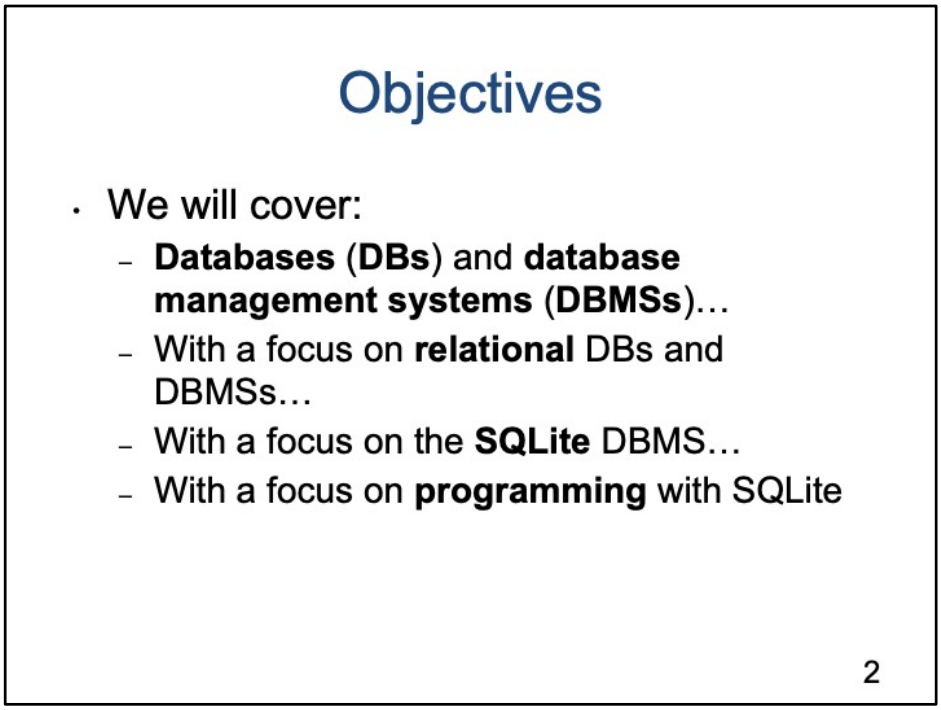
Database Programming (Part 3)

Robert M. Dondero, Ph.D.
Princeton University



Objectives

Continued from Part 2…

[see slide]

Agenda

- **Relational DB transactions: atomicity**
- Relational DB transactions: locking
- Relational DB design

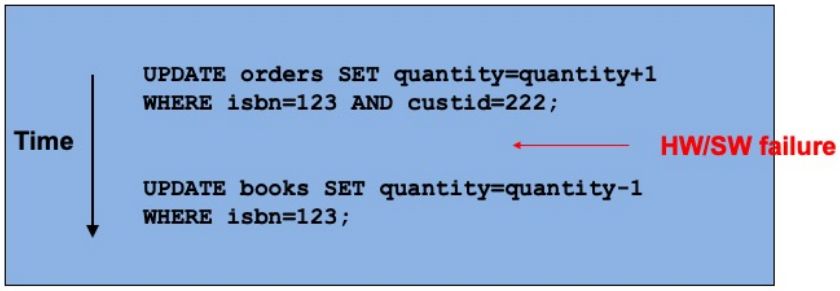
DB Transactions: Atomicity

- **Problem**: DBMS must preserve DB consistency in the presence of HW/SW failures

DB Transactions: Atomicity

- **Example**:
  - Customer 222 has purchased 1 copy of book 123
  - Must change ORDERS:
    - Add 1 to quantity of appropriate row (20 → 21)
  - Must change BOOKS:
    - Subtract 1 from quantity of appropriate row (500 → 499)

DB Transactions: Atomicity



One logical update consists of 2 UPDATE stmts
Failure occurs between execution of stmts
=> DB is corrupted

DB Transactions: Atomicity

- **Solution**:
  - DBMS must execute each logical DB update ***atomically***

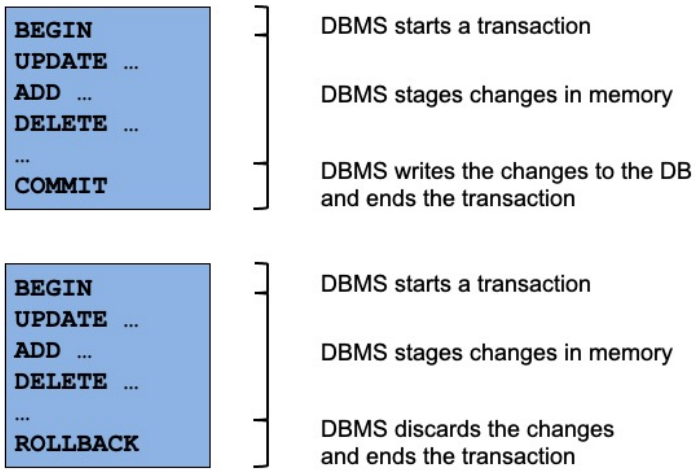
- ***Atomically***: either completely, or not at all

DB Transactions: Atomicity

- **Problem**: How to implement atomicity?
- **Solution**: ***Transaction***
  - Defines an atomic unit of work

- DBMS executes a transaction either completely or not at all

DB Transactions: Atomicity

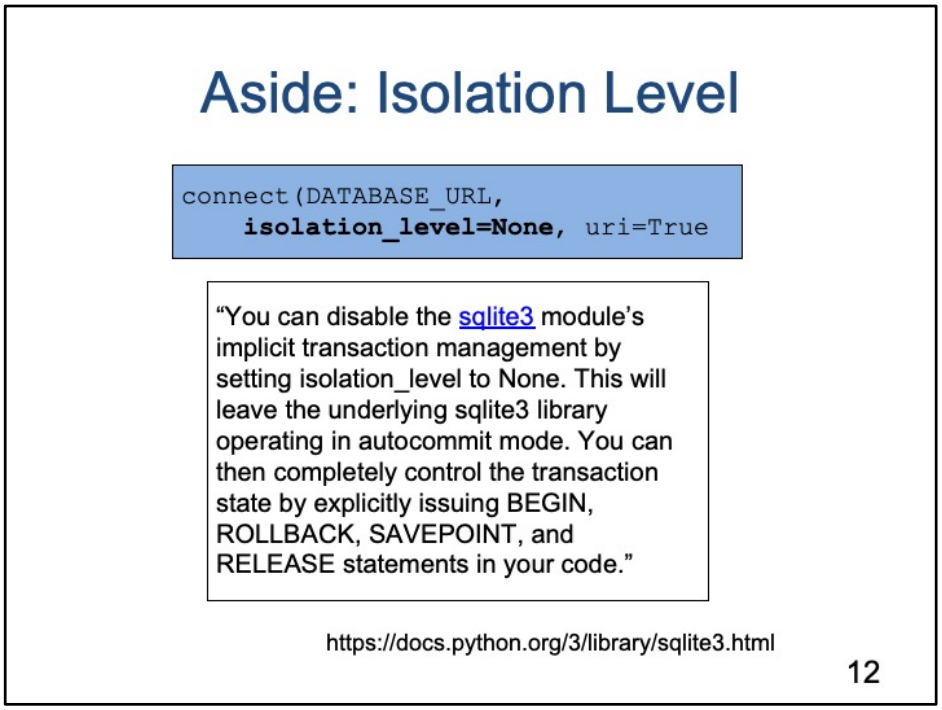


DB Transactions: Atomicity

- See **purchase.py**
  - The job:
    - Accept `isbn` and `custid` as command-line arguments
    - In ORDERS table increment `quantity` of row with given `isbn` and `custid`
    - In BOOKS table decrement `quantity` of row with given `isbn`
  - Note:
    - Two updates must be executed atomically

DB Transactions: Atomicity

```
$ python display.py
----------------------------------------
books
----------------------------------------
('123', 'The Practice of Programming', 500)
…
----------------------------------------
orders
----------------------------------------
('123', '222', 20)
…
$ python purchase.py 123 222
Transaction committed.
$ python display.py
----------------------------------------
books
----------------------------------------
('123', 'The Practice of Programming', 499)
…
----------------------------------------
orders
----------------------------------------
('123', '222', 21)
…
$
```



Aside: Isolation Level

isolation_level=None

Informally

Turns off the transaction management of the sqlite3 database driver

Allows you directly to use the transaction management of the underlying SQLite DB

Thereby…

Transactions when using Python behave the same as do transactions when using the sqlite3 command-line client program

DB Transactions: Atomicity

- Do SQLite transactions in Python really handle HW/SW errors?
- Do they really enforce atomicity?

DB Transactions: Atomicity

- See **recovery.py**
  - The job (artificial):
    - Repeatedly, 20 times…
    - In orders table increment `quantity` of row with `isbn` 123 and `custid` 222
    - In books table decrement `quantity` of row with `isbn` 123
    - "Failure" may occur between the updates
    - Is DB consistency preserved?

DB Transactions: Atomicity

```
$ python display.py
----------------------------------------
books
----------------------------------------
('123', 'The Practice of Programming', 500)
…
----------------------------------------
orders
----------------------------------------
('123', '222', 20)
…
$
```

DB Transactions: Atomicity

```
$ python recovery.py
Transaction 0 committed.
Transaction 1 committed.
Transaction 2 committed.
Transaction 3 committed.
Transaction 4 committed.
Simulated failure with i = 5
Transaction 5 rolled back.
Transaction 6 committed.
Transaction 7 committed.
Transaction 8 committed.
Transaction 9 committed.
Transaction 10 committed.
Simulated failure with i = 11
Transaction 11 rolled back.
Transaction 12 committed.
Simulated failure with i = 13
Transaction 13 rolled back.
Transaction 14 committed.
Transaction 15 committed.
Transaction 16 committed.
Transaction 17 committed.
Transaction 18 committed.
Transaction 19 committed.
$
```

DB Transactions: Atomicity

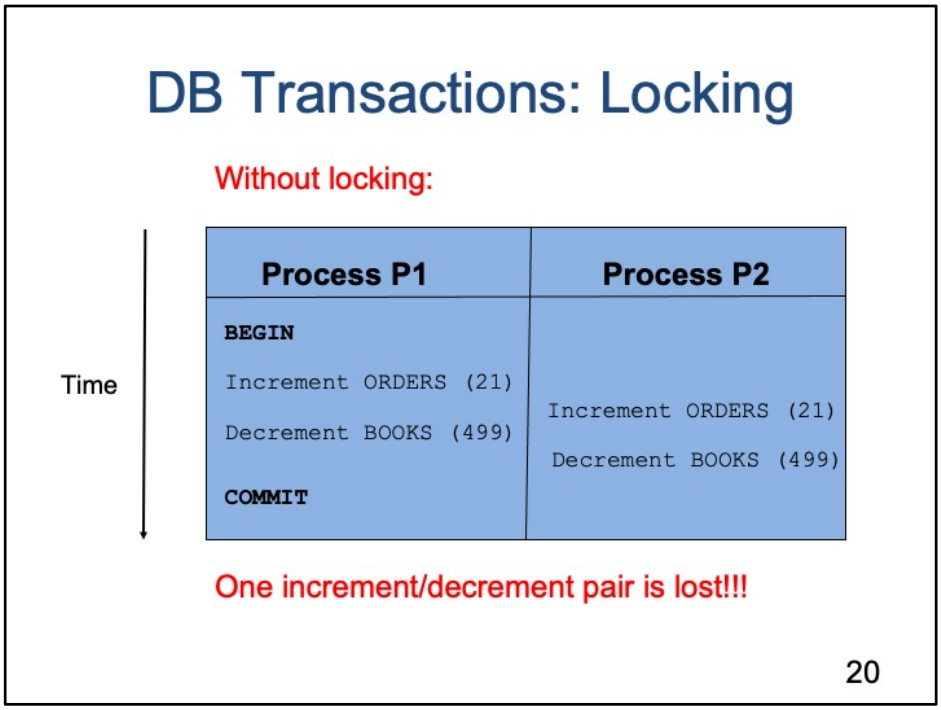
```
$ python display.py
----------------------------------------
books
----------------------------------------
('123', 'The Practice of Programming', 483)
…
----------------------------------------
orders
----------------------------------------
('123', '222', 37)
…
$
```

Agenda

- Relational DB transactions: atomicity
- **Relational DB transactions: locking**
- Relational DB design

DB Transactions: Locking

- **Problem**: DBMS must preserve DB consistency in the presence of concurrent updates



Relational DB Transactions: Locking

Without locking, consider this sequence

[see slide]

P1 begins a transaction

P1 increments the quantity in ORDERS, thus changing it to 21
But the increment occurs within a transaction, and so is staged

P2 increments the quantity in ORDERS, thus changing it to 21

P1 decrements the quantity in BOOKS, thus changing it to 499
But the decrement occurs within a transaction, and so is staged

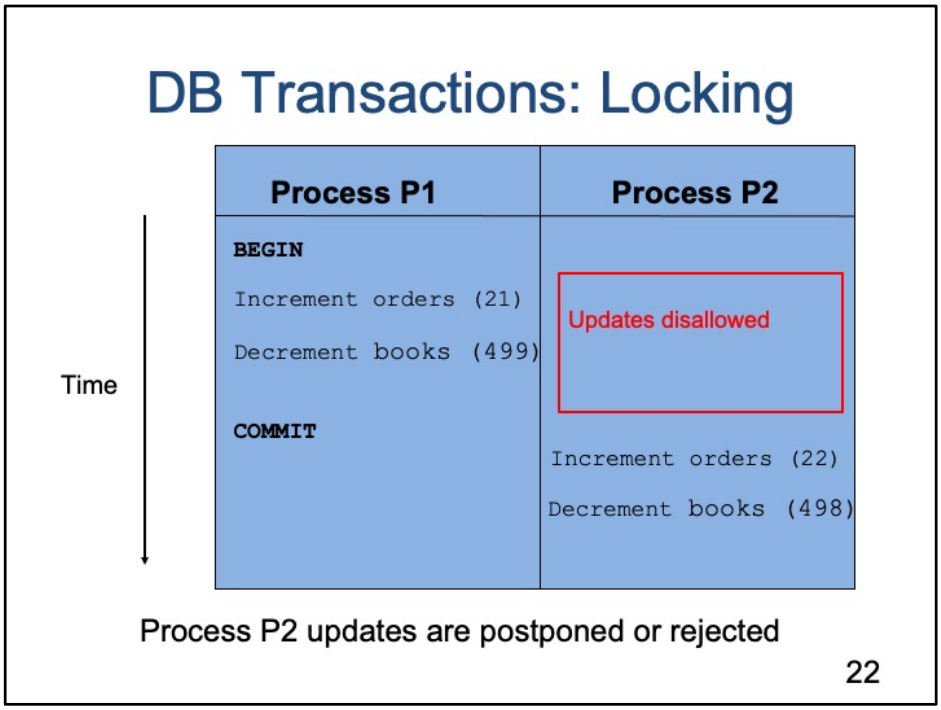
P2 decrements the quantity in BOOKS, thus changing it to 499

P1 commits the transaction
The quantity in ORDERS is changed to 21
The quantity in BOOKS is changed to 499

One increment/decrement pair is lost!

DB Transactions: Locking

- **Solution**: Locking within transactions
  - Process P1 asks DBMS to execute transaction involving row X
    - DBMS ***locks*** row X
  - Concurrently, process P2 asks DBMS to update row X
    - DBMS forces P2 to wait, or rejects P2's request, until transaction completes



Relational DB Transactions: Locking

With locking, consider this similar sequence

[see slide]

P1 begins a transaction

During the transaction, P2's attempts to update are disallowed (postponed or rejected)

P1 increments the quantity in ORDERS, thus changing it to 21
But the increment occurs within a transaction, and so is staged

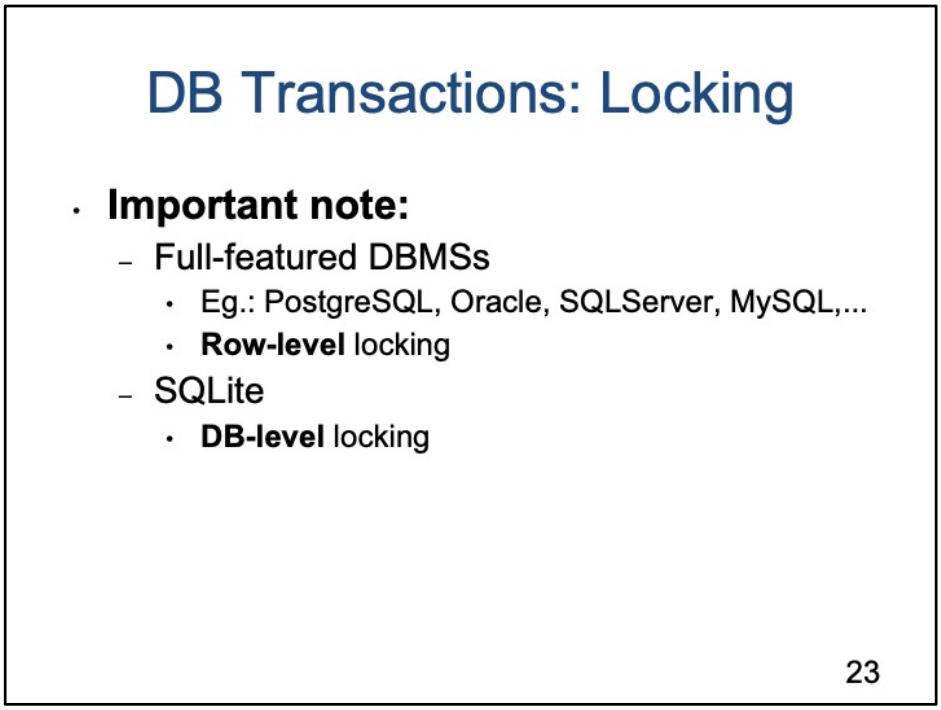
P1 decrements the quantity in BOOKS, thus changing it to 499
But the decrement occurs within a transaction, and so is staged

P1 ends/commits the transaction
The quantity in ORDERS is changed to 21
The quantity in BOOKS is changed to 499

P2 increments the quantity in ORDERS, thus changing it to 22

P2 decrements the quantity in BOOKS, thus changing it to 498

No increments/decrements are lost!



Relational DB Transactions: Locking

Important note:

> Full-featured DBMSs (PostgreSQL, Oracle, SQLServer, MySQL, …), do ***row-level*** locking
>
> SQLite does ***DB-level*** locking

A big reason why SQLite isn't used in production applications

More precisely

> SQLite was designed to be used in production applications as a **strong** alternative to simple flat files
> Thunderbird email client uses a SQLite database
>
> SQLite was not designed to be used in production applications as a **weak** alternative to a full-featured DBMS

DB Transactions: Locking

Transaction locking in SQLite

After P1 does this on some DB:	P2 can do this on that DB:
`BEGIN`	`SELECT | UPDATE`
`SELECT | UPDATE`	`SELECT`
`COMMIT | ROLLBACK`	`SELECT | UPDATE`

DB Transactions: Locking

Terminal session 1:

```
$ sqlite3 bookstore.sqlite
sqlite> BEGIN;
sqlite> UPDATE books SET quantity = 11111 WHERE isbn = 123;
sqlite>
```

Terminal session 2:

```
$ python purchase.py 123 222
database is locked
$
```

DB Transactions: Locking

Terminal session 1 (cont):

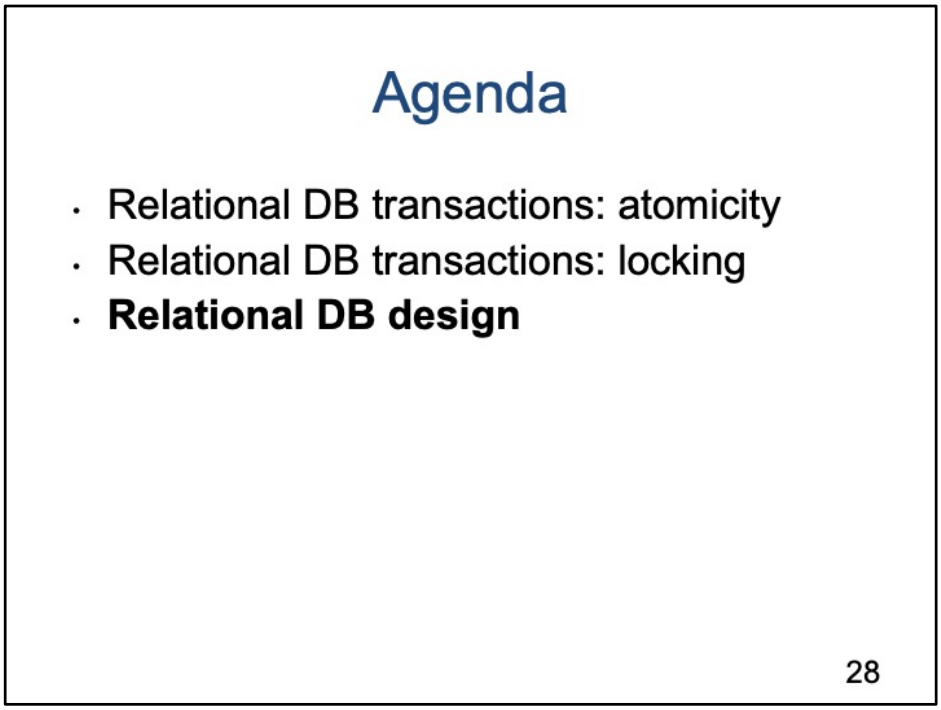
```
$ sqlite3 bookstore.sqlite
sqlite> BEGIN;
sqlite> UPDATE books SET quantity = 11111 WHERE isbn = 123;
sqlite> COMMIT;
sqlite>
```

Terminal session 2 (cont):

```
$ python purchase.py 123 222
database is locked
$ python purchase.py 123 222
Transaction committed.
$
```

Transaction Summary

- DBMSs use **transactions** to:
  - Recover from HW/SW errors
    - Transactions implement **atomicity**
  - Handle concurrent updates
    - Transactions implement **locking**



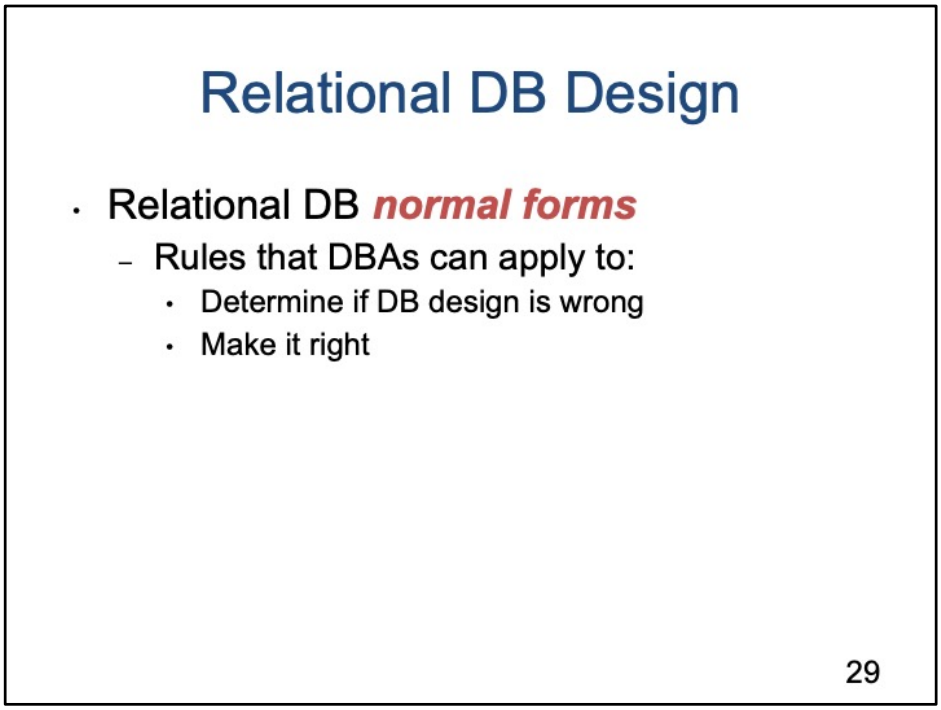
Relational DB Design

So far:

How to **use** a relational DB

Now:

How to **design** a relational DB



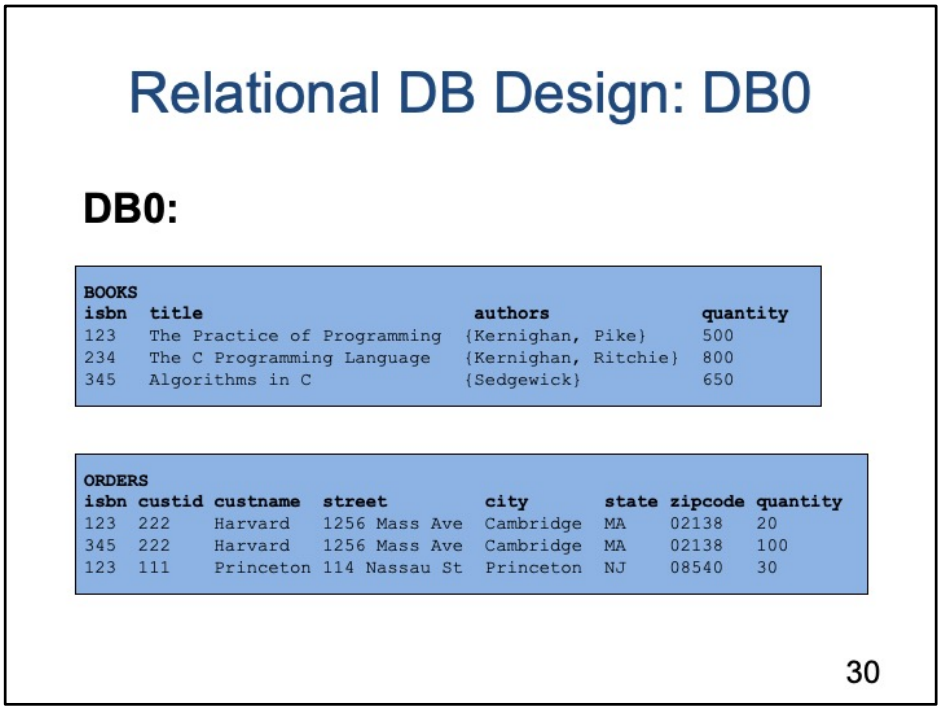
Relational DB Design

To design a proper relational DB, you must know about the relational DB ***normal forms***

Developed by Codd

Rules that DBAs can apply to:
- Determine if DB design is wrong
- Make it right

There are many of them
We'll cover the first 3…



Relational DB Design: DB0

Let's consider a series of "bookstore" relational databases
We'll start with a very flawed version
We'll incrementally apply normal forms
The final version in the series will be the database that we used in previous lectures

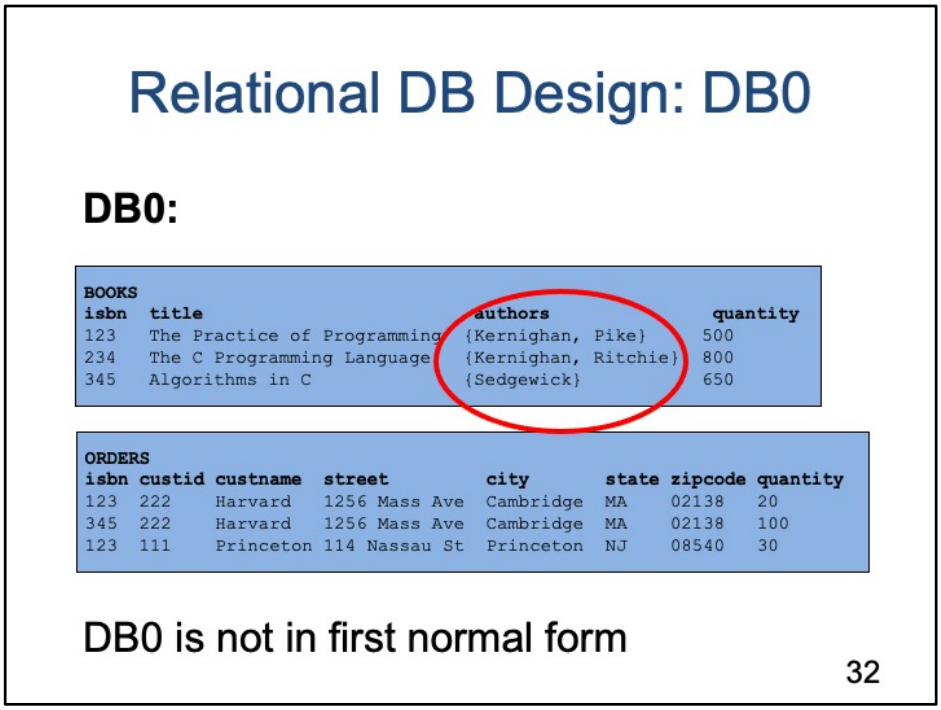
Consider DB0…

BOOKS:
There is a book whose isbn is 123
Whose title is "The Practice of Programming"
Whose authors are "Kernighan" and "Pike"
And there are 500 copies of it available in our warehouse

ORDERS:
There is a customer whose custid is 222
That customer is the Harvard bookstore
It's located at 1256 Mass Ave in Cambridge, MA 02138
That customer ordered 20 copies of the book with isbn 123

Relational DB Design: DB0

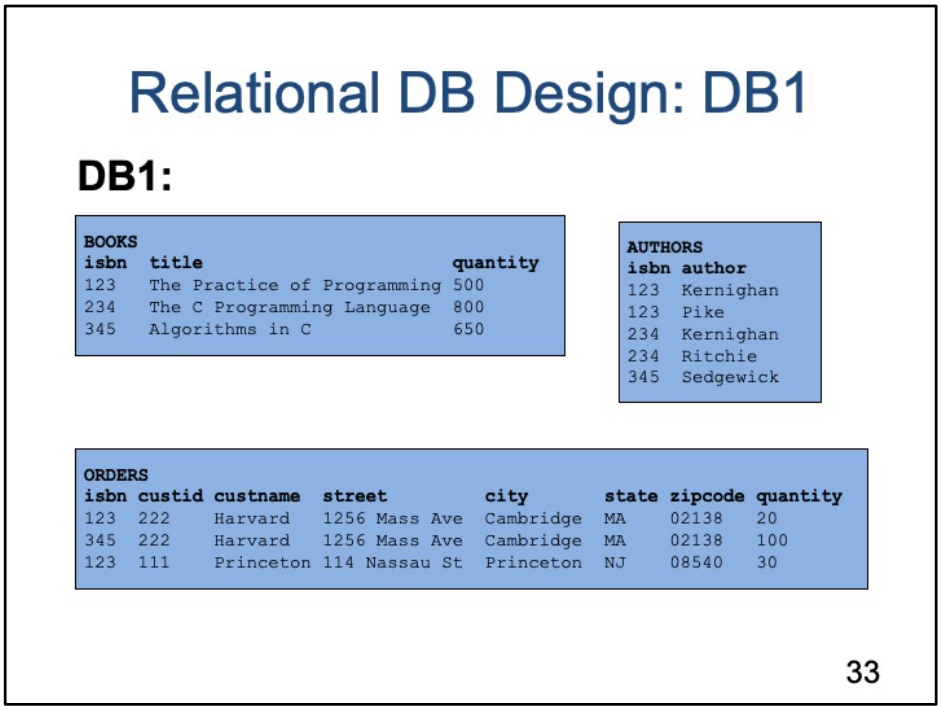
- **Def**: A table is in ***first normal form*** iff all underlying domains contain atomic values only
  - All modern relational DBMSs enforce first normal form



Relational DB Design: DB0

Recall: A table is in ***first normal form*** iff all underlying domains contain atomic values only

DB0 has cells that contain non-atomic values



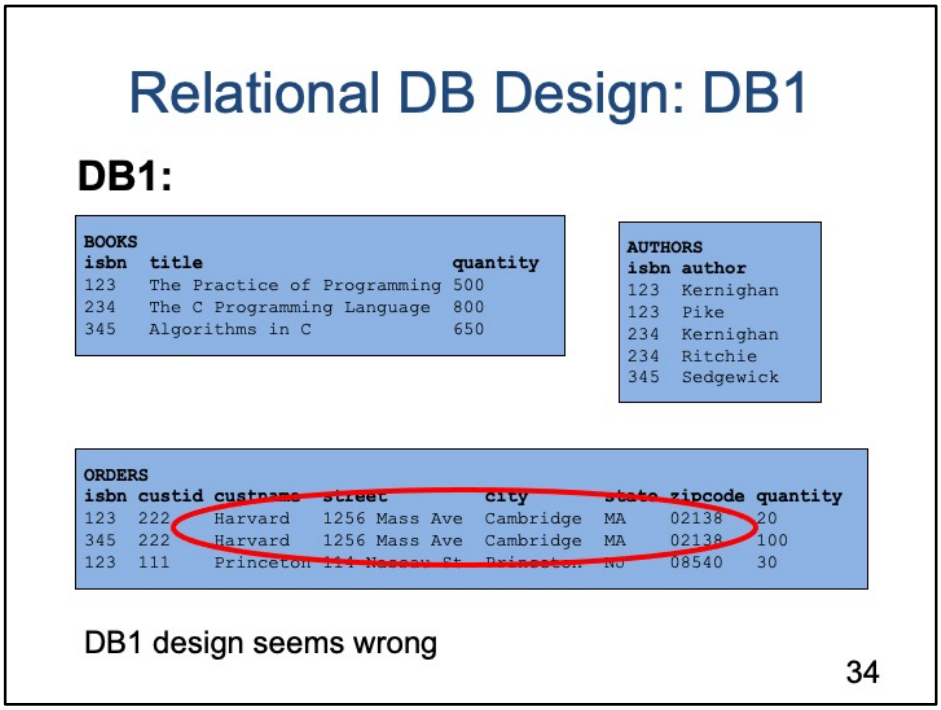
Relational DB Design: DB1

The solution: split BOOKS into BOOKS and AUTHORS

Yields DB1

[see slide]

DB1 is in first normal form



Relational DB Design: DB1

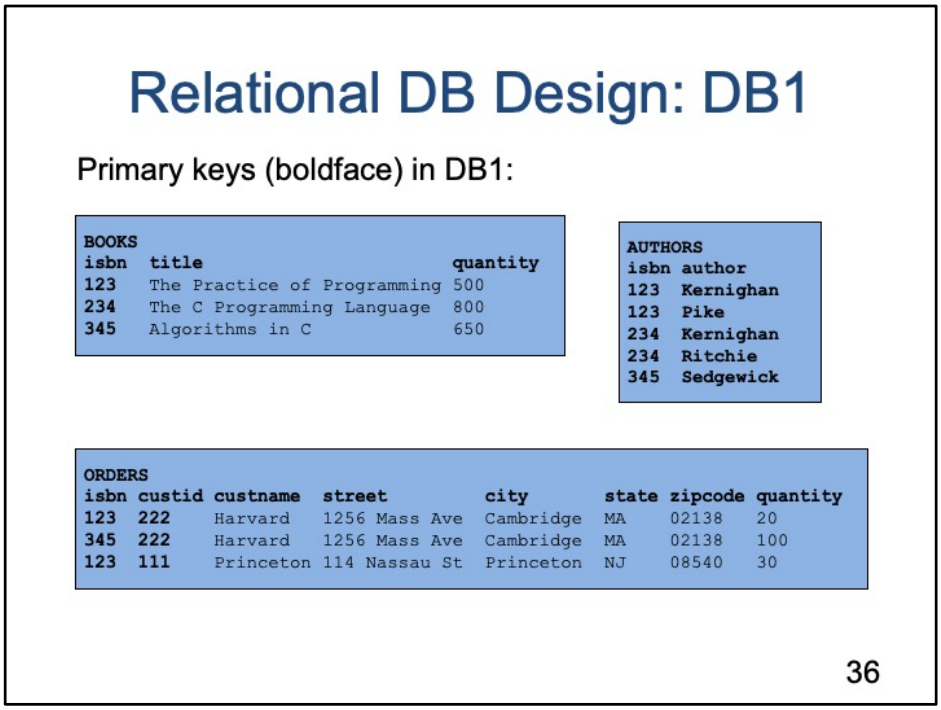
[see slide]

Intuitively: Still not quite right

The location of the Harvard bookstore is stored redundantly
Suppose the Harvard bookstore moves?

Relational DB Design: DB1

- Somewhat informally...
- **Def**: The ***primary key*** for a table is the minimal set of columns that uniquely identifies any particular row of that table



Relational DB Design: DB1

[see slide]

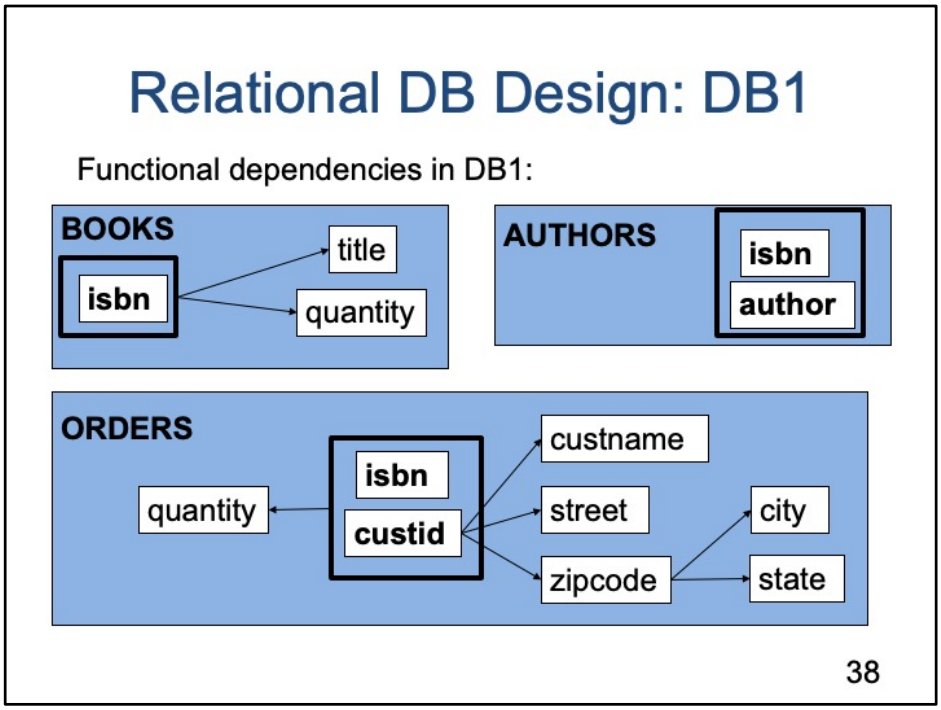
In BOOKS the primary key is isbn

In ORDERS the primary key is isbn+custid

In AUTHORS the primary key is isbn+author
Degenerate case

Relational DB Design: DB1

- **Def**: A column C2 of a table is ***functionally dependent*** on a column C1 iff, for each row in the table, the value of C1 determines the value of C2
- In DB1...



Relational DB Design: DB1

[see slide]

BOOKS:

title and quantity are functionally dependent upon isbn

isbn uniquely identifies a row

If we know the isbn, then we can determine the title and quantity

AUTHORS: degenerate case

ORDERS:

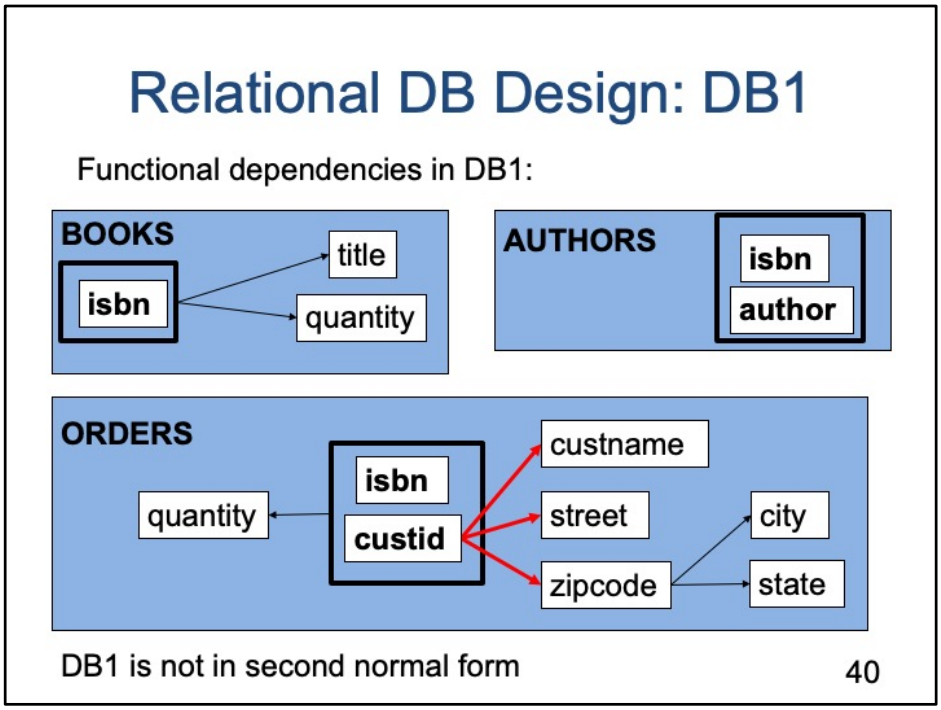
Quantity is functionally dependent upon isbn+custid

Custname, street, and zipcode are functionally dependent upon custid

City and state are functionally dependent upon zipcode

Relational DB Design: DB1

- Somewhat informally…
- A table is in ***second normal form*** iff:
  - It is in first normal form, and
  - Every non-primary-key column is dependent on all fields of the primary key



Relational DB Design: DB1

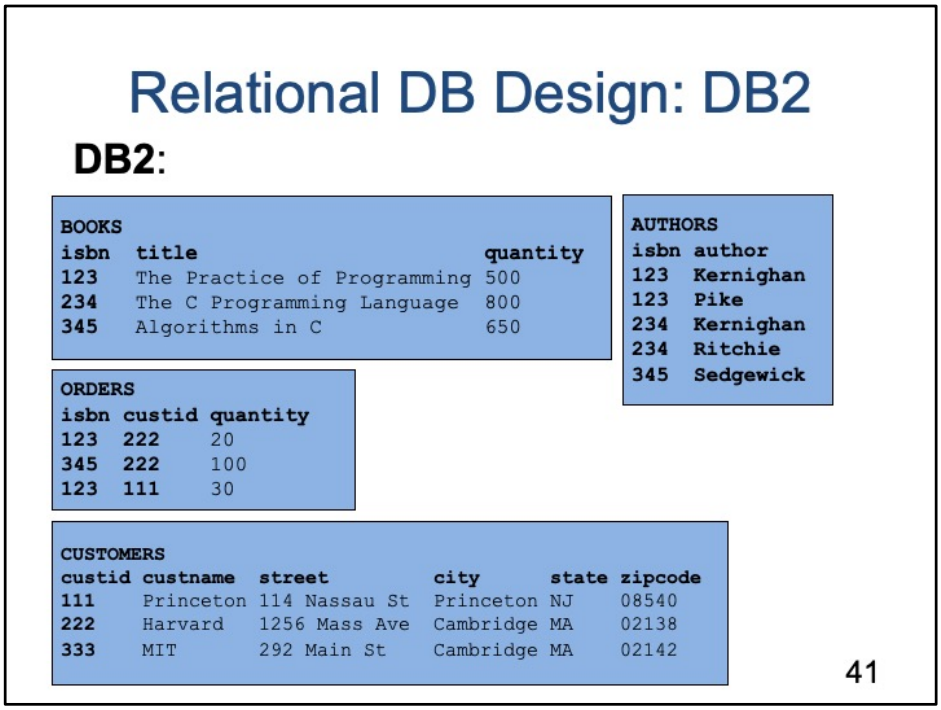
*Recall: A table is in **second normal form** iff:*

It is in first normal form, and

Every non-primary-key column is dependent on the entire primary key

In ORDERS:

Custname, street, and zipcode are functionally dependent upon only one field of the two-field isbn+custid primary key



Relational DB Design: DB2

*Recall: A table is in **second normal form** iff:*

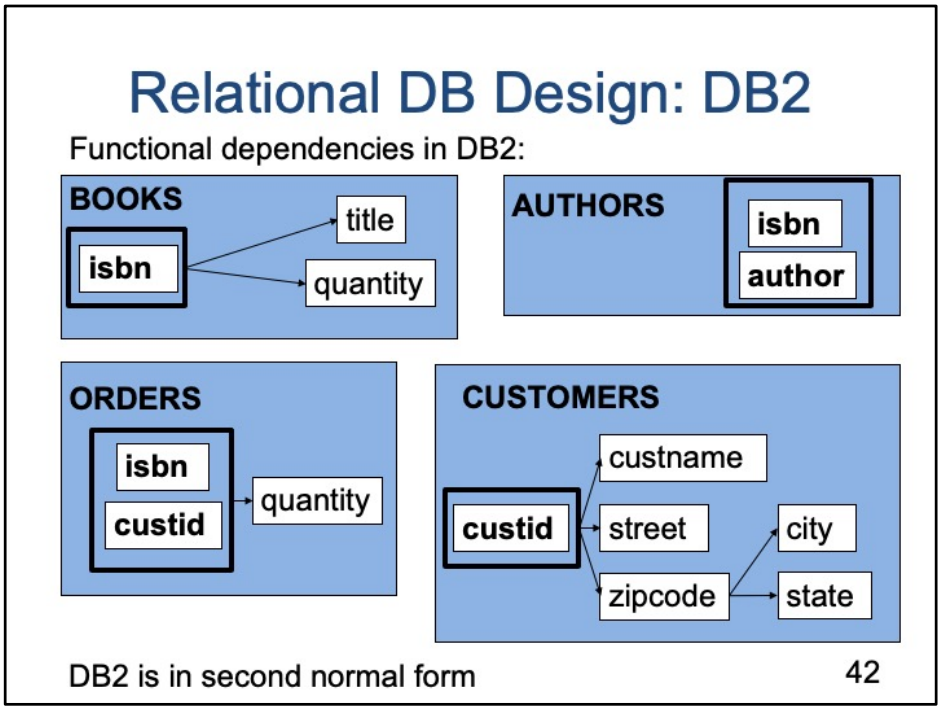
It is in first normal form, and

Every non-primary-key column is dependent on the entire primary key

Solution: Split ORDERS into ORDERS and CUSTOMERS

Yields DB2

[see slide]



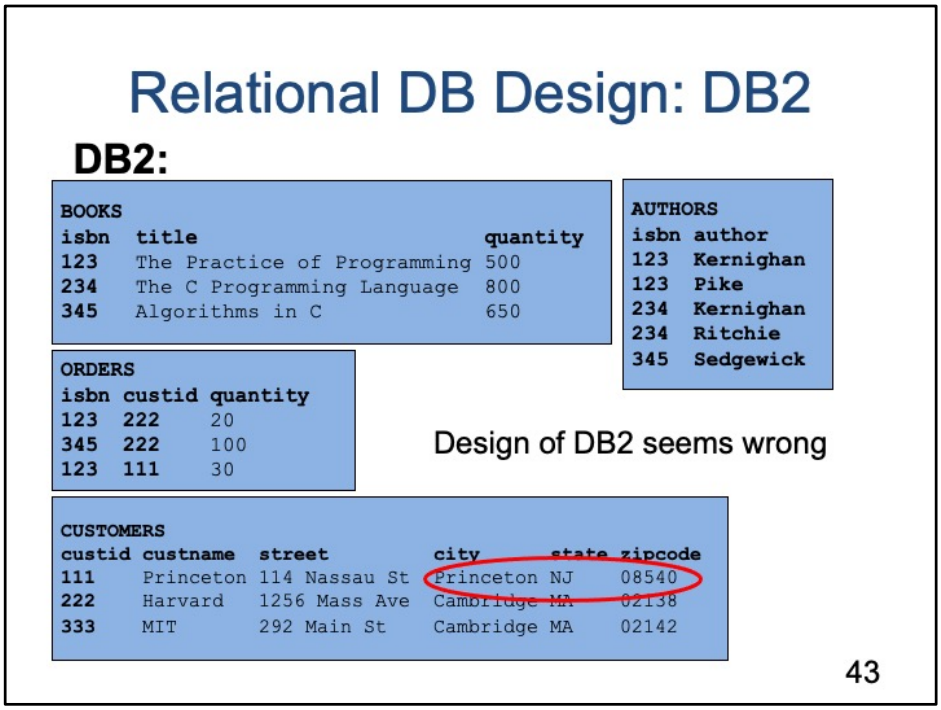
Relational DB Design: DB2

*Recall: A table is in **second normal form** iff:*

It is in first normal form, and

Every non-primary-key column is dependent on the entire primary key

Note that DB2 is in second normal form



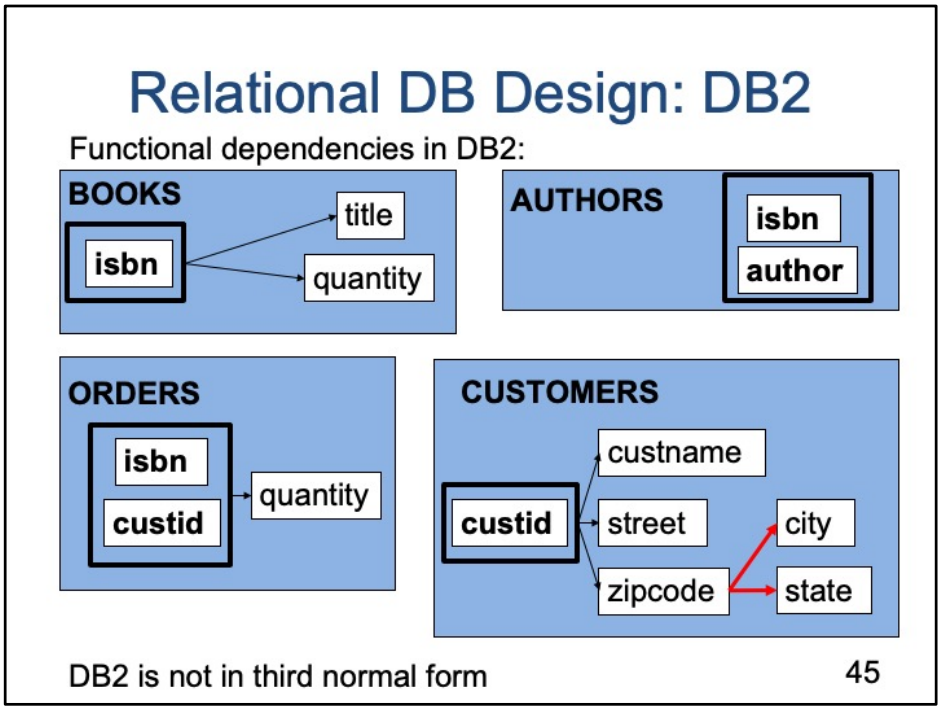
Relational DB Design: DB2

[see slide]

Intuitively: Still not quite right.
The fact that zip code 08540 maps to Princeton, NJ could be stored multiple times

Relational DB Design: DB2

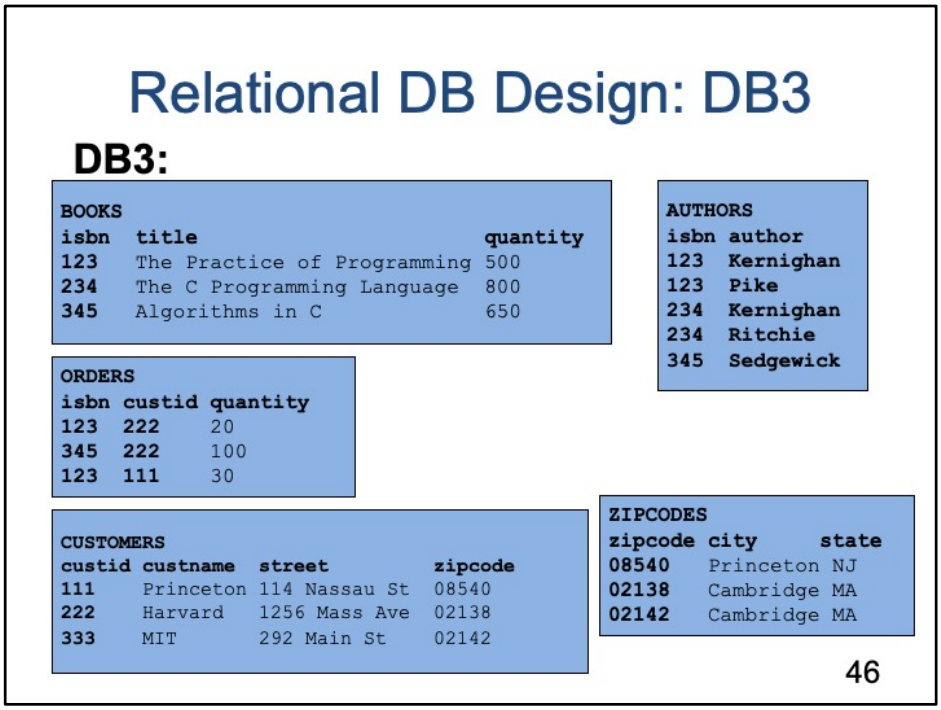
- Somewhat informally...
- A table is in ***third normal form*** iff:
  - It is in second normal form, and
  - Every non-primary-key column is non-transitively dependent on the primary key



Relational DB Design :DB2

*Recall: A table is in **third normal form** iff:*
It is in second normal form, and
Every non-primary-key column is non-transitively dependent on the primary key

City and state are functionally dependent on zipcode
Zipcode is functionally dependent on custid
So city and state are transitively functionally dependent upon custid



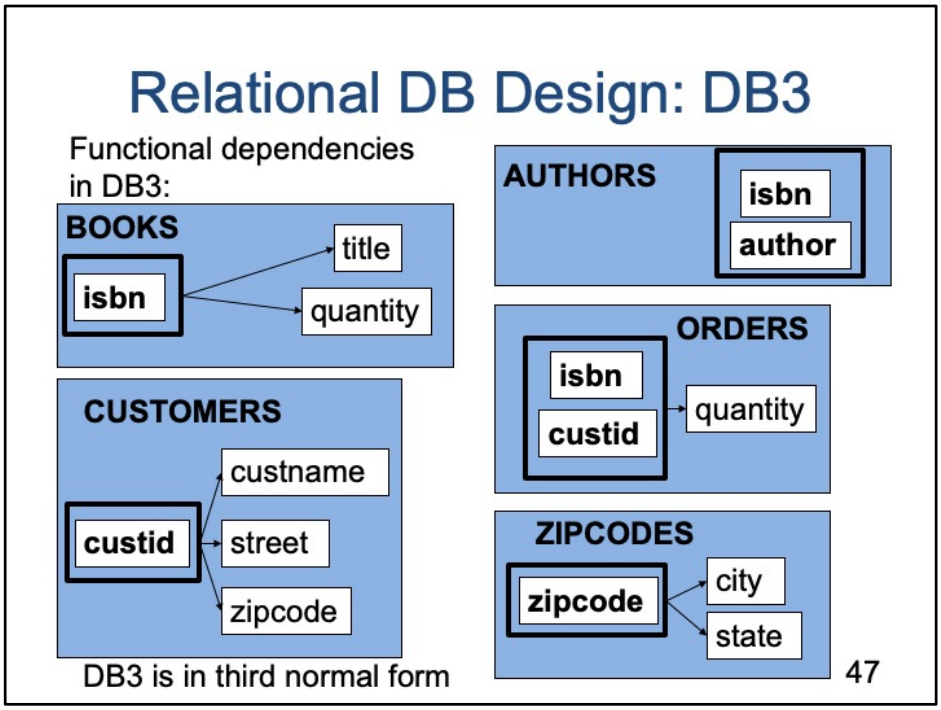
Relational DB Design: DB3

*Recall: A table is in **third normal form** iff:*
It is in second normal form, and
Every non-primary-key column is non-transitively dependent on the primary key

The solution: Split CUSTOMERS into CUSTOMERS and ZIPCODES

[see slide]

Yields DB3



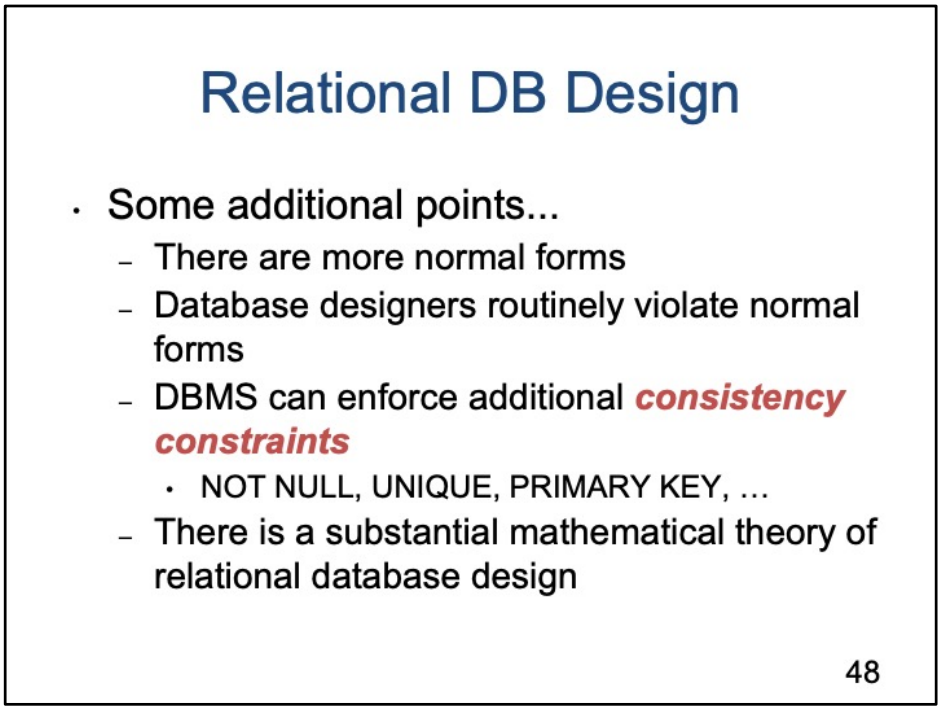
Relational DB Design: DB3

*Recall: A table is in **third normal form** iff:*
It is in second normal form, and
Every non-primary-key column is non-transitively dependent on the primary key

[see slide]

Note that DB3 is in third normal form

We've now incrementally designed the DB that we used in previous lectures



Relational DB Design

Some additional points to wrap up…

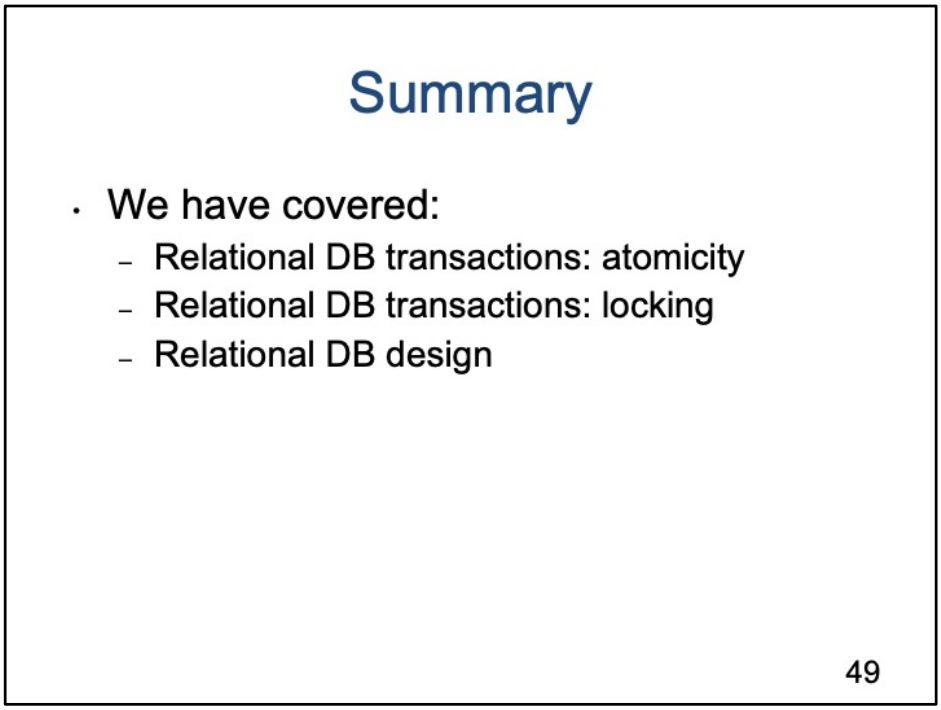
There are more normal forms
See Date book

Database designers routinely violate normal forms
But a good one does so:
Only with a purpose
Knowing the consequences

DBMS can enforce additional ***consistency constraints***; e.g.:
NOT NULL: The values in this column cannot be NULL'
UNIQUE: The values in this column must be unique
PRIMARY KEY: This column is part of the primary key; implies NOT NULL and UNIQUE across multiple columns

There is a substantial mathematical theory of relational database design
More precise definitions of normal forms
Several additional normal forms
Relational algebra, relational calculus
Foreign keys, integrity rules

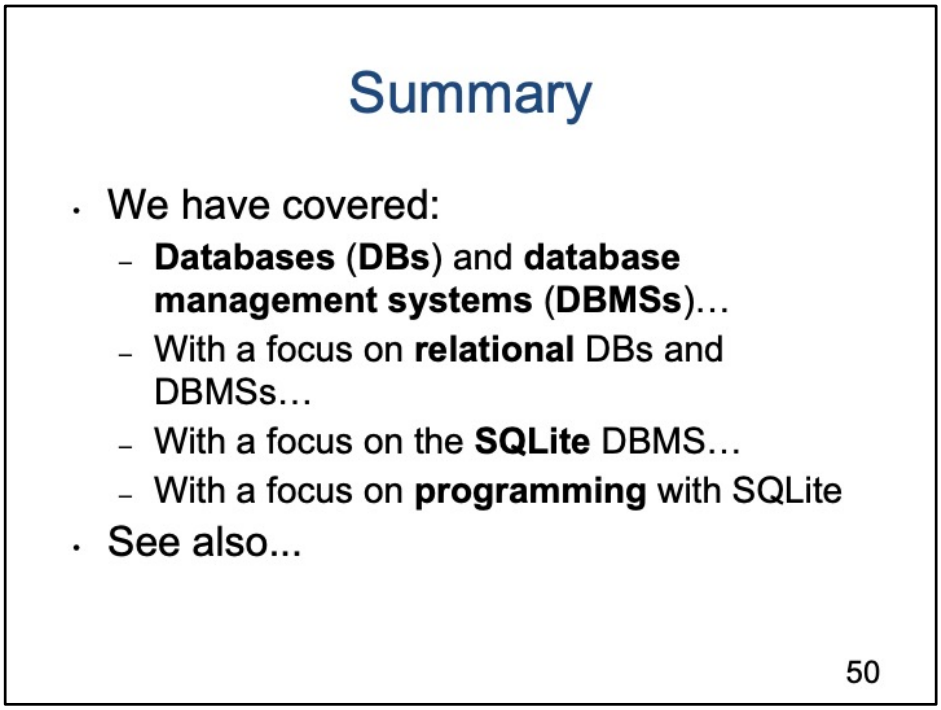
...
See *An Introduction to Database Systems* (C. J. Date)



Summary

In Database Programming (Part 3) we have covered:

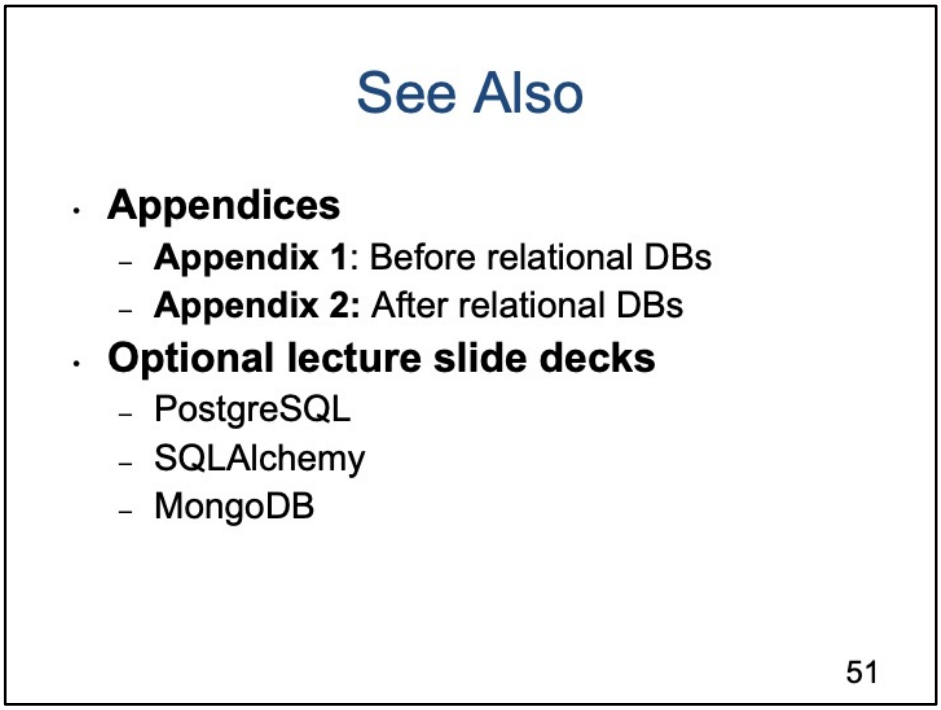
[see slide]



Summary

In the 3 Database Programming lectures we have covered…

[see slide]



See Also

Appendices
- Before relational DBs
  - A look backward through time at the DB field
- After relational DBs
  - An overview of some more recent developments in the DB field

Optional database lecture slide decks

- Available via Topics web page
- Cover some popular database-related technologies that are:
  - Not used in COS 333 assignments
  - Often used in COS 333 projects

- PostgreSQL
  - A full-fledged relational DBMS
  - In contrast with SQLite…
    - Runs as a server
    - Provides authentication
    - Row-level locking
    - Preferred by the Heroku web service
  - Good choice for many COS 333 projects

SQLAlchemy

- An object-relational mapper (ORM)
- Allows you to use a relational DBMS without using SQL
- Maps relational DB tables to classes, and rows to objects of that class
- Used by many COS 333 projects

MongoDB

- A popular non-relational (noSQL) DBMS
- Used by some COS 333 projects

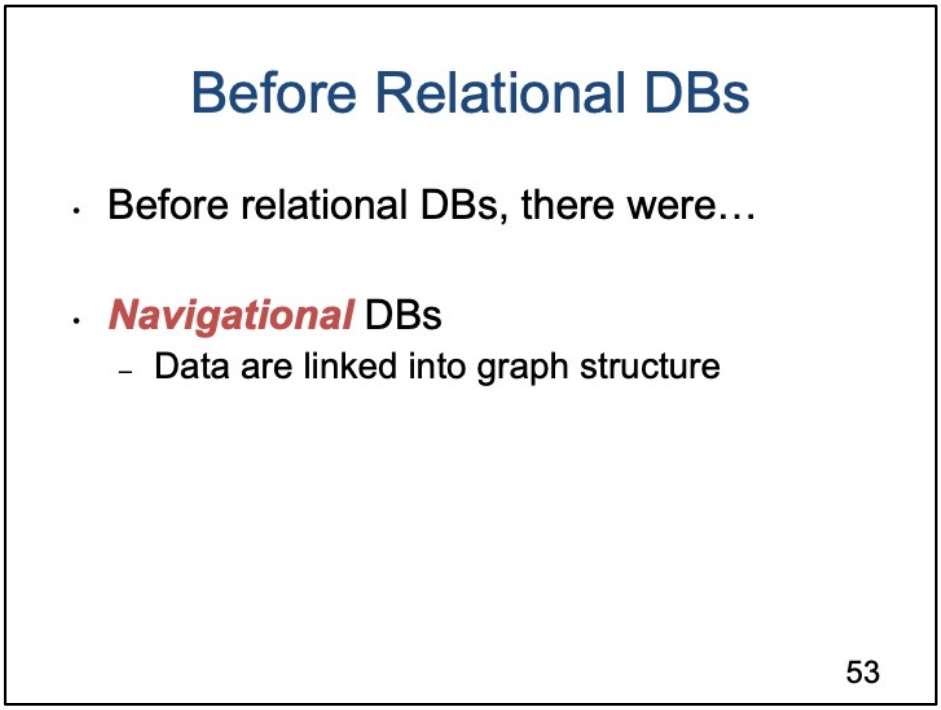
Goal

- Give you enough information about those technologies to:
  - Help you decide if you want to use them in your project
  - Help you get started with them

Suggestions

- View **quickly** now
  - To understand the general concepts
  - So you know what material is given, in case you need it later
- View **thoroughly** later
  - As needed

Appendix 1:
Before Relational DBs



Before Relational DBs

Before relational DBs, there were…

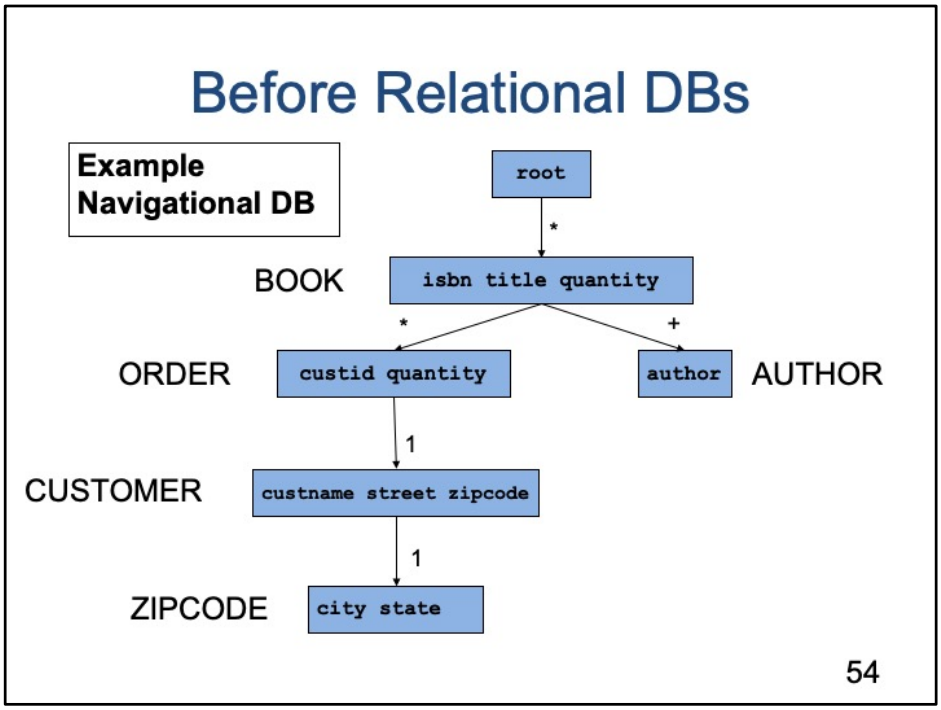
Navigational DBs

Data are **linked** into graph structure

Client is given “root” node

Client follows links to desired data

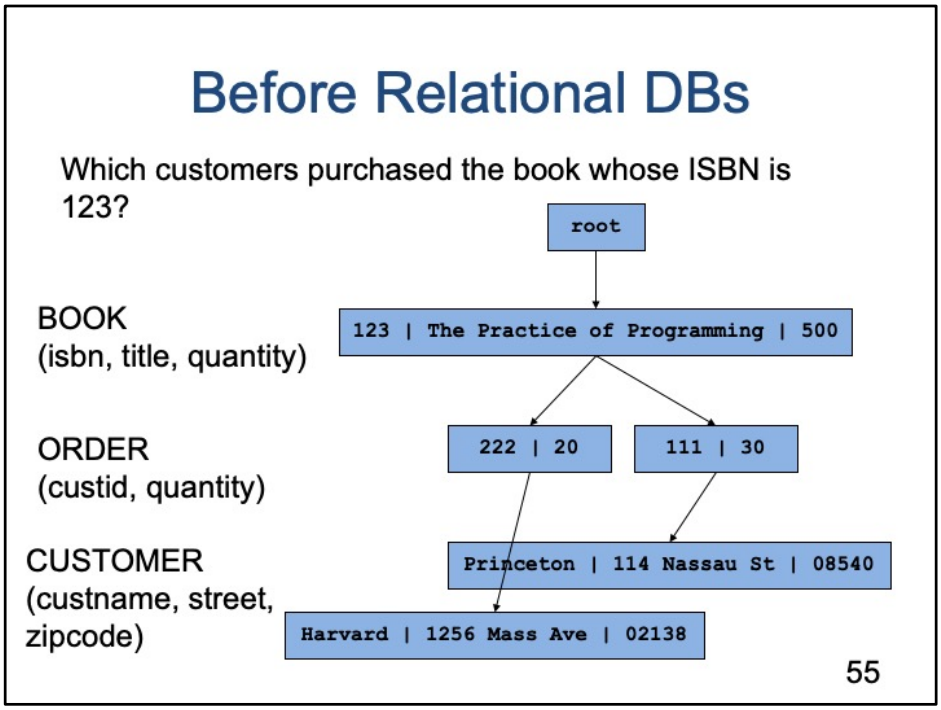
Example…



Before Relational DBs

[see slide]

The root of the graph points to 0 or more BOOK records
Each BOOK record contains an isbn, a title, and a quantity
Each BOOK record points to 0 or more ORDER records
Each ORDER record contains a customerid and a quantity
Each ORDER record points to one CUSTOMER record
Each CUSTOMER record contains a custname, street, and zipcode
Each CUSTOMER record points to one ZIPCODE record
Each ZIPCODE record contains a city and state
Each BOOK record points to one or more AUTHOR node
Each AUTHOR record contains an author (name)

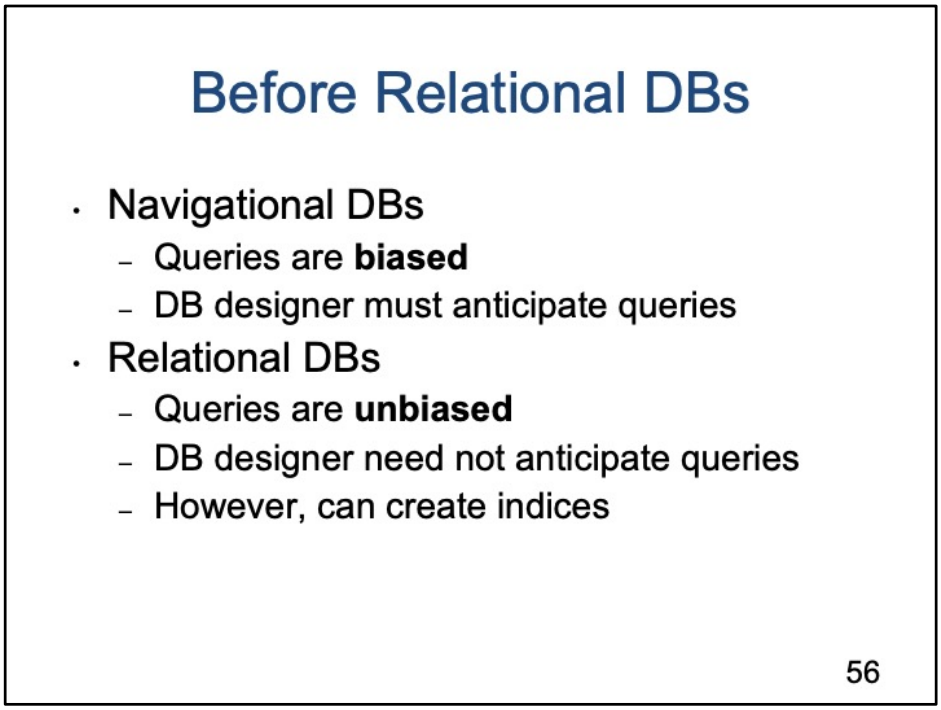


Before Relational DBs

[see slide]

Given a book, it's easy to find the customers who purchased that book

However, given a customer, it would be hard to find the books purchased by that customer



Before Relational DBs

Navigational DBs

Queries are **biased**

DB designer must anticipate queries to create appropriate links

Relational DBs

Queries are **unbiased**

That was Codd's original motivation

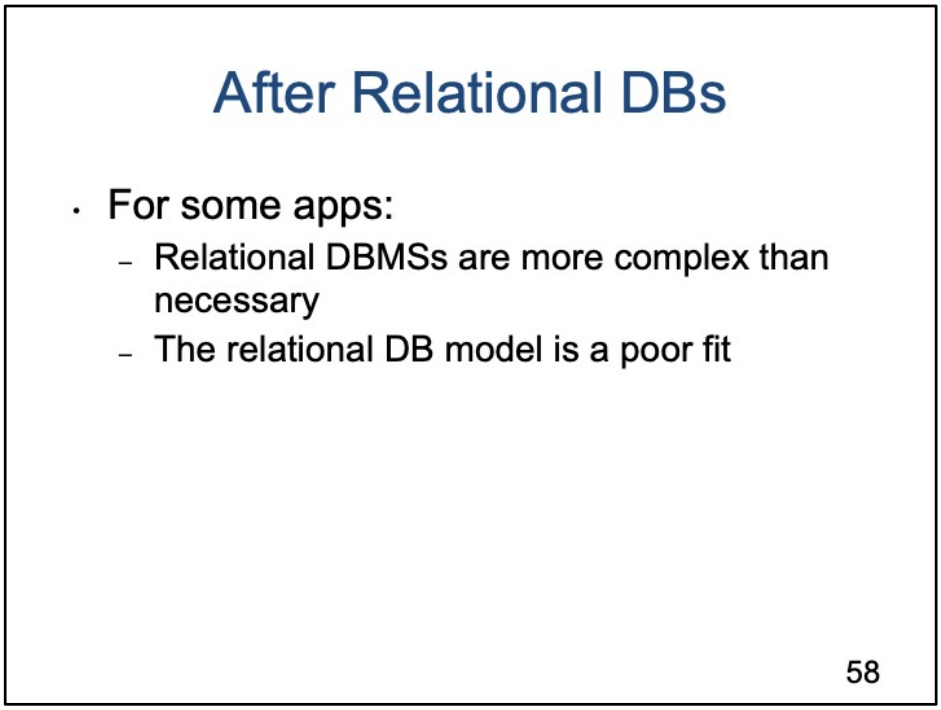
DB designer need not anticipate queries

However, can create indices

DBA can create indices, based upon *anticipated* usage patterns

DBMS can create indices, based upon *observed* usage patterns

Appendix 2:
After Relational DBs



After Relational DBs

Observations:

For some apps, relational DBMSs are more complex than necessary

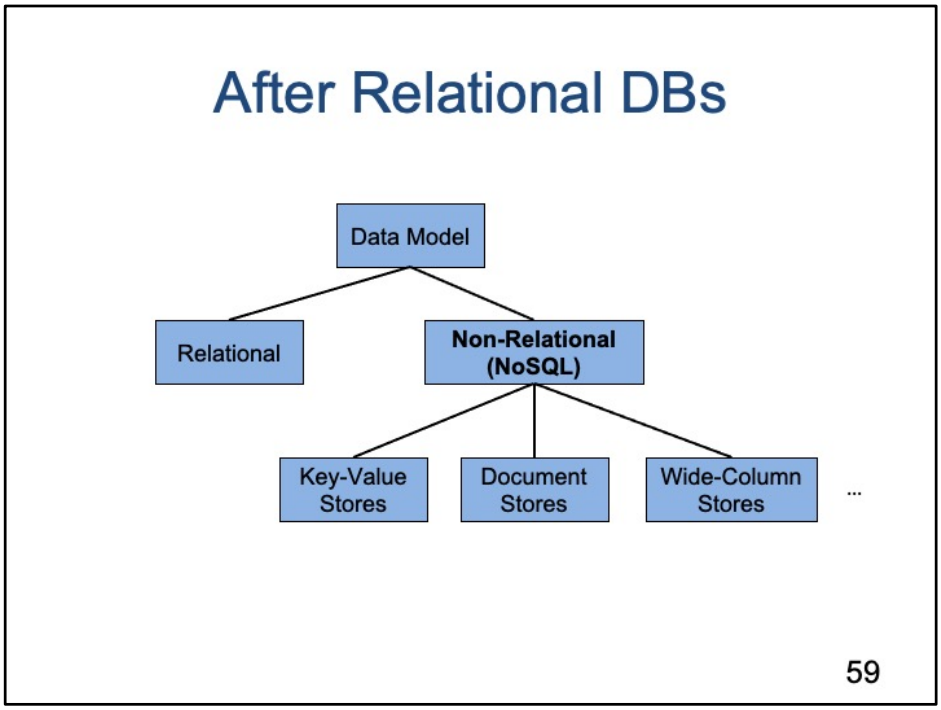
Many apps don't need query neutrality, transactions, …

For some apps, the relational DB model (tables, rows) is a poor fit

Example: apps that use data that is hierarchical to undefined levels

~~It's not the case that one size fits all~~

One size does not fit some

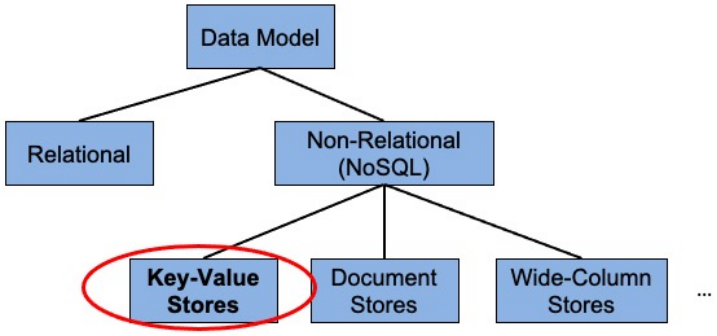


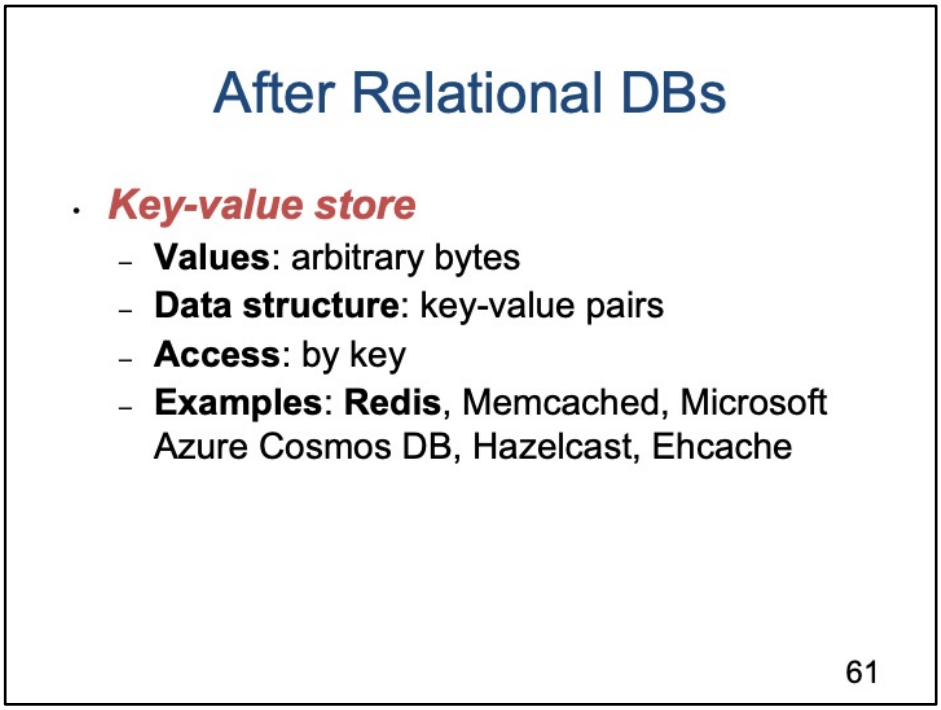
After Relational DBs

Those observations motivate the non-relational (“NoSQL”) data models

Note that there are three kinds…

After Relational DBs





After Relational DBs

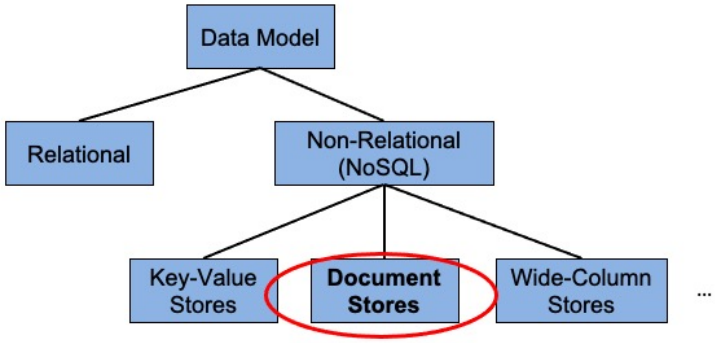
Key-value store

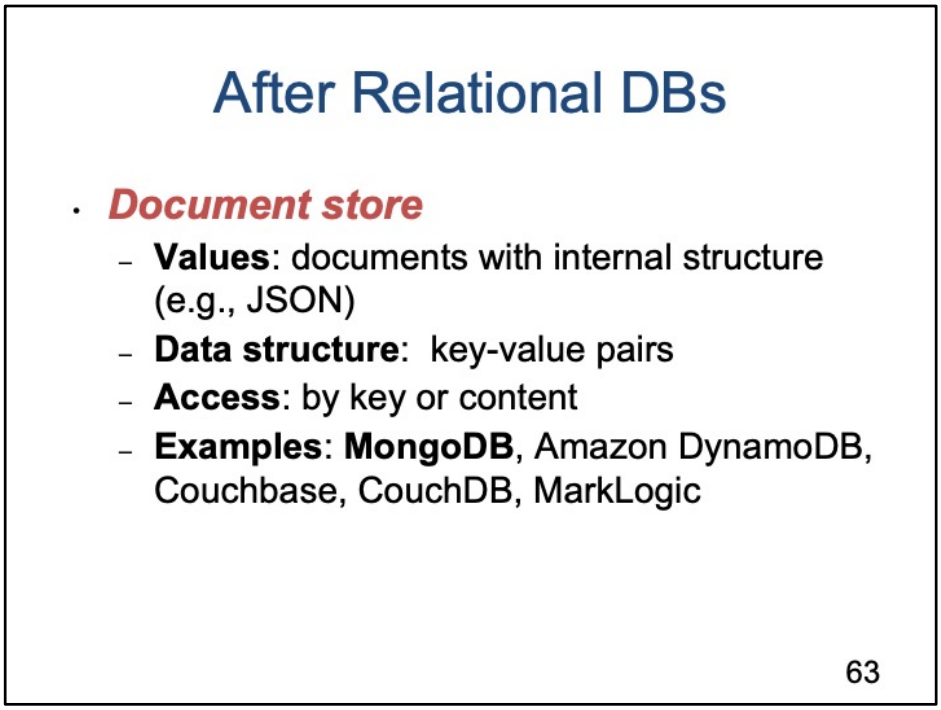
Values: Arbitrary bytes
Data structure: key-value pairs
Access: by key
Examples: **Redis**, Memcached, Microsoft Azure Cosmos DB, Hazelcast, EhcacheValue Stores

Essentially a key-value store database is a persistent associative array (hash table or search tree)
Lookup is strictly by key

Very simple; very fast

After Relational DBs





After Relational DBs

Document store

Values: Documents with internal structure (e.g. JSON) which DBMS can process

Data structure: key-value pairs

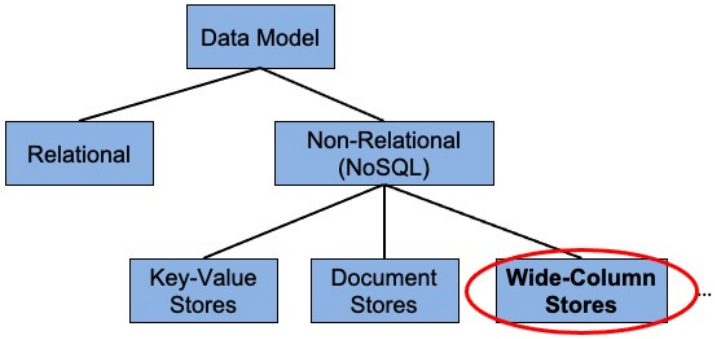
Can access document by key or sometimes by content

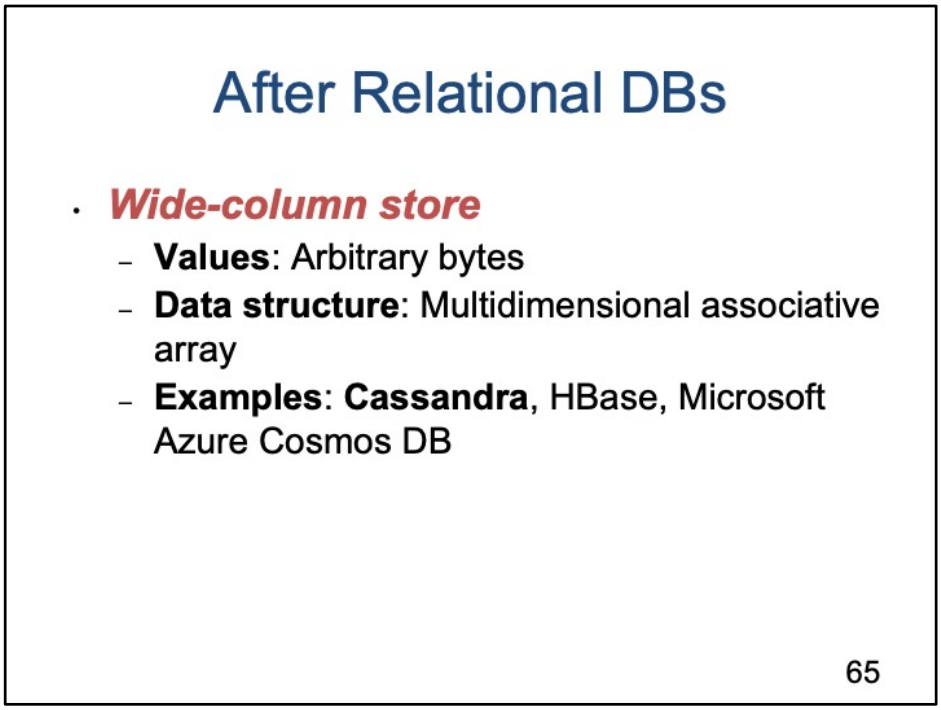
Access: by key or content

Examples:

MongoDB, Amazon DynamoDB, Couchbase, CouchDB, MarkLogic

After Relational DBs





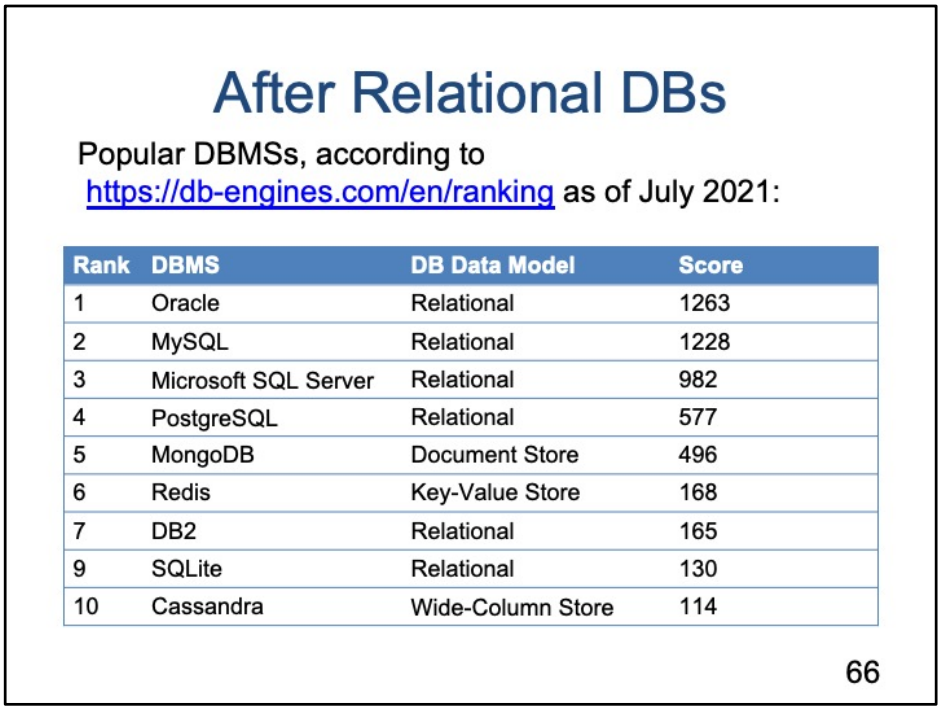
After Relational DBs

Wide-column store

Values: Arbitrary bytes
Data structure: Multidimensional associative array
Typically sparsely populated
Examples:
Cassandra, HBase, Microsoft Azure Cosmos DB

I haven't used

Origin: Google BigTable



After Relational DBs

[see slide]

Notes

Relational data model still dominates, by a wide margin
PostgreSQL is the preferred DBMS when deploying on Heroku
SQLite is in the top 10!