COSC 460 Lecture 22: No SQL Overview

Instructor: Michael Hay
Fall 2018

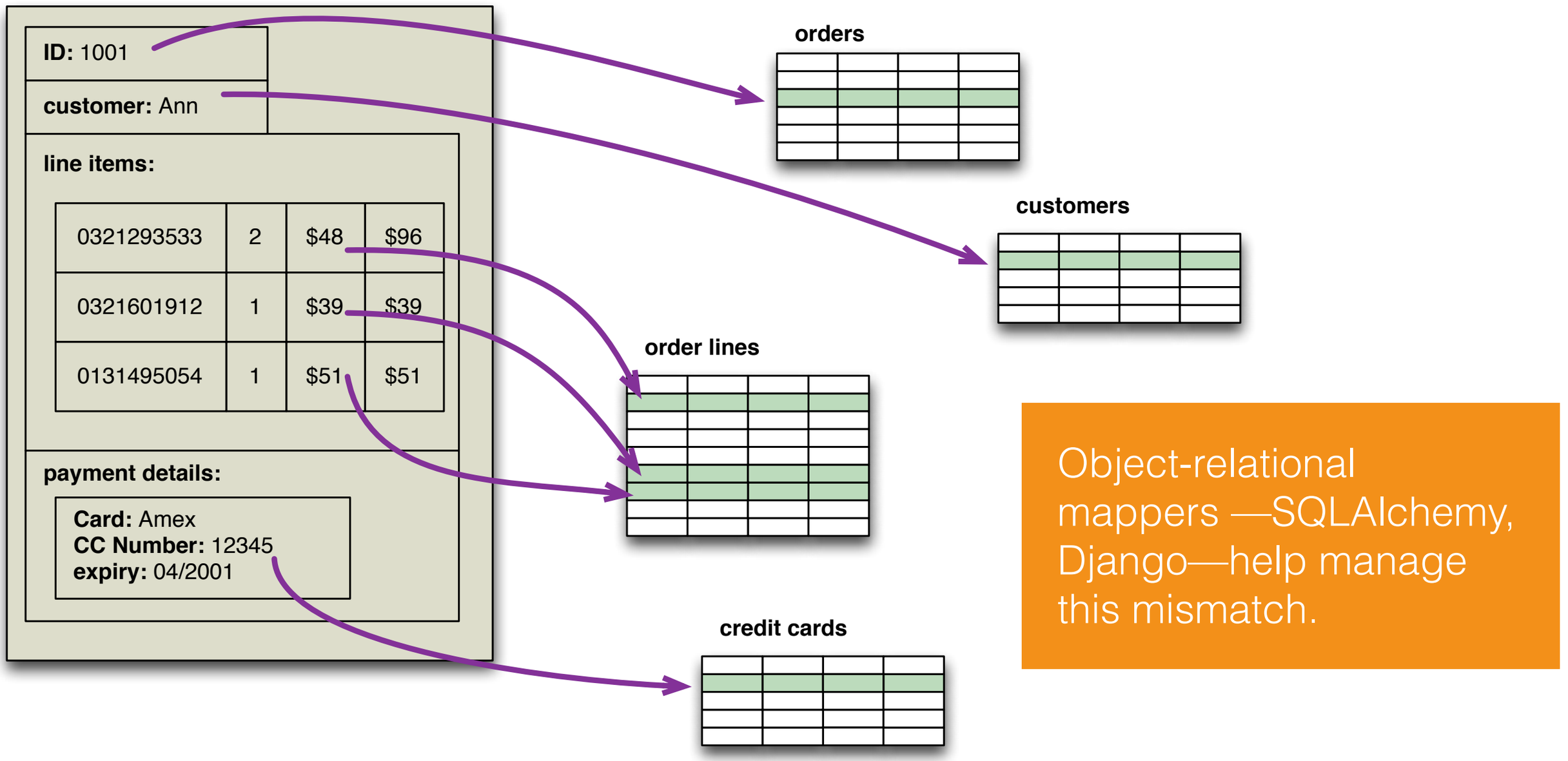
No SQL

- Name is historical accident
- Name for SF meetup in 2009 to discuss emerging database technologies
  - Google's BigTable 2006 (→ open-source HBase)
  - Amazon's Dynamo 2007 (→ open-source Riak)

Driving forces

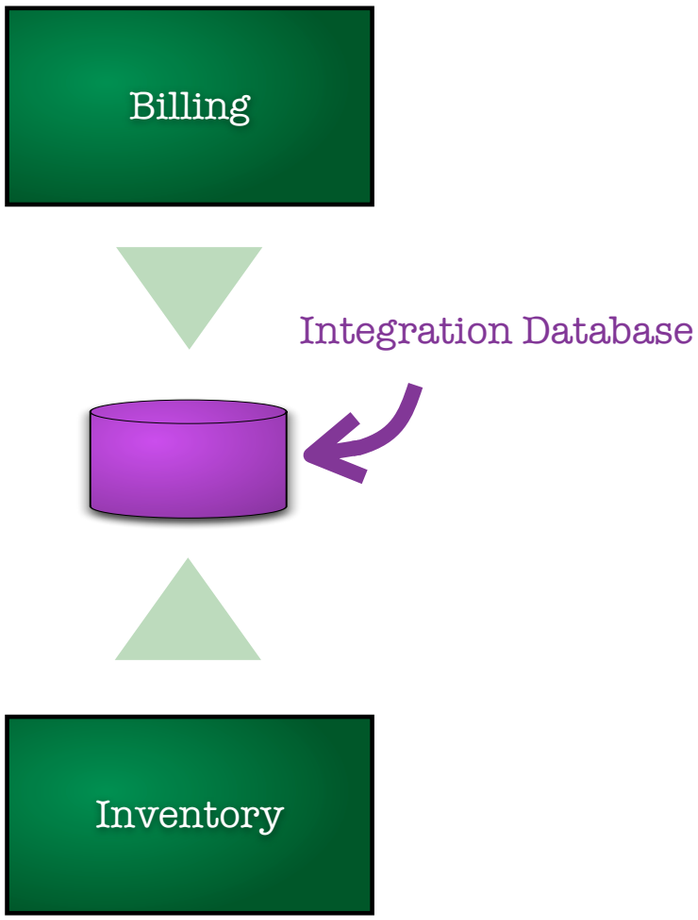
- Impedance mismatch
- Role of database: integrator vs. app backend
- Scaling out

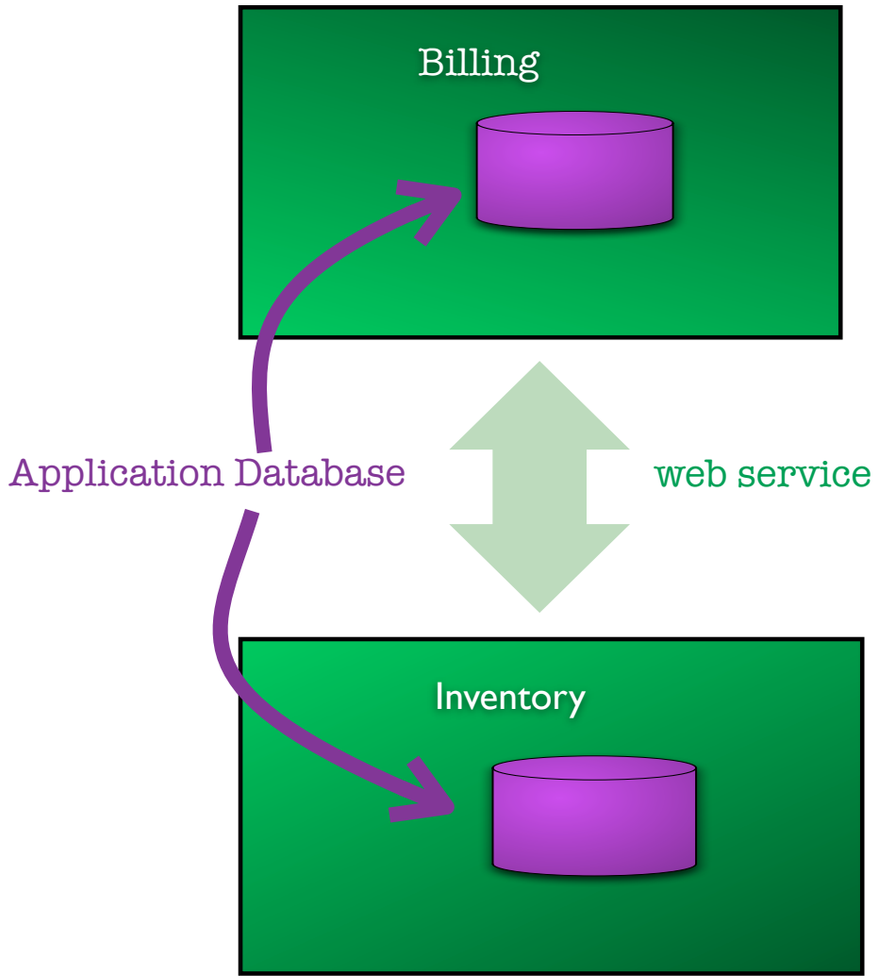
Impedance mismatch



Object-relational mappers —SQLAlchemy, Django—help manage this mismatch.

Application vs. Integration Database

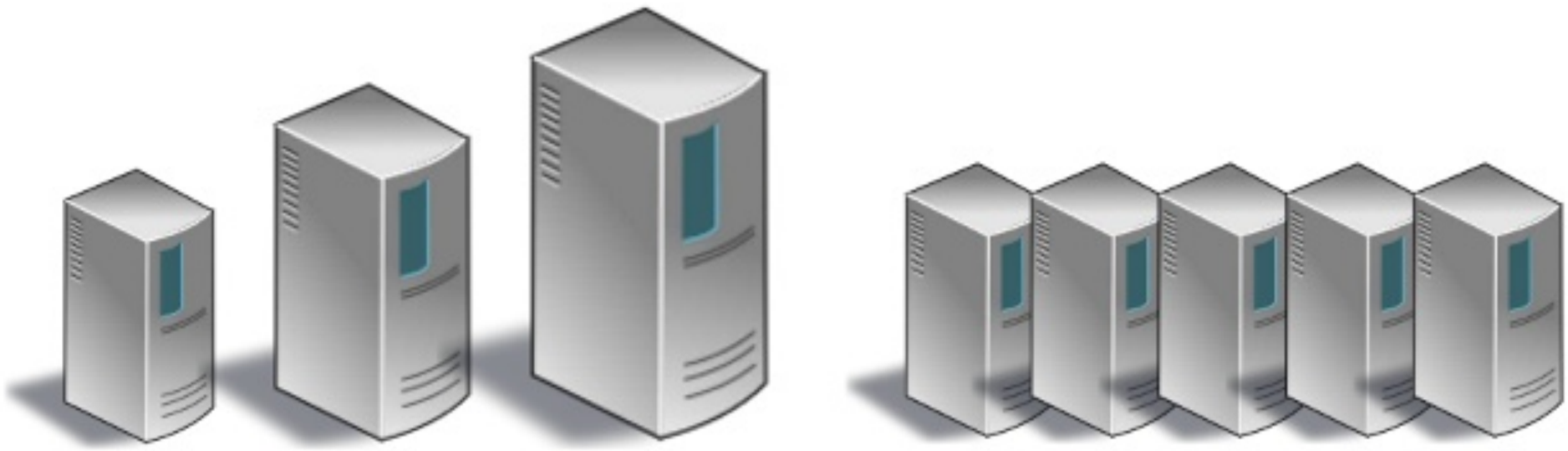




SCALE UP VS SCALE OUT

Scale Up – add bigger resources

Scale Out – add more of the same



Source: https://codopia.wordpress.com/2017/10/28/
capacity-planning-and-scaling-the-azure-function-apps/

Characteristics of “No SQL”

- Not using relational model
- Designed for web services
- Designed to scale out on clusters
- Open-source
- “Schemaless”

Data Models

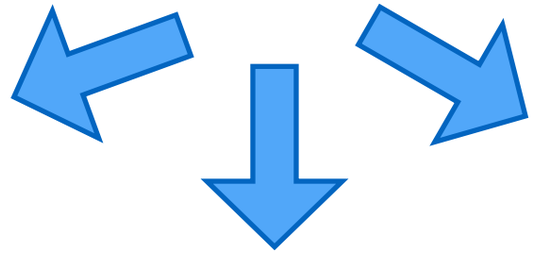
Relational

- Familiar to us
- Important properties *(for NoSQL discussion)*
  - No nesting: attribute values must be simple data types (string, int, double)
  - Relations are *normalized* to eliminate redundancy

Course Offerings

cid	title	prof_first	prof_last	offering	location
290	Discrete Structures	Philip	Mulry	Fall 2018	314 McGregory
290	Discrete Structures	Michael	Hay	Spring 2018	315 McGregory
290	Discrete Structures	Michael	Hay	Fall 2017	315 McGregory
290	Discrete Structures	Vijay	Ramachandran	Spring 2017	315 McGregory

normalize



Courses

cid	title
290	Discrete Structures

Professors

pid	prof_first	prof_last
1	Philip	Mulry
2	Michael	Hay
3	Vijay	Ramachandran

Offerings

cid	pid	offering	location
290	1	Fall 2018	314 McGregory
290	2	Spring 2018	315 McGregory
290	2	Fall 2017	315 McGregory
290	3	Spring 2017	315 McGregory

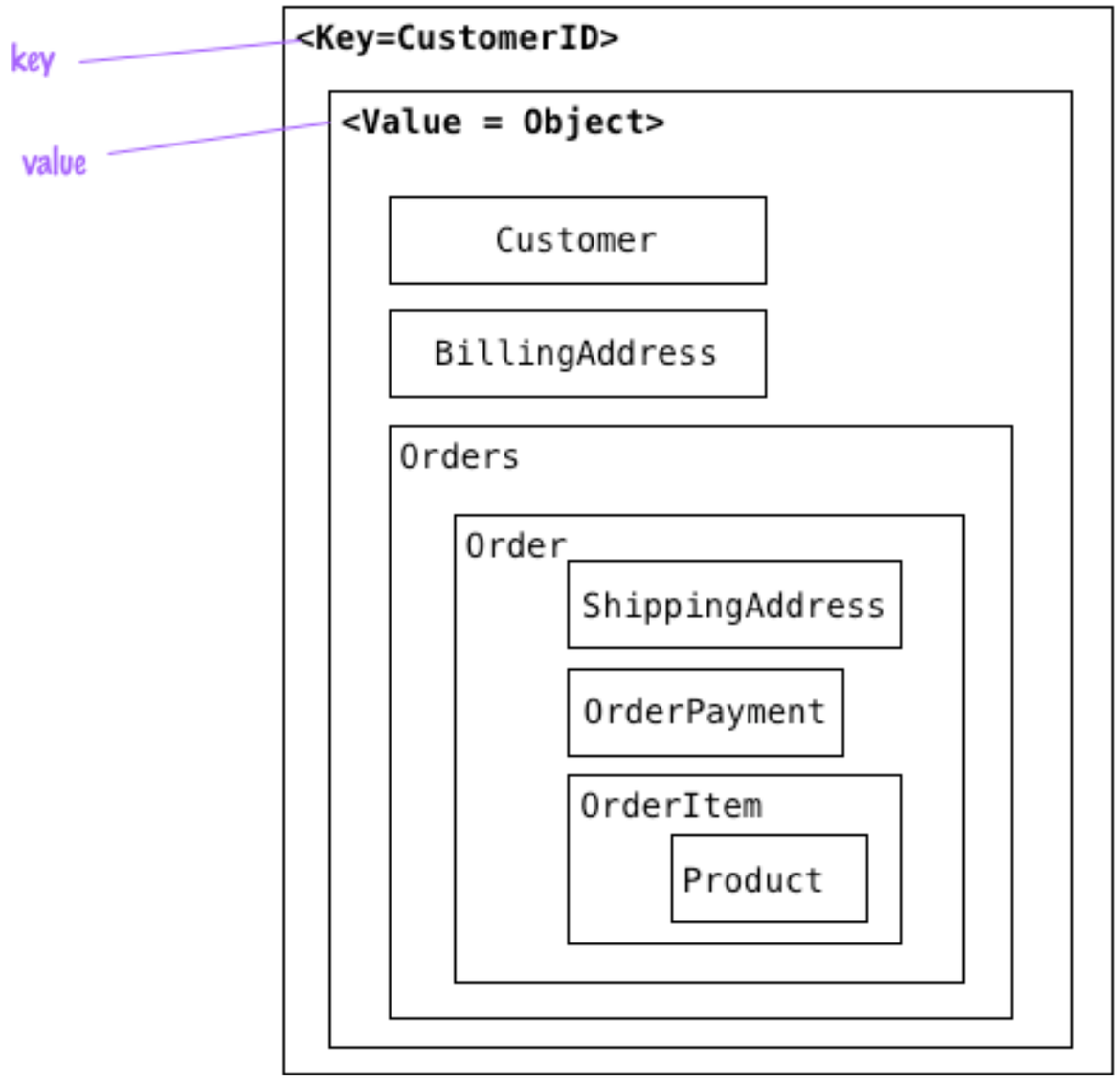
Aggregates

- An *aggregate* is collection of data that want to treat as a unit

- Aggregate can have complex, nested structure

- Data is often denormalized (same fact may appear in multiple aggregates)

Aggregate model



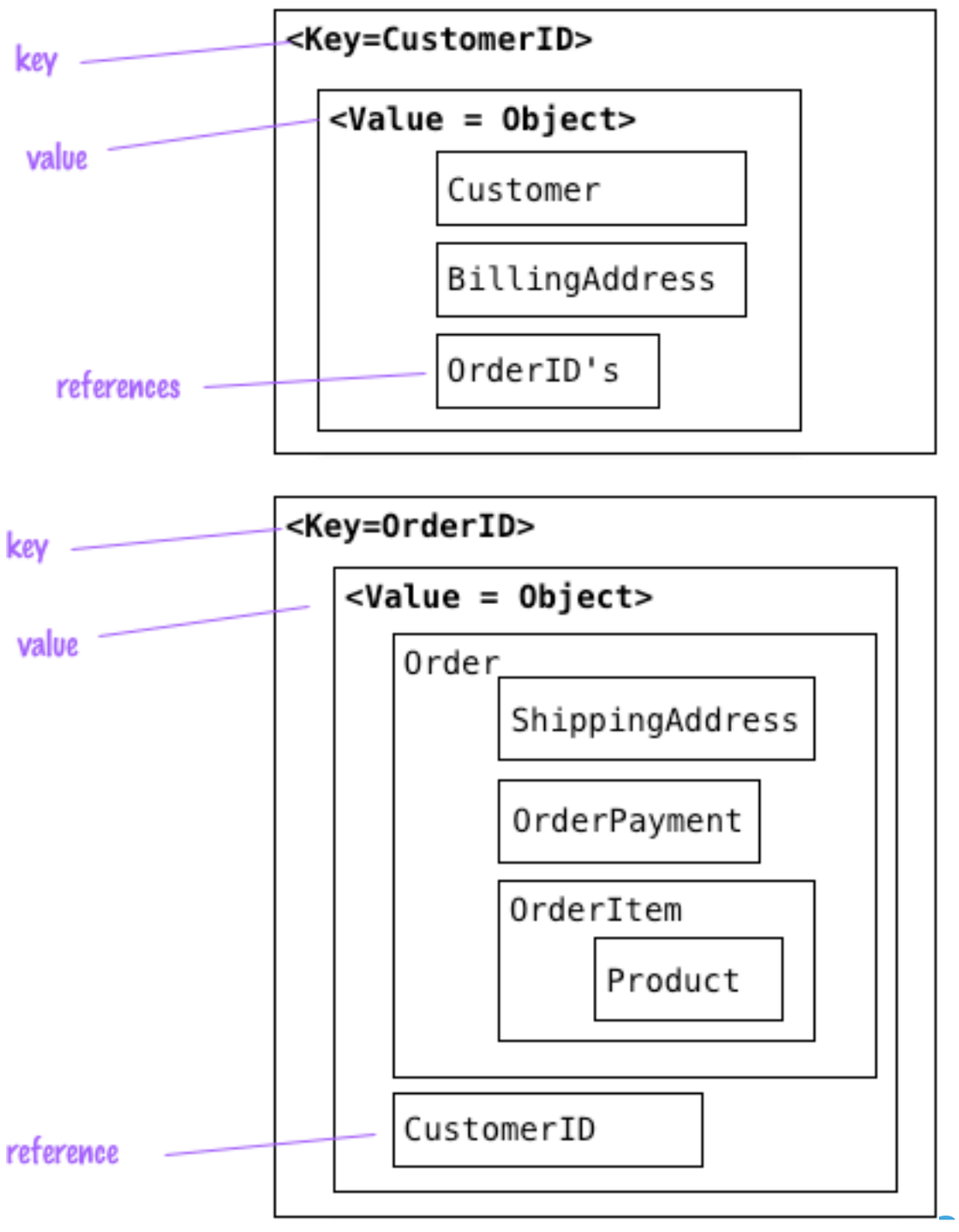
Order is *embedded* inside customer aggregate

Aggregate data

```
    // in customers
{
  "customer": {
    "id": 1,
    "name": "Martin",
    "billingAddress": [{"city": "Chicago"}],
    "orders": [
        {
          "id":99,
          "orderItems":[
            {
              "productId":27,
              "price": 32.45,
              "productName": "NoSQL Distilled"
            }
          ],
          "shippingAddress":[{"city":"Chicago"}]
          "orderPayment":[
            {
              "ccinfo":"1000-1000-1000-1000",
              "txnId":"abelif879rft",
              "billingAddress": {"city": "Chicago"}
            }
          ],
      }
  ]
}}
```

Do you see examples of denormalized data?

Alternative aggregation



Two types of aggregates — linked by *references*.

```
// in Customers
{
   "id":1,
   "name":"Martin",
   "billingAddress":[{"city":"Chicago"}]
}
// in Orders
{
   "id":99,
   "customerId":1,
   "orderItems":[
     {
       "productId":27,
       "price": 32.45,
       "productName": "NoSQL Distilled"
     }
   ],
   "shippingAddress":[{"city":"Chicago"}]
   "orderPayment":[
     {
       "ccinfo":"1000-1000-1000-1000",
       "txnId":"abelif879rft",
       "billingAddress": {"city": "Chicago"}
     }
   ],
}
```

Aggregate-oriented databases

- Treats an aggregate as unit and supports updates to units
- Why aggregates?
  - Makes it easier to distribute data storage across cluster
  - Collocate related data into single aggregate
- Work best when most data interaction is on same aggregate
- Inter-aggregate relationships are difficult to handle!

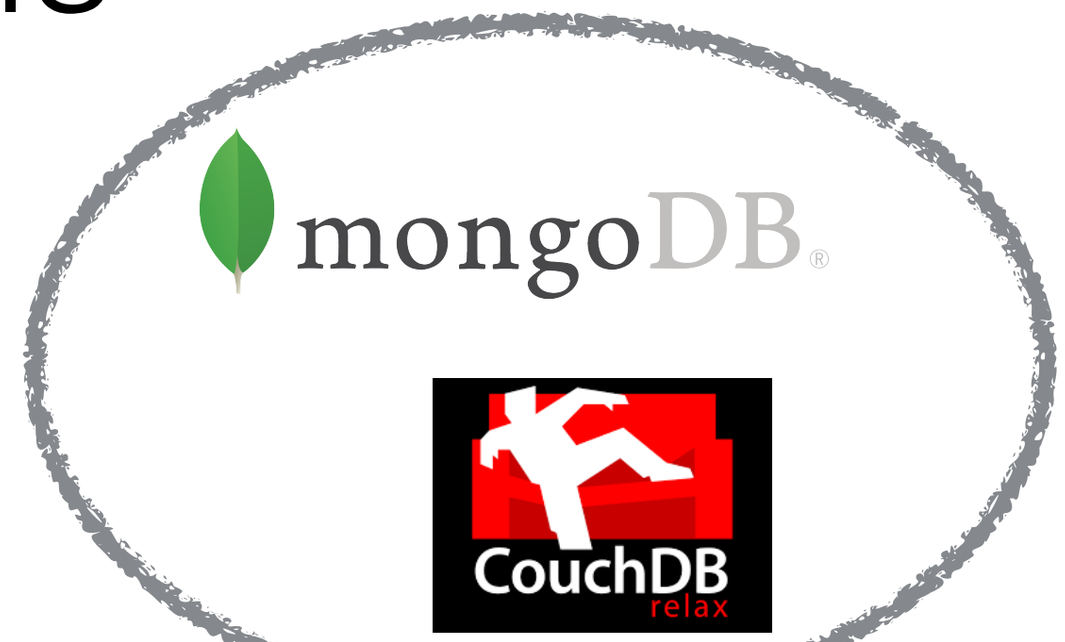
Aggregate choice

- Context: data about courses
- What is the “right” aggregate?
  - Professor aggregate: list of courses taught, students in those courses
  - Student aggregate: list of courses taken, professor of those courses
  - Multiple aggregate types… students, professors, courses, offerings…
- Suppose…
  - … facilities wants to see which classrooms are under utilized
  - … professor wants to (atomically) swap lab sections between pair of students

Aggregate-oriented data models

Boundaries are fuzzy

Document



Key-value

Column-family

(Graph DBs are not aggregate-oriented.)

“Schemaless”

- Sales pitch: Flexibility! 😀
- Truth:
  - Implicit schema is encoded in app code ☹️
  - Flexibility is *within* aggregate only — changing aggregates is painful! 🤯

Scaling out

- **Sharding**: divide collection of units across multiple machines (why?)

- **Replication**: copy *same* unit multiple places (why?)

Consistency

- Two distinct consistency problems arise:
  - Logical — updates involving multiple units
  - Replication — propagate update of single unit to all its replicas

Logical consistency

- Aggregates reduce need for full blown transaction support
  - Aggregate updates are designed to be atomic, durable
  - Since most database interactions update a single aggregate, this is all you need.
- Updates that involve multiple aggregates are *harder to handle.* Solutions: offline locks, version stamps, ….

Replication consistency

- Same data item appears multiple places
- If you write, make sure change gets propagated to others
- If you read, make sure you have most recent update.
- You don't have to talk to every replica: *quorums*

CAP Theorem

- **Consistency**: a read guaranteed to return the most recent write for a given client.

- **Availability**: non-failing node will return reasonable response within reasonable time.

- **Partition Tolerance**: the system will continue to function when network partitions occur.

- **Theorem** (informal): in distributed setting, you can't have both consistency and availability, must choose!

Presentations

- Data model
- Scaling
- Querying: individual aggregate, across aggregate
- Consistency: logical, replication
  - What sort of consistency does it achieve?
  - How does it do it?