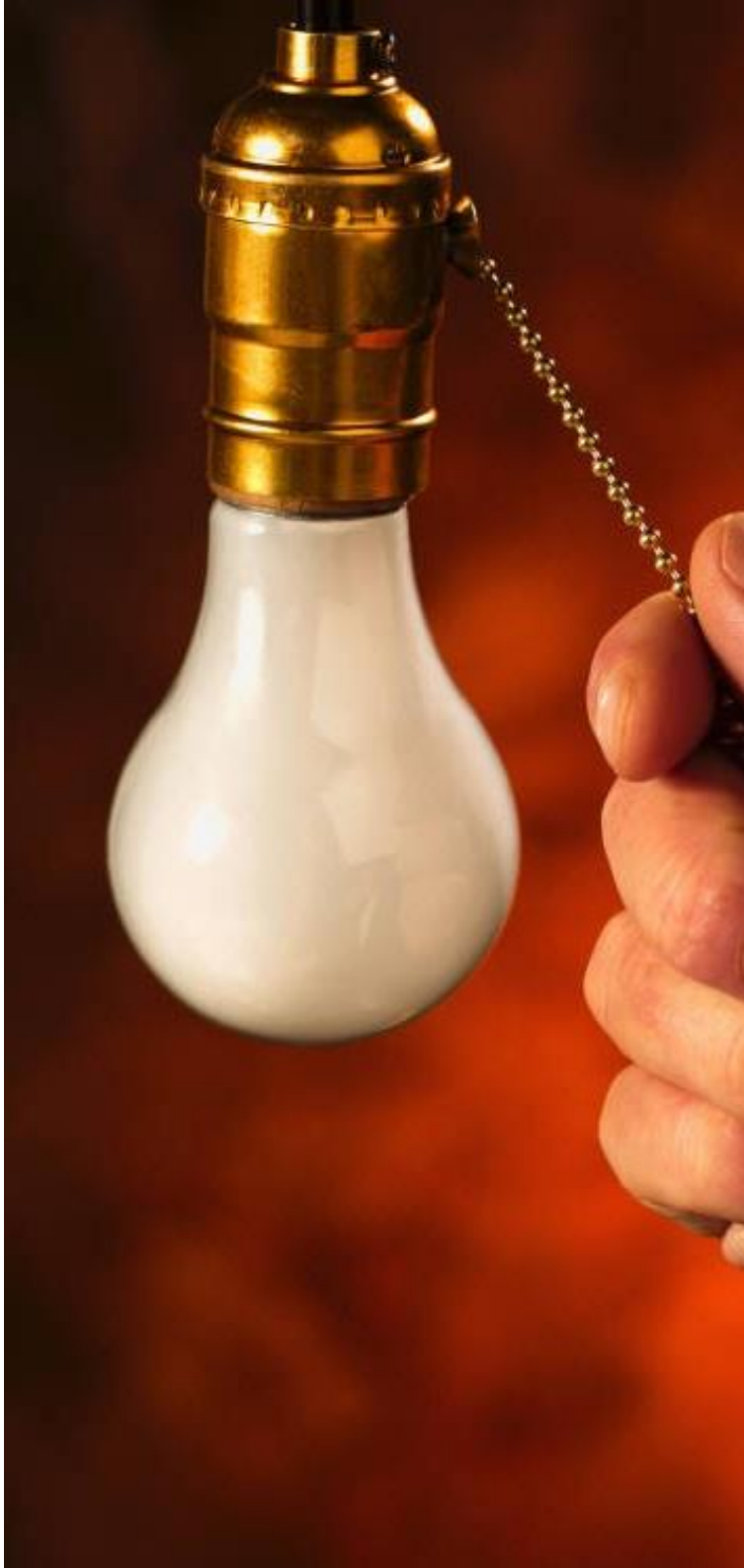
Tradeoffs of indices

- Only works on ACID nodes
- Duplicates while rebalancing
- Partial answers if node is down

… but actually you are interested in a property that logically belongs to a single customer

Solution #2

Denormalize



Denormalized Customer entity

```
{
    "id" : <id>,
    "name" : "<name>",
    "loanedBooks" : [
        {
            "id" : <id>,
            "title" : "<title>",
            "author" : "<author>"
        },
        ...
    ],
    "timestamp" : <timestamp>
}
```

Can be empty

Denormalized Book entity

```
{
    "id" : <id>,
    "title" : "<title>",
    "author" : "<author>",
    "loanedBy" :
        {
            "id" : <id>,
            "name" : "<name>"
        },
    "timestamp" : <timestamp>
}
```

Optional value

Denormalized Customer domain class

```
public class Customer {
    private Long id;
    private String name;
    private List<BookRef> loanedBooks;

    <Methods>
}

public class BookRef { object
    private Long id;
    private String title;
    private String author;

    <Methods>
}
```

Here we go again

Denormalized Book domain class

```
public class Book {
    private Long id;
    private String title;
    private String author;
    private CustomerRef customer;

    <Methods>
}

public class CustomerRef {
    private Long id;
    private String name;

    <Methods>
}
```

Here we go again

Actions „loan“ & „return“

```
public void loan(long customerId, long bookId) {
    Customer customer = readCustomer(customerId);
    Book book = readBook(bookId);
    customer.getLoanedBooks().add(new BookRef(book));
    book.setCustomer(new CustomerRef(customer));
    writeCustomer(customer);
    writeBook(book);
}

public void return(long customerId, long bookId) {
    Customer customer = readCustomer(customerId);
    Book book = readBook(bookId);
    customer.getLoanedBooks().remove(new BookRef(book));
    book.setCustomer(null);
    writeCustomer(customer);
    writeBook(book);
}
```

Also mutates the customer

fromCustomer

```
private String fromCustomer(Customer c) {
    JSONObject jo = new JSONObject();
    jo.put("id", c.getId());
    jo.put("name", c.getName());
    JSONArray ja = new JSONArray();
    for (BookRef b : c.getLoanedBooks()) {
        JSONObject jb = new JSONObject();
        jb.put("id", b.getId());
        jb.put("title", c.getTitle());
        jb.put("author", c.getAuthor());
        ja.put(jb);
    }
    jo.put("loanedBooks", ja);
    jo.put("timestamp", System.currentTimeMillis());
    return jo.toString();
}
```

Also map book references

toCustomer

```
private Customer toCustomer(String json) {
    JSONObject jo = new JSONObject(json);
    JSONArray ja = jo.getJSONArray("loanedBooks");
    List<BookRef> loanedBooks = new ArrayList<BookRef>();
    for (int i = 0; i < ja.length(); i++) {
        JSONObject jb = ja.getJSONObject(i);
        BookRef b = new BookRef(jb.getLong("id"),
                                jb.getString("title"),
                                jb.getString("author"));
        loanedBooks.add(b);
    }
    Customer c = new Customer(jo.getLong("id"), jo.getString("name"),
                              loanedBooks);
    return c;
}
```

Also map book references

fromBook

```
private String fromBook(Book b) {
    JSONObject jo = new JSONObject();
    jo.put("id", b.getId());
    jo.put("title", c.getTitle());
    jo.put("author", c.getAuthor());
    if (c.getCustomer() != null) {
        JSONObject jc = new JSONObject();
        jc.put("id", c.getCustomer().getId());
        jc.put("name", c.getCustomer().getName());
        jo.put("loanedBy", jc);
    }
    jo.put("timestamp", System.currentTimeMillis());
    return jo.toString();
}
```

Also map customer reference

toBook

```
private Book toBook(String json) {
    JSONObject jo = new JSONObject(json);
    CustomerRef c = null;
    JSONObject jc = jo.optJSONObject("loanedBy");
    if (jc != null) {
        c = new CustomerRef(jc.getLong("id"), jc.getString("name"));
    }
    Book b = new Book(jo.getLong("id"), jo.getString("title"),
                      jo.getString("author"), c);
    return b;
}
```

Also map customer reference

Actions „booksLoaned“ & „isBookLoaned“

```
public List<BookRef> booksLoaned(long customerId) {
    Customer customer = readCustomer(customerId);
    return customer.getLoanedBooks();
}

public boolean isBookLoaned(long bookId) {
    Book book = readBook(bookId);
    return book.getCustomer() != null;
}
```

Only one read required

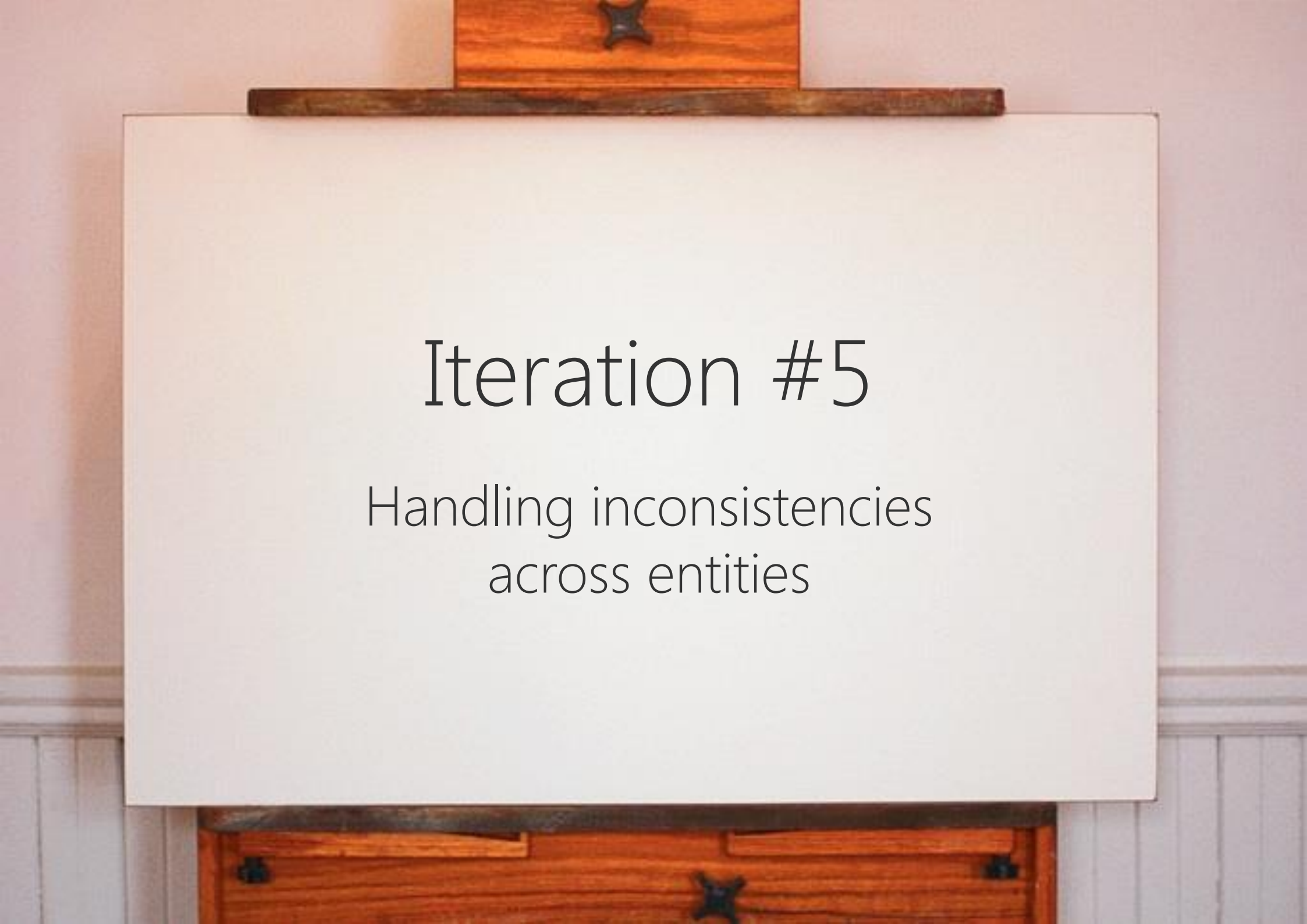
Check becomes more expressive

Method “readAllBooks” not required anymore in this scenario

More denormalization …

- All data within one entity
- Coarse grained entities
- Often huge read speedups
- No OR-Mapper equivalent

- Can result in inconsistencies across entities


Iteration #5
Handling inconsistencies
across entities

Quick exercise

Assume the following code sequence

```
loan(2312L, 4711L);
long bookId = readCustomer(2312L)
    .getLoanedBooks().get(0).getId();
long customerId = readBook(4711L)
    .getCustomer().getId();
```

Question: What do you get?

Remember the implementation

```
public void loan(2312L, 4711L) {
    Customer customer = readCustomer(2312L);
    Book book = readBook(4711L);
    customer.getLoanedBooks().add(new BookRef(book));
    book.setCustomer(new CustomerRef(customer));
    writeCustomer(customer);
    writeBook(book);
}
```

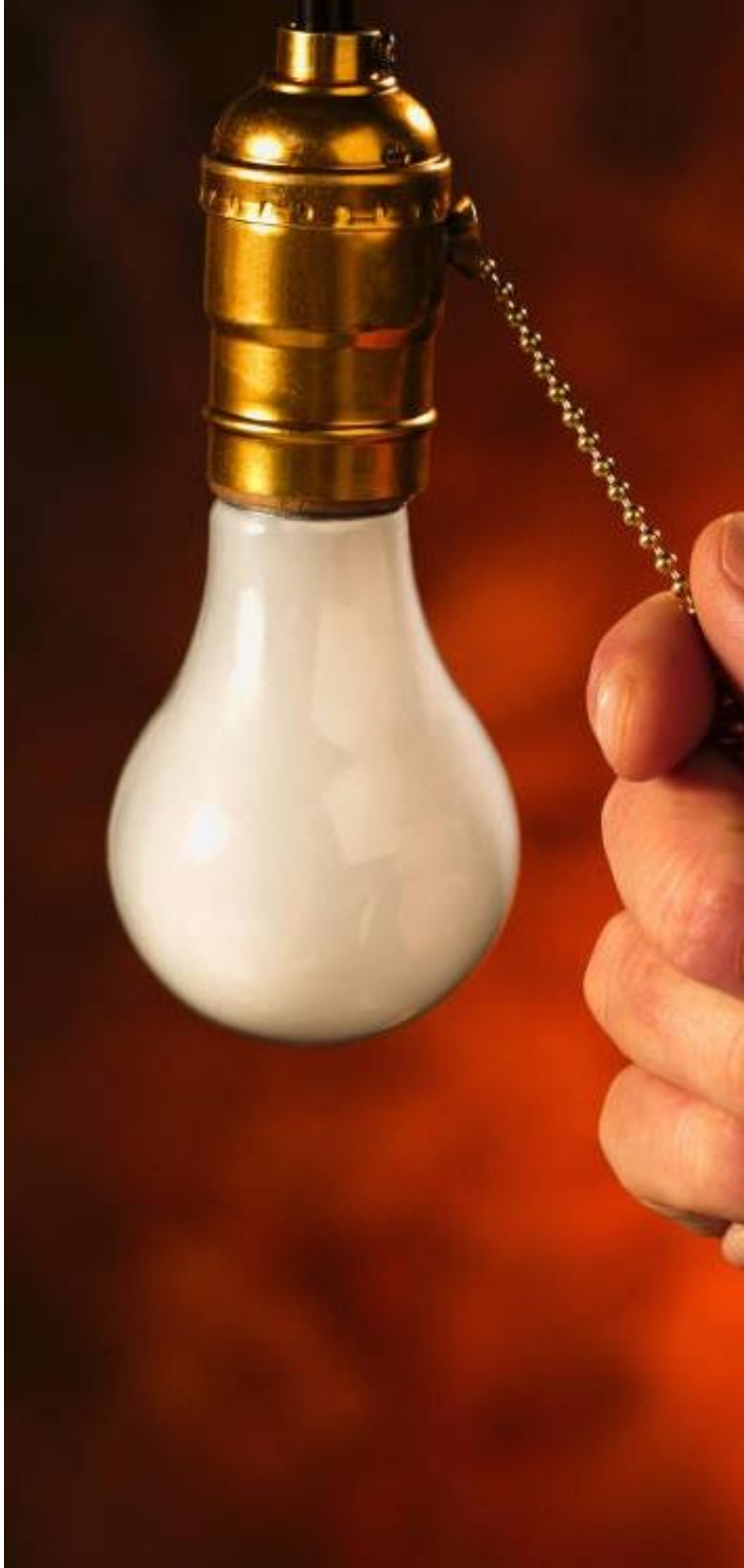
This call fails

Leads to the following entity state

```
{
    "id" : 2312L,
    "name" : "Uwe Friedrichsen",
    "loanedBooks" : [
        {
            "id" : 4711L,
            "title" : "The Catcher in the Rye",
            "author" : "J. D. Salinger"
        }
    ],
    "timestamp" : <timestamp>
}

{
    "id" : 4711L,
    "title" : "The Catcher in the Rye",
    "author" : "J. D. Salinger",
    "timestamp" : <timestamp>
}
```

Solution #1

On-the-fly resolver



Actions „loan“ & „return“

```
public void loan(long customerId, long bookId) {
    Customer customer = readCustomer(customerId);
    Book book = readBook(bookId);
    customer.getLoanedBooks().add(new BookRef(book));
    book.setCustomer(new CustomerRef(customer));
    writeBook(book);
    writeCustomer(customer);
}

public void return(long customerId, long bookId) {
    Customer customer = readCustomer(customerId);
    Book book = readBook(bookId);
    customer.getLoanedBooks().remove(new BookRef(book));
    book.setCustomer(null);
    writeBook(book);
    writeCustomer(customer);
}
```

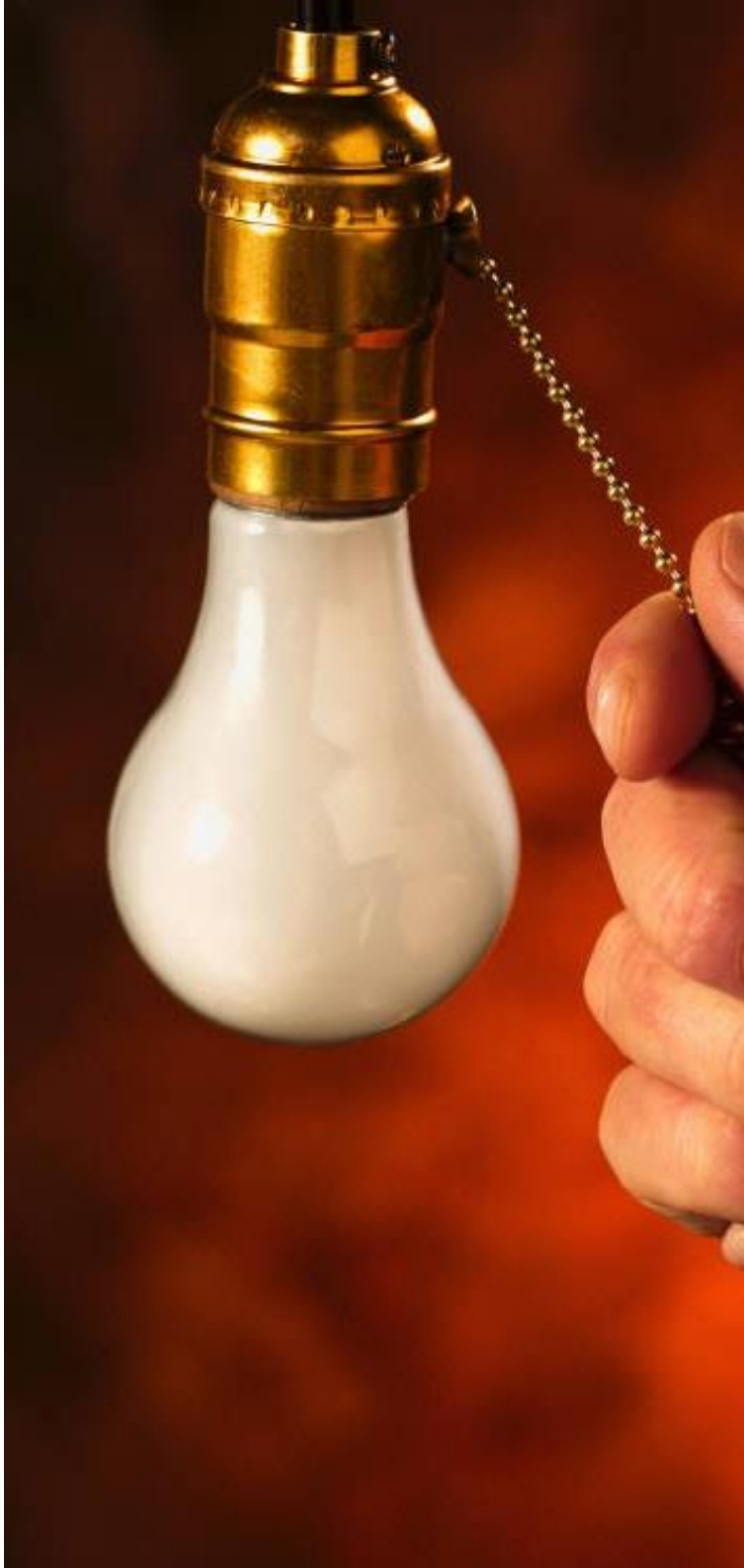
Switched write order

resolveIfRequired

```
private void resolveIfRequired(Customer c, Book b) {
    BookRef br = new  BookRef(b.getId(), b.getTitle(), b.getAuthor());
    if (!c.getLoanedBooks().contains(br)) {
        c.getLoanedBooks.add(br);
    }
}

// some code that (for some reason) reads both related entities
...
Book book = readBook(bookId);
Customer customer = null;
If (book.getCustomer() != null) {
    readCustomer(book.getCustomer().getId());
    resolveIfRequired(customer, book);
}
// Do stuff with customer and book
...
```

Solution #2

On-the-fly resolver with write-back (“read repair”)



resolveIfRequired

```
private void resolveIfRequired(Customer c, Book b) {
    BookRef br = new  BookRef(b.getId(), b.getTitle(), b.getAuthor());
    if (!c.getLoanedBooks().contains(br)) {
        c.getLoanedBooks.add(br);
        writeCustomer(c);
    }
}
```

Best effort principle

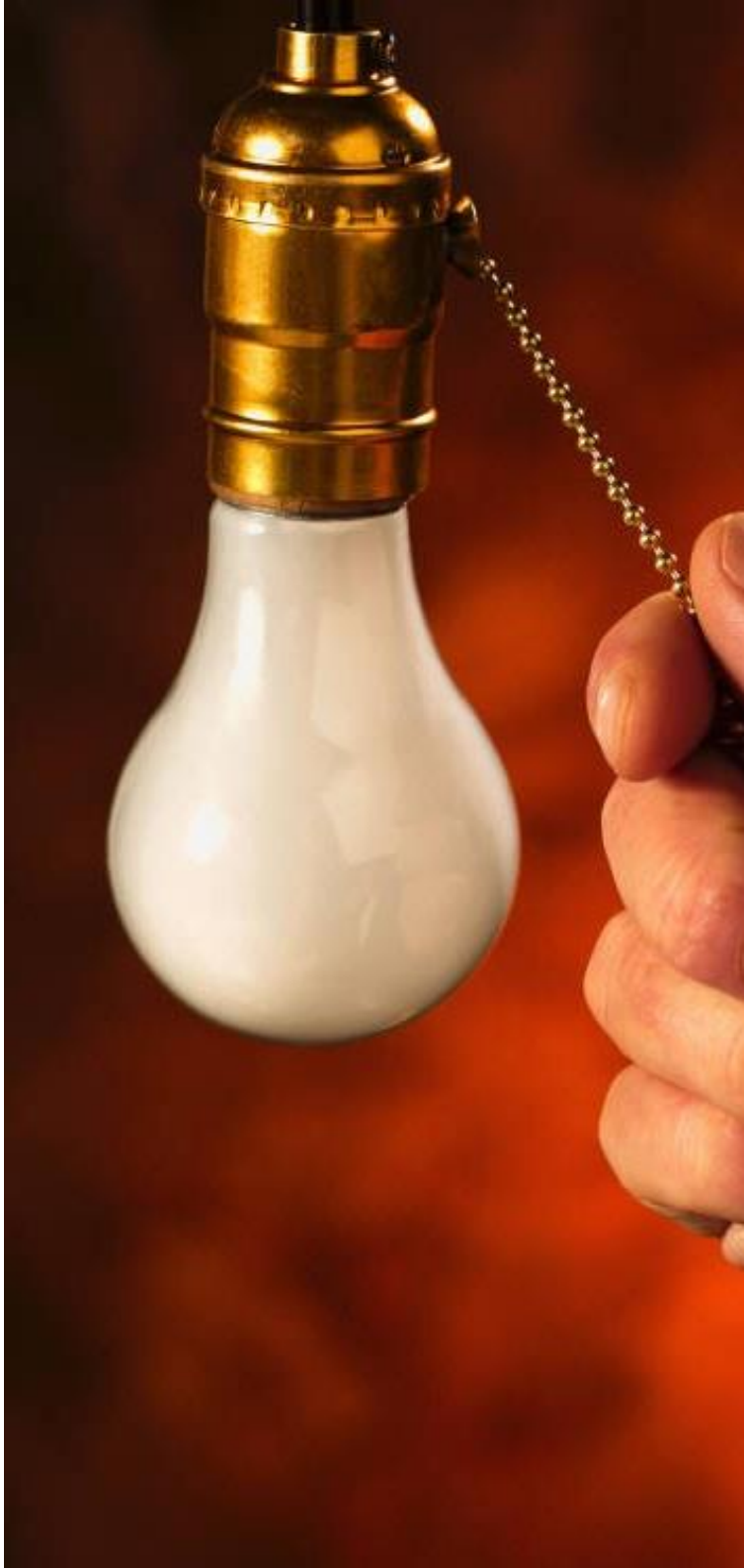
Advanced solutions

Consistency crawler

Compensating actions ???

Change journal

…



Wrap-up

- BASE is sufficient, but different
- New challenges for a developer

- Some databases promise to hide BASE …
  … but you can't fool the laws of distribution

- Inconsistencies will happen – be prepared …
- … and then have lots of fun!

@ufried

Uwe Friedrichsen | uwe.friedrichsen@codecentric.de | http://slideshare.net/ufried | http://ufried.tumblr.com