NoSQL Databases

CSCI 220: Database Management and Systems Design

Slides adapted from
Simon Miner
Gordon College

Practice Quiz: Database Architectures

- Discuss with a neighbor:
  - The pros and cons of hosting a database on multiple servers
  - Describe the differences between:
    - Horizontal fragmentation
    - Vertical fragmentation
    - Replication

Today you will learn…

- About alternatives to relational databases

Agenda

- Emergence of NoSQL
- NoSQL Databases
  - Graph databases
  - Aggregate databases: key-value, document, and wide-column store
  - Column-oriented
- Related Topics
  - Distributed Databases and Consistency with NoSQL
  - Schema Migrations
  - Polyglot Persistence
  - When (not) to use NoSQL

Emergence of NoSQL

Pros and Cons of Relational Databases

- Advantages:
  - ACID: Atomicity, Consistency, Isolation, Durability
    - Transactions, crash recovery, concurrency control, etc.
  - Integration across multiple applications
  - (Mostly) standard model (i.e., tables and SQL)
- Disadvantages:
  - Impedance mismatch
  - Shift from integration DBs to application DBs
  - Older RDBMSs were not designed for clustering
    - Counterexamples: Distributed RDBMSs like CockroachDB, Google Spanner, etc.

Impedance Mismatch

- Different representations of data when it is in the RDBMS vs in memory
  - In-memory data structures use lists, dictionaries, nested and hierarchical data structures
  - Relational DBs store atomic values (no lists or nested records)
  - Translating between these representations can limit developer productivity
- Object-relational mapping (ORM) can help
  - However, abstraction can lead to neglect of query performance tuning
- Caveat: Modern RDBMS's support JSON data

Impedance Mismatch Example

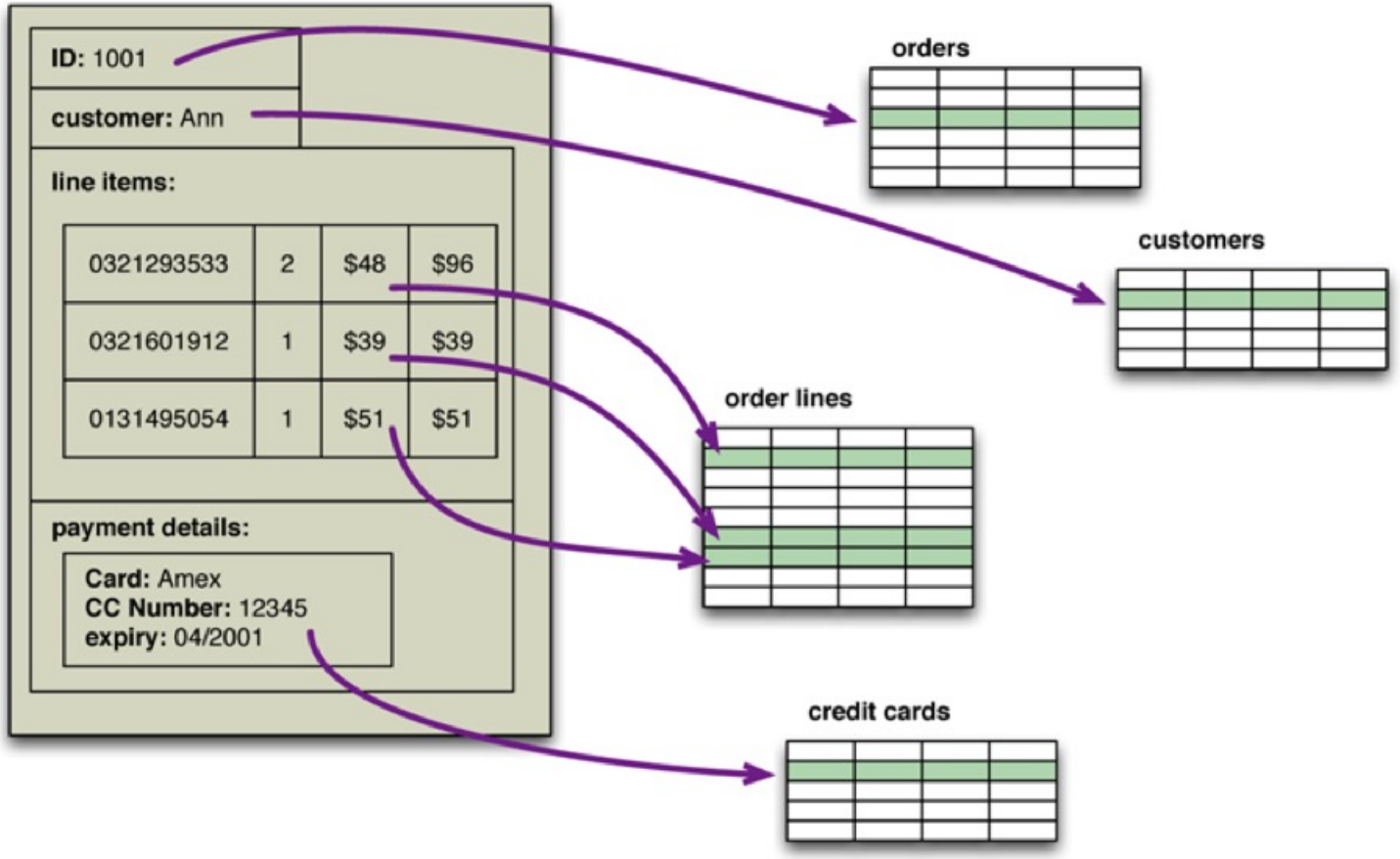


Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

Integration vs Application Databases

- Integration databases support multiple applications
  - Can be problematic if the applications have very different needs and are maintained by separate teams
    - Who maintains the database?
- SQL can be limiting as the only shared layer
  - Web services have become a more flexible alternative
- Application databases offer greater flexibility
  - Application is the only thing using the database
  - Access to the data is mediated through the application's API

The Need for Clusters

- The Internet created the need to store and process huge amounts of data
- Traditionally, relational databases could scale “up” (bigger machine) , but not “out” (many machines) as well
  - Disk subsystem was a single point of failure
  - Distributing/fragmenting/sharding data was complicated
  - High licensing costs for many database machines and CPUs
- Large web companies began developing their own alternative technologies to deal with these issues
  - Google’s BigTable and Amazon’s Dynamo
  - Issues addressed by these solutions have become relevant to smaller companies wanting to capture and analyze lots of data

The Emergence of NoSQL

- Ironically, the term "NoSQL" was first used as a name for an open source relational database released in the late 1990's
- Term as it is used today was a hastily-chosen Twitter hash tag for a conference meet-up on the topic in 2009
- No official general definition for *NoSQL*, but common characteristics include:
  - Does not use the relational model (mostly)
  - Driven by the need to run on clusters
  - Built for the need to run modern web properties
  - Schema-less
- More of a movement than a technology
  - Relational databases are not going away
  - *Polyglot persistence:* use the type of data store most appropriate for the situation

Future: Which DBMS should you choose for your application?

- Anecdote: "Relational databases can't scale as well as NoSQL"
  - Reality: Most applications don't need to scale beyond a single server. For applications that do, see CockroachDB, Google Spanner, etc.
- Anecdote: "Relational databases are harder to use than NoSQL"
  - Reality: Although relational DBs require you to define your schema, this will probably save you effort in the long-term. A defined schema also allows tools like the Django admin interface.
    - If you wanted to, you could just store JSON in a relational DB…
- **Knowledge of different DBMSs is necessary for an informed choice.** My opinion: some NoSQL products *often* meet a genuine need (e.g., Redis for caching, graph DBs for complex graph queries), while other products are *usually* not the best choice.

Graph Databases

Graph Databases

- Excel at modeling relationships between entities
- Terminology
  - *Node:* an entity or record in the database
  - *Edge*: a directed relationship connecting two entities
    - Two nodes can have multiple relationships between them
  - *Property*: attribute on a node or edge
- Graphs are queried via *traversals*
  - Traversing multiple nodes and edges is very fast
    - Because relationships are determined when data is inserted into the database
  - Relationships (edges) are persisted just like nodes
    - Not computed at query time (as in relational databases)

Graph Database Example

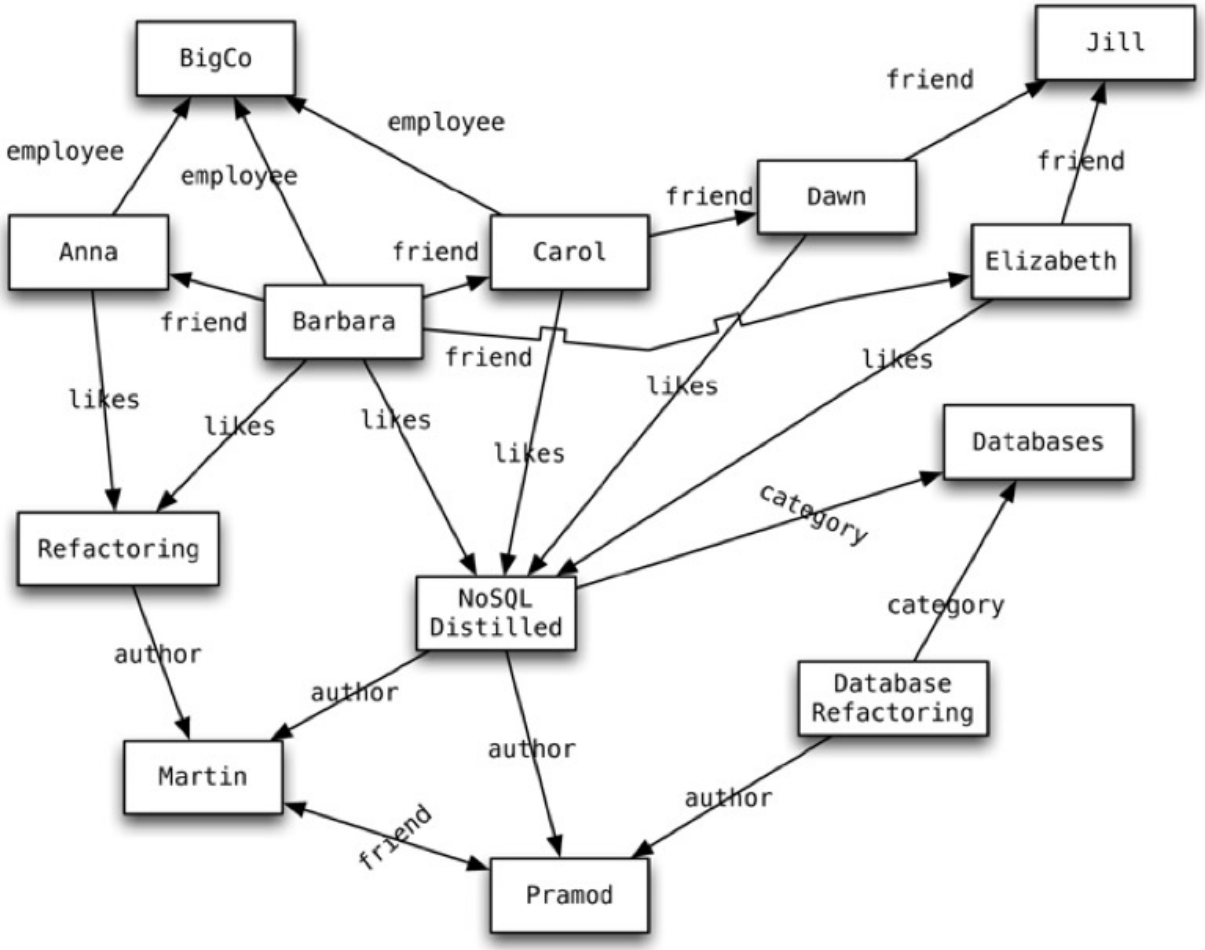


Figure 3.1. An example graph structure

Graph Database Example

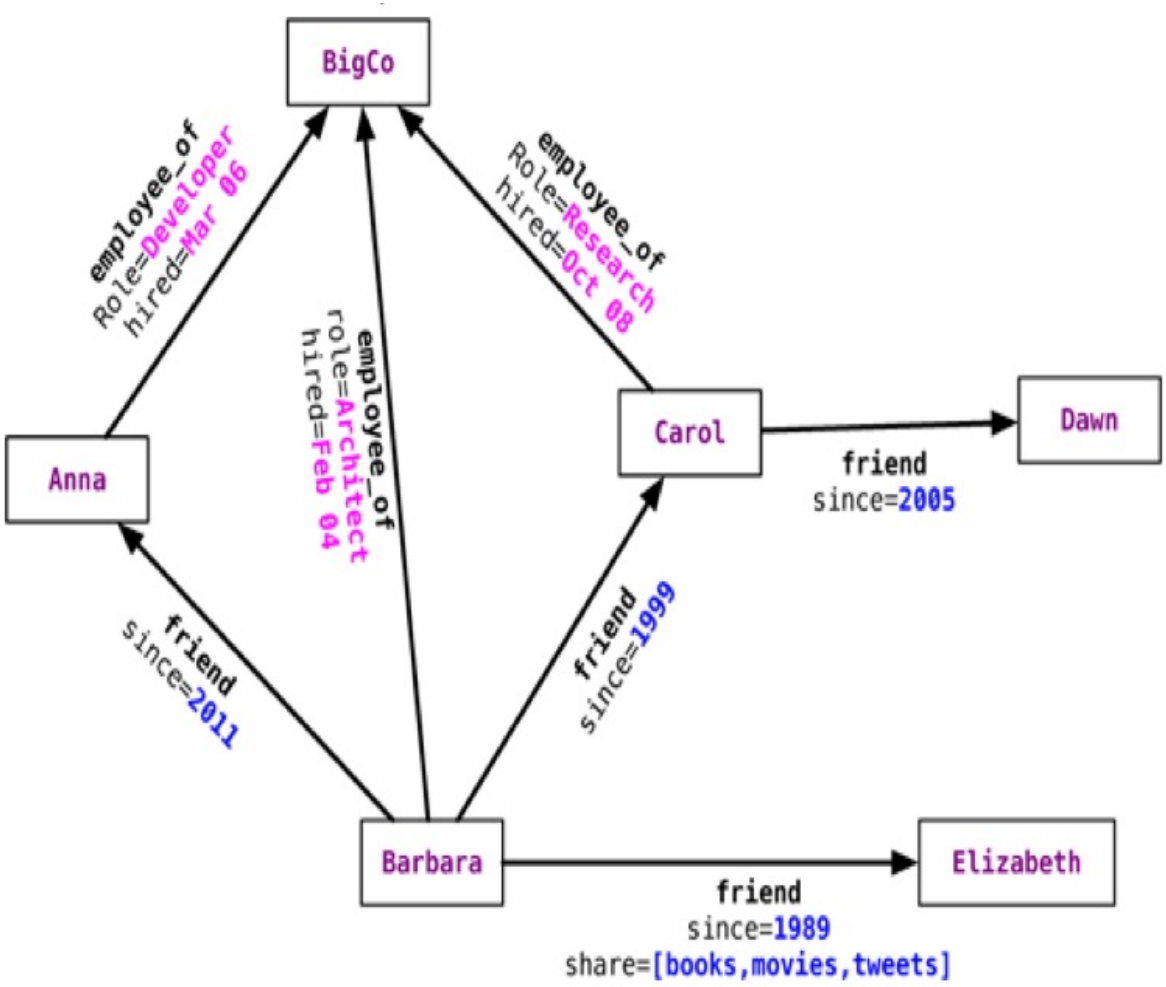


Figure 11.2. Relationships with properties

Graph Database Features

- Transaction support: graph can only be modified within a transaction
  - No “dangling relationships” allowed
  - Nodes can only be deleted if they have no edges connected to them
- Availability via replication
- Scaling via sharding is difficult since the graph relies heavily on the relationships between its nodes
  - Fragmentation can be done using domain knowledge (i.e. separating relationships by different geographic regions, categories, time periods, etc. – factors don’t get traversed much)
    - Traversal across shards is very expensive

Interacting with Graph Databases

- Web services / REST APIs exposed by the database
- Language-specific libraries provided by the database vendor or community

```
// Find the names of people who like NoSQL Distilled
Node nosqlDistilled = nodeIndex.get("name",
                                    "NoSQL Distilled").getSingle();
relationships = nosqlDistilled.getRelationships(INCOMING, LIKES);
for (Relationship relationship : relationships) {
    likesNoSQLDistilled.add(relationship.getStartNode());
}
```

- Query languages – allow for expression of complex queries on the graph
  - Gremlin with Blueprints (JDBC-like) database connectors
  - Cypher (for neo4j)

Graph Database Query Language Example

- A "select" statement in Cypher

```
START beginingNode = (beginning node specification)
MATCH (relationship, pattern matches)
WHERE (filtering condition: on data in nodes and relationships)
RETURN (What to return: nodes, relationships, properties)
ORDER BY (properties to order by)
SKIP (nodes to skip from top)
LIMIT (limit results)
```

- Find the names and locations of Barbara's friends
  - Cypher

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name,friend_node.location
```

  - Gremlin

```
g = new Neo4jGraph('/path/to/graph/db')
barbara = g.idx(T,v)[[name:'Barbara']]
friends = barbara.out('friend').map
```

Using Graph Databases

- Use graph databases for…
  - Connected data in link-rich domain (i.e., friends, colleagues, employees, customers, etc.)
  - Routing or dispatch applications with location data (i.e., maps, directions, distances)
  - Recommendation engines (i.e., for products, dating services, etc.)
- Don't use graph databases for…
  - Applications where many or all data entities need to be updated at once or frequently
  - Data that needs lots of partitioning

Popular Graph Databases

Supports graph, key-value, and document-based access patterns

https://en.wikipedia.org/wiki/Graph_database#List_of_graph_databases

Aggregate Data Models

Aggregate Data Models

- *Aggregate*: a collection of related objects treated as a unit
  - Particularly for data manipulation and consistency management
- *Aggregate-oriented database:* a database comprised of aggregate data structures
  - Supports atomic manipulation of a single aggregate at a time
  - Makes it simple to scale out across clusters
    - Aggregates make natural units for replication and fragmentation/sharding
  - Aggregates match up nicely with in-memory data structures
  - Use a key or ID to look up an aggregate record
- An *aggregate-ignorant* data model has no concept of how its components can aggregate together (e.g., relational and graph DBs)
  - Can efficiently support query data in multiple ways, but makes it more challenging to scale across clusters

Aggregate Database Example: An Initial Relational Model

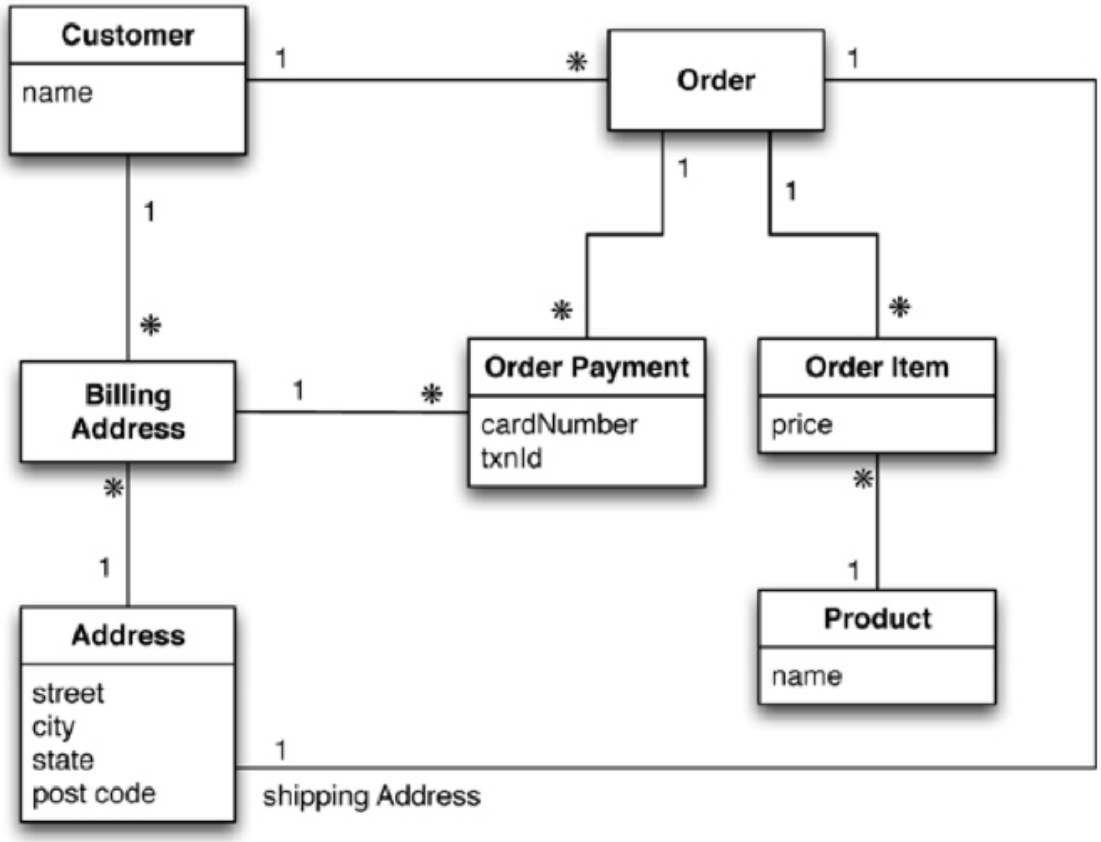


Figure 2.1. Data model oriented around a relational database (using UML notation [Fowler UML])

Aggregate Database Example: An Aggregate Data Model

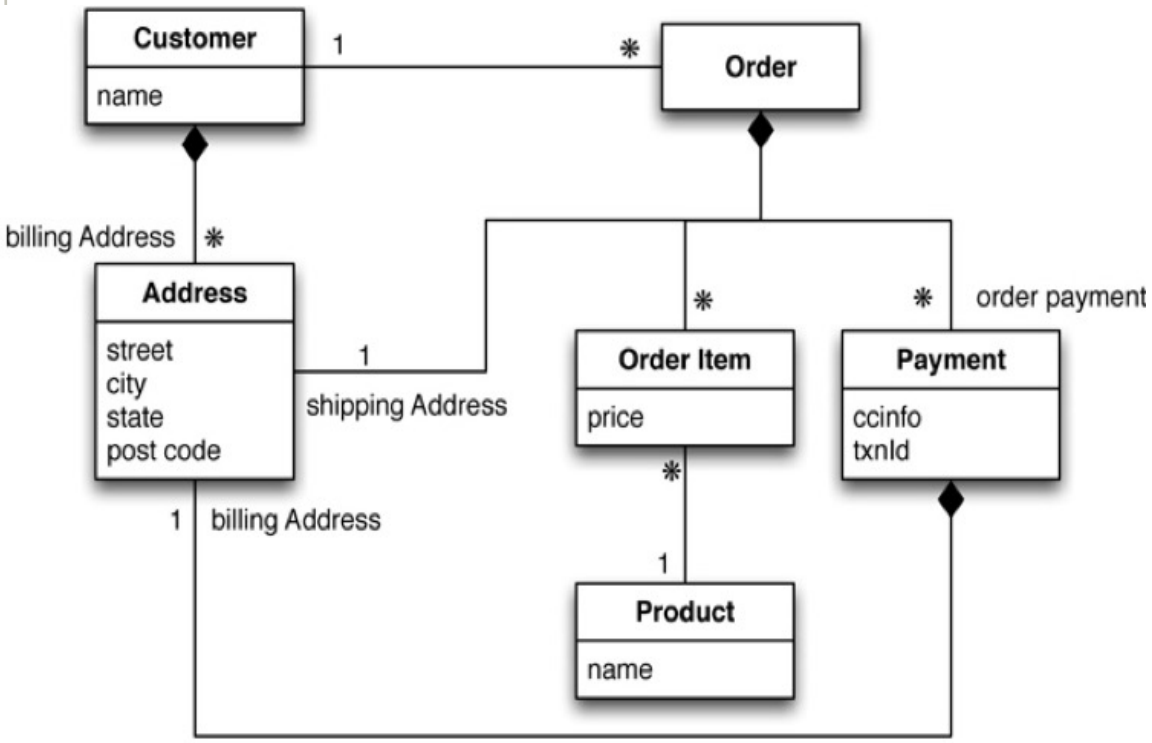


Figure 2.3. An aggregate data model

```
// 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 Database Example: Another Aggregate Model

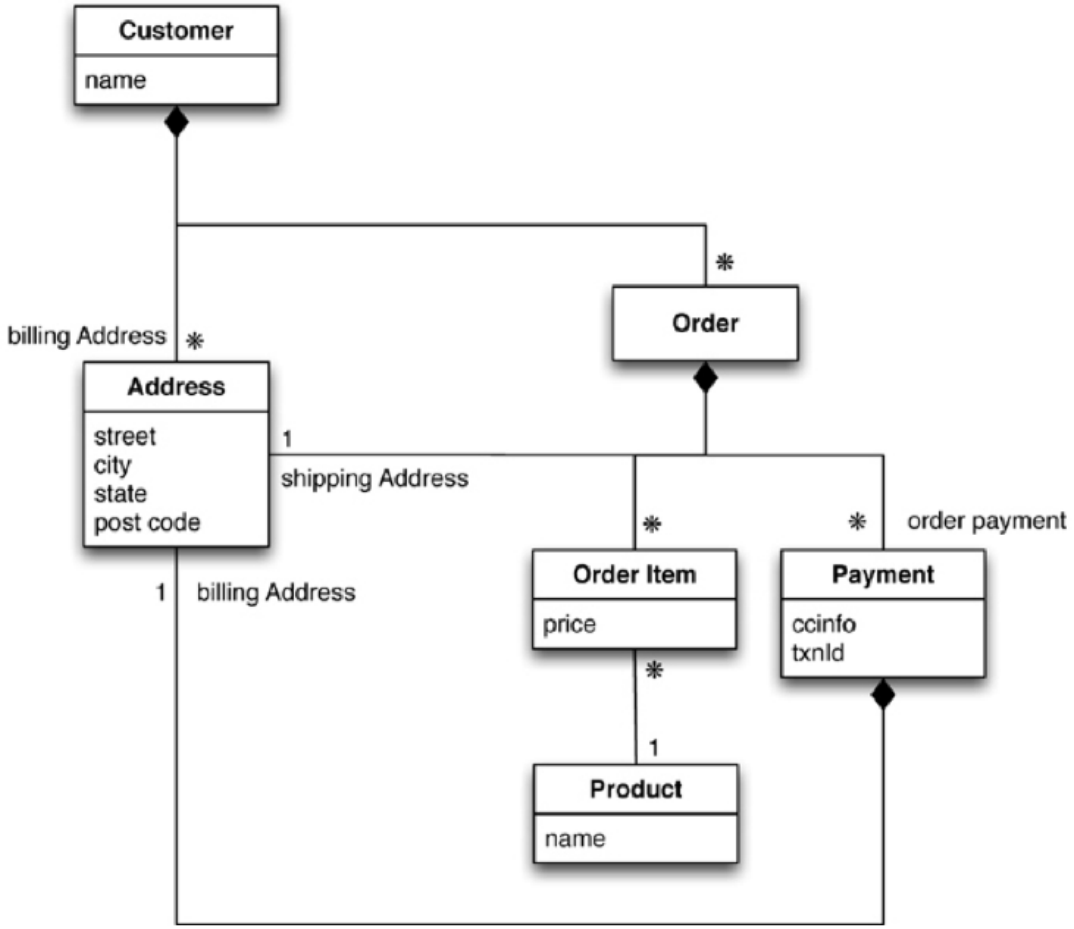


Figure 2.4. Embed all the objects for customer and the customer's orders

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"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

- Key-value databases
  - Stores data that is opaque to the database
    - The database cannot see the structure of records, just has a key to access a record
    - Application needs to deal with this
  - Allows flexibility regarding what is stored (i.e., text or binary data)
- Document databases
  - Stores data whose structure is visible to the database
    - Imposes limitations on what can be stored
    - Allows more flexible access to data (i.e., partial records) via querying
- Both key-value and document databases consist of aggregate records accessed by ID values

Aggregate-Oriented Databases

- Wide-column databases
  - Two levels of access to aggregates (and hence, two pars to the "key" to access an aggregate's data)
    - ID: to look up aggregate record
    - Column name: either a label for a value (name) or a key to a list entry (order id)
  - Columns are grouped into column families

Relationships in Aggregate Databases

- Aggregates contain ID attributes to related aggregates
  - Require multiple database accesses to traverse relationships
    - One to lookup ID(s) of related aggregate(s) in main aggregate
    - One to retrieve each of the related aggregates
  - Many NoSQL databases provide mechanisms to make relationships visible to the database (to make link-walking easier)
- Atomicity is limited to each aggregate, so updates to relationships **require the application to maintain consistency** (which is difficult!)
- Aggregate databases become awkward when it is necessary to navigate around many aggregates

Comparison of Data Management Capabilities

- Key-value databases
  - Opaque data store
  - Almost no capabilities requiring value introspection
- Document databases
  - Transparent data store
  - Some advanced capabilities (e.g., partial record queries, indexes)
- Wide-column databases
  - Transparent data store and dynamic schema
  - Some advanced capabilities (e.g., key spaces, query languages)
- Relational databases
  - Static uniform schema
  - Many advanced capabilities (e.g., integrity constraints, indexes, etc.)

Schema-less Databases

- Common to many NoSQL databases – also called *emergent schemas*
- Advantages
  - No need to predefine data structure
  - Good support for *non-uniform data*
- Disadvantages
  - Potentially inconsistent names and data types for a single value
    - Example: "quantity, Quantity, QUANTITY, qty, count, …" or "5, 5.0, five, V …"
    - The database does not enforce these things because it has no knowledge of the *implicit schema*
  - Management of the implicit schema **migrates into the application layer**
    - Failure to do this properly can lead to hard-to-catch bugs (e.g., a bug affecting all customers who registered from Jan 1st 2018 until July 4th 2019)
    - Need to look at code to understand what data and structure is present
      - No standard location or method for implementing the logic to do this
    - What do you do if multiple applications need access to the database?

Key-Value Databases

Key-Value Databases

- Key-value store is a simple hash table
  - Records accessed via *key*
    - Akin to a primary key for relational database records
    - Quickest (or only) way to access a record
  - *Values* can be of any type
    - Like blob data type in relational database
- Operations:
  - Get a value for a given key
  - Set (or overwrite or append) a value for a given key
  - Delete a key and its associated value

Key-Value Database Features

- No ACID transactions
  - Though each operation is atomic (e.g., set), a series of operations are unrelated (like autocommit)
  - Weak consistency and durability
    - Default configuration is aimed at high performance
    - Some options for higher durability (e.g., more frequent writes to disk, synchronous writes)
- Scale by both fragmentation and replication
  - Shard by key values (using a uniform function)
  - Replicas should be available in case a shard fails
    - Otherwise, all reads and writes to the unavailable shard fail

Interacting with Key-Value Databases

- Applications can only query by key, not by values in the data
- Design of key is important
  - Must be unique across the entire database
  - Bucket can provide an implicit top-level namespace (e.g., university_, loans_, etc.)
- Expiration times can be assigned to key-value pairs
  - Good for storing transient data

Interacting with Key-Value Databases

- How and what data gets stored is managed entirely by the application
- Single key for related data structures
  - Key incorporates identification data (i.e. user_<sessionID>)
  - Data can include various nested data structures (i.e. user data including session, profile, cart info)
  - All data is set and retrieved at once
- Multiple keys for related data structures
  - Key incorporates name of object being stored (i.e. user_<sessionID>_profile
  - Multiple targeted fetches needed to retrieve related data
  - Decreases chance of key conflicts (aggregates have their own specific namespaces)

Key-Value Aggregate Examples

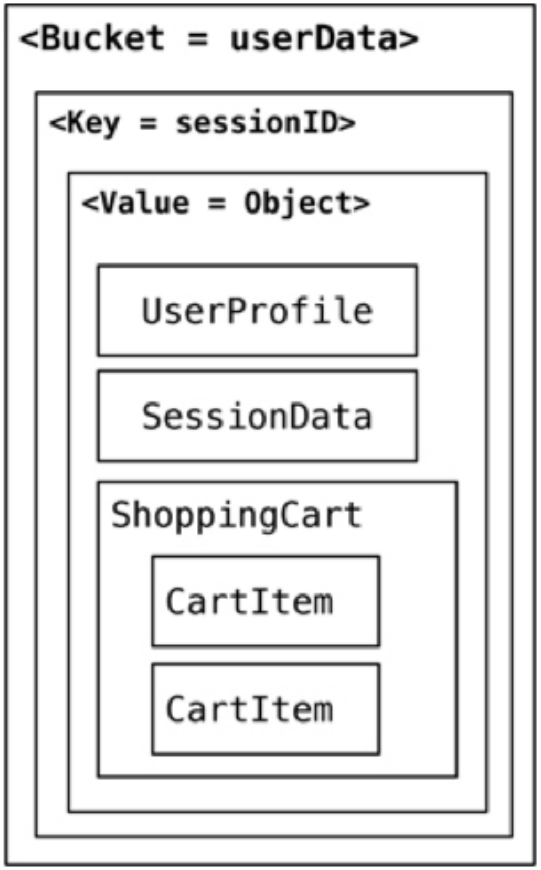


Figure 8.1. Storing all the data in a single bucket

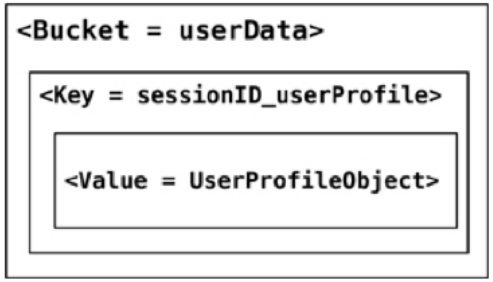


Figure 8.2. Change the key design to segment the data in a single bucket.

Using Key-Value Databases

- Use key-value databases for:
  - Caching
  - Transient data
  - Data accessed via a unique key (i.e., page id, session, user profile, shopping cart, etc.)
- Don't use key-value databases for:
  - Relationships among data
  - Multi-operation transactions
  - Operations on sets of records
  - Querying by data (value instead of key)

Popular Key-Value Databases

Memcached

Practice Quiz: Aggregate Data Models

- Working with a neighbor, draw one or more aggregates to represent the following information:
  - Alice has borrowed a book, "Through the Looking-Glass," by Lewis Carroll (ISBN: 1949460894, copy number 2, accession number 4837). The book is due back on December 1st, 2021.

Document Databases

Document Databases

- Store of documents with keys to access them
  - Similar to key-value databases except…
  - Can see and dynamically manipulate the structure of the documents
    - Often structured as JSON (textual) data
    - Each document can have its own structure (non-uniform)
  - Each document is (automatically) assigned an ID value (_id)
- Consistency and transactions apply to single documents
  - If this isn't sufficient for your application, then document databases are a poor fit
- Replication and sharding are by document
- Queries to documents can be formatted as JSON
  - Able to return partial documents

Document Database Example

```
// in order collection
[{
  "customerId":12345,
  "orderId":67890,
  "orderDate:"2012-12-06",
  "items":[{
    "product":{
      "id":112233,
      "name":"Refactoring",
      "price":"15.99"
    },
    "discount":"10%"
   },
   {
    "product":{
      "id":223344,
      "name":"NoSQL Distilled",
      "price":"24.99"
    },
    "discount":"3.00",
    "promo-code":"cybermonday"
  },
  ],
  ""
}
```

SQL	Document Database Query
select * from order	db.order.find()
select * from order where customerId = 12345	db.order.find({ "customerId":12345 })
select orderId, orderDate from order where customerId = 12345	db.order.find({"customerId":12345}, {"orderId":1,"orderDate":1})
select * from order natural join orderItem natural join product p where p.name like '%Refactoring%'	db.order.find({ "items.product.name": "/Refactoring/" })

Using Document Databases

- Document databases *can* be used for…
  - Content management or blogging platforms
  - Web analytics stores
  - E-commerce applications
  - Event logging: central store for different kinds of events with various attributes
- …*but* relational databases can be used for many of the same things
- Do not use document databases for…
  - Transactions across multiple documents (records)
  - Ad hoc cross-document queries

Popular Document Databases

Supports graph, key-value, and document-based access patterns

Wide-Column Store Databases

Wide-Column Store Databases

- Structure of data records:
  - Records are indexed by key
  - Columns are grouped into column families (like RDBMS tables)
  - Efficient support for sparse data
- Data access:
  - Get, set, delete operations
  - Query language (e.g., CQL: Cassandra Query Language)
- Also known as *column-based* or *column family*
  - **Not** the same as *column-oriented*

Wide-Column Store Database Example

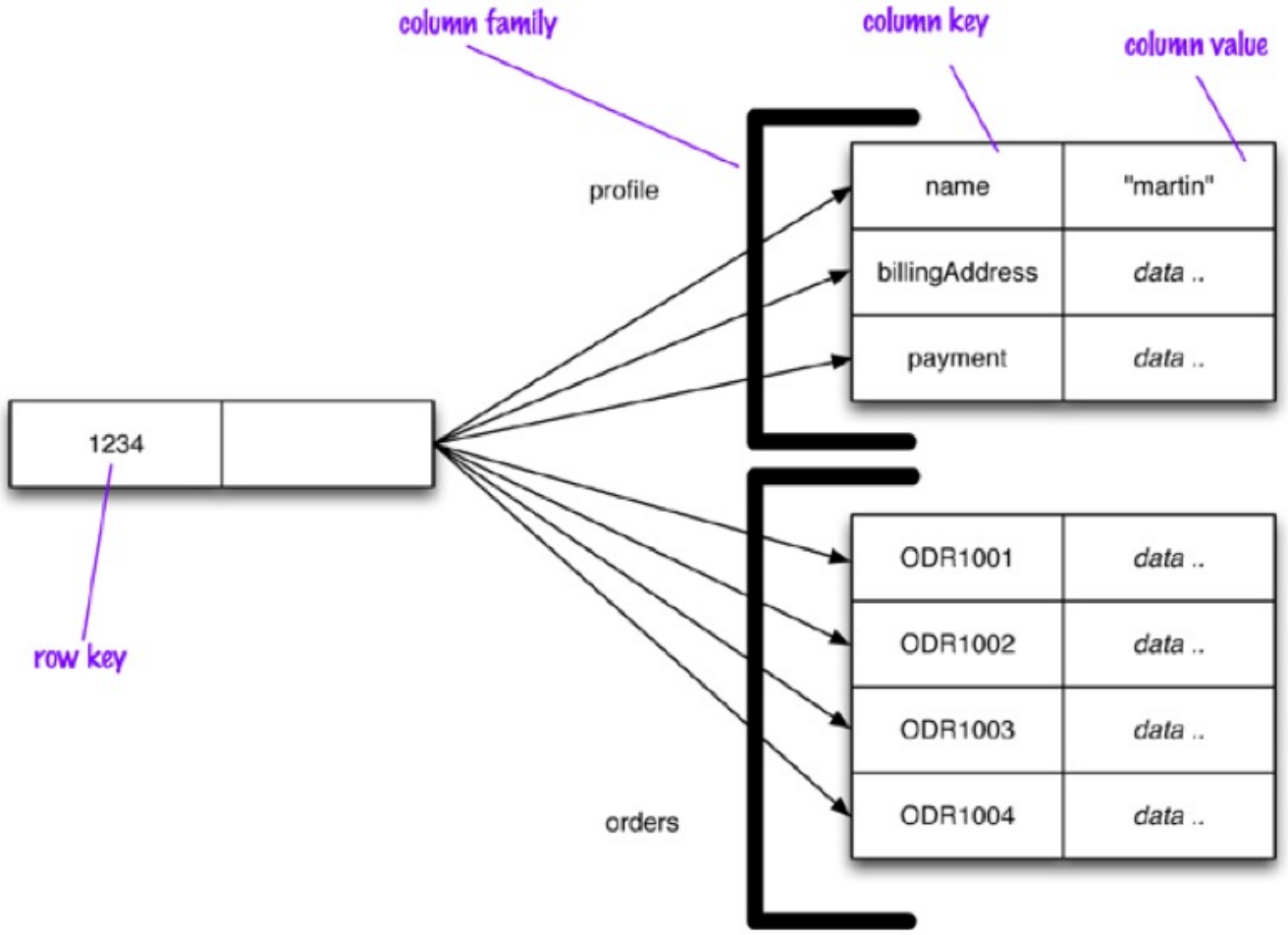


Figure 2.5. Representing customer information in a column-family structure

Wide-Column Store Database Example

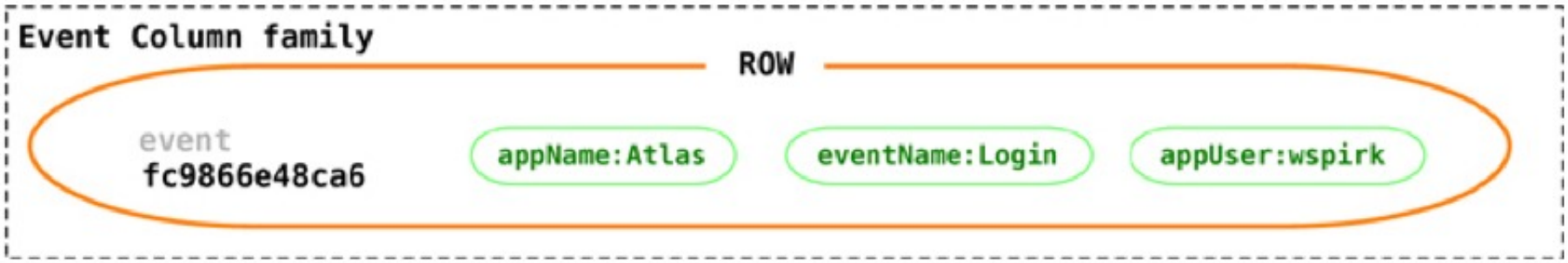


Figure 10.2. Event logging with Cassandra

```
CREATE COLUMNFAMILY Customer (
  KEY varchar PRIMARY KEY,
  name varchar,
  city varchar,
  web  varchar);


INSERT INTO Customer (KEY,name,city,web)
  VALUES ('mfowler',
            'Martin Fowler',
            'Boston',
            'www.martinfowler.com');


SELECT * FROM Customer;


SELECT name,web FROM Customer WHERE city='Boston'
```

Using Wide-Column Store Databases

- Useful for:
  - Big data (e.g., web crawling)
  - Sparse data
- Not useful:
  - If only a few database servers are needed
  - For systems requiring ACID transactions
  - For flexible access patterns (e.g., joins between tables)

Popular Wide-Column Store Databases

Open Source

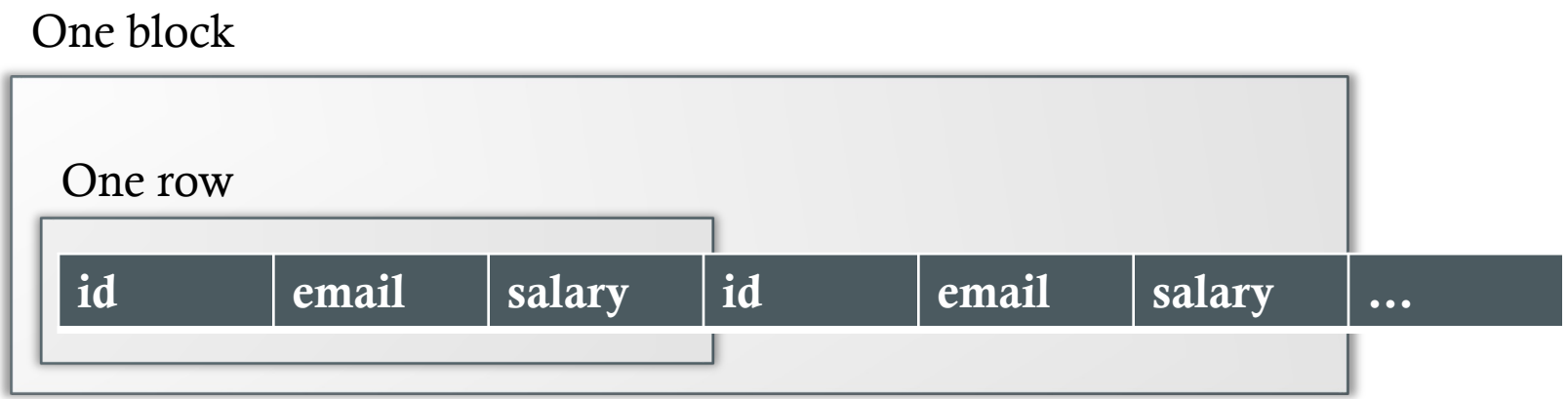
Cloud Services

Google Cloud Bigtable

Amazon DynamoDB

Column-Oriented Databases

Storage of a Row-Oriented Database



Consider: `SELECT salary FROM info WHERE email = a@gmail.com`

Consider a query including: `sum(salary)`

Storage of a Column-Oriented Database

One block

One key-value pair

id	email	id	email	id	email	…

One block

One key-value pair

id	salary	id	salary	id	salary	…

Consider: `SELECT salary FROM info WHERE email = a@gmail.com`

Consider a query including: `sum(salary)`

Using Column-Oriented Databases

- Useful for:
  - Time series data
  - OnLine Analytical Processing: OLAP (e.g., event logging and analysis)
- Not useful for:
  - OnLine Transaction Processing: OLTP (e.g., financial transactions)

Popular Column-Oriented Databases

Amazon Redshift

NoSQL Design Activity

NoSQL Design Activity

- Working with your project team:
- Identify a NoSQL DB that might serve a purpose for your application. Describe how you might use it, and the pros and cons of using it in combination with or instead of a relational database:
  - Key-value store (e.g., for caching or transient data)
  - Document database (think about possible aggregates)
  - Wide-column store (e.g., for big data or sparse data)
  - Column-oriented database (e.g., for analytics)
  - Graph database (e.g., for recommendations)

Relational vs NoSQL Migrations

Schema Migrations

- The structure of data changes regardless of what kind of database it resides in
- System requirements evolve and the supporting database(s) must keep pace
- *Transition phase*: period of time in which the old and new schema versions must be maintained in parallel
- For example: suppose our application originally only allowed customers to store a shipping address and a billing address
  - Now, customers can have multiple shipping and billing addresses
  - Converting two one-to-one relationships to one-to-many

Schema Migration Challenges

- Minimize transition phase
  - How can all data be migrated as quickly as possible?
  - Does all data need to be migrated?
- Avoid downtime of production databases
  - Challenging to avoid for large systems, as DDL to alter structure often requires locking entire tables
- Ensure database remains usable to all applications during transition phase
  - Different applications will integrate the schema changes at different times
  - Don't cause errors
  - Don't corrupt or lose data

Schema Changes in Relational Databases

- Must keep database and applications in sync
  - Schema changes applied separately to database and applications
- Schema changes need to be applied in the correct order
- Need to ensure that schema changes can be rolled back if there is a problem
- Schema changes need to be applied to all environments in the same fashion
  - Development, test, staging, production

Database Migration Frameworks

- Logic to execute each schema change is stored in a file which contains a version string
  - Scripts to generate initial database or take a “snapshot” of the current structure of an existing database get the initial version (if the database already exists)
- May contain logic to upgrade and downgrade the database to/from its version
- Migration framework is responsible for applying changes up/down to a certain version of the database in the right order
- Can be integrated into the project build process so it automatically gets executed in various environments when a new version of the application is introduced there

MiniFacebook Migrations

1. Generate migrations
   - python manage.py makemigrations
2. Apply migrations
   - python manage.py migrate

- Example:
  - `django/djangoproject/minifacebook/migrations/0001_initial.py`
  - `django/djangoproject/minifacebook/migrations/0002_alter_status_options.py`
  - `django/djangoproject/minifacebook/migrations/0003_poke.py`

Migration Python Code

0001_initial.py

```
from django.db import migrations, models
import django.db.models.deletion
import uuid


class Migration(migrations.Migration):
    initial = True
    dependencies = []
    operations = [
        migrations.CreateModel(
            name='Profile',
            fields=[
                ('id', models.UUIDField(default=uuid.uuid4, editable=False,
primary_key=True, serialize=False)),
                ('first_name', models.CharField(max_length=100)),
                ('last_name', models.CharField(max_length=100)),
                ('email', models.EmailField(max_length=254)),
                ('activities', models.TextField()),
            ],
        ),
...
```

Migration SQL Code

```
> python manage.py sqlmigrate minifacebook 0001
BEGIN;
--
-- Create model Profile
--
CREATE TABLE "minifacebook_profile" ("id" uuid NOT NULL PRIMARY KEY, "first_name"
varchar(100) NOT NULL, "last_name" varchar(100) NOT NULL, "email" varchar(254) NOT NULL,
"activities" text NOT NULL);
--
-- Create model Status
--
CREATE TABLE "minifacebook_status" ("id" uuid NOT NULL PRIMARY KEY, "message" text NOT
NULL, "date_time" timestamp with time zone NOT NULL, "profile_id" uuid NOT NULL);
ALTER TABLE "minifacebook_status" ADD CONSTRAINT
"minifacebook_status_profile_id_dfb04e9b_fk_minifaceb" FOREIGN KEY ("profile_id")
REFERENCES "minifacebook_profile" ("id") DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "minifacebook_status_profile_id_dfb04e9b" ON "minifacebook_status"
("profile_id");
COMMIT;
```

Schema Changes in a NoSQL Store

- Implicit schema: the database may be "schema-less," but the application still must manage the way data is structured
- Incremental migration: read from both schemas and gradually write changes
  - Read methodology:
    - Read the data from the new / updated field(s)
    - If the data is not in the new field(s), read it from the old ones
  - Write methodology:
    - Write data only to the new field(s)
    - Old field may be removed
  - Some data may never be migrated
- Changes to top-level aggregate structures are more difficult
  - Example: make nested order records (inside customers) into top-level aggregates
  - Application must work with both old and new structures

Incremental Migration Example

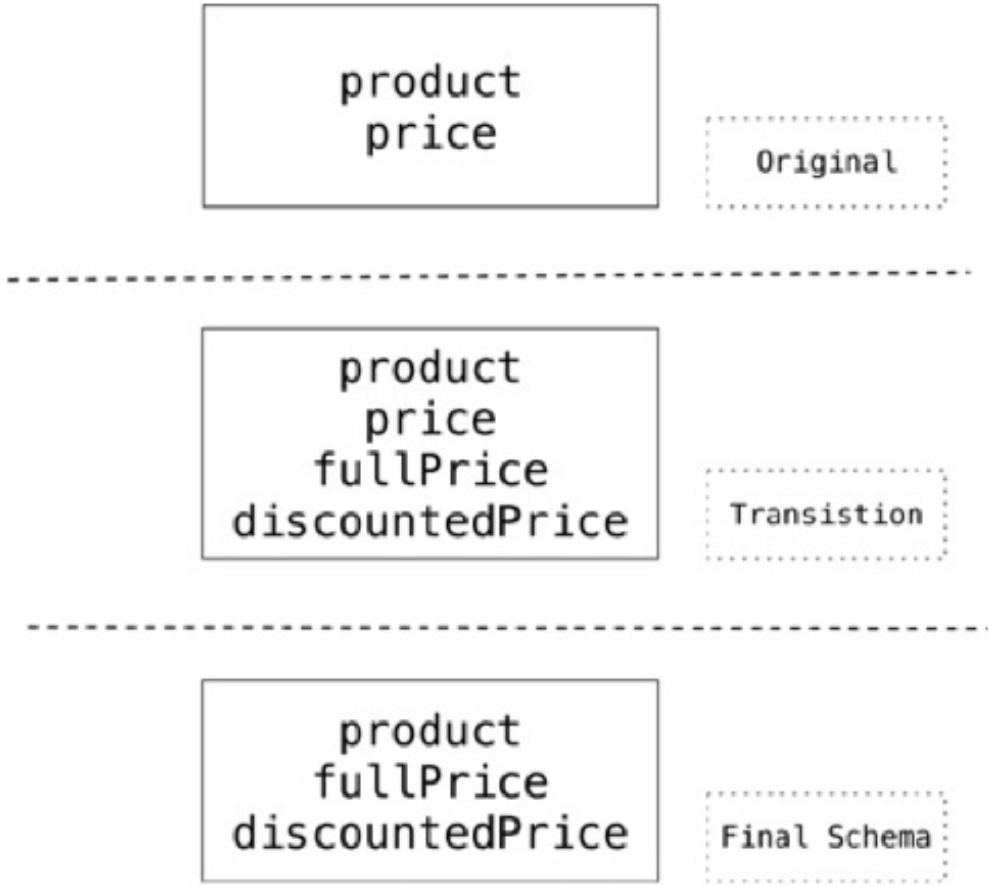


Figure 12.6. Transition period of schema changes

Relational vs NoSQL Migrations

- If you are willing to accept a small amount of downtime (to restart your web application), automatic migrations from a framework like Django are a clear win
- If downtime is unacceptable, then regardless of your DBMS your application code will need to support data in **both the old and new schemas**
  - With a relational database, after the migration has finished you can remove the code supporting the old data format
  - With incremental migrations in NoSQL, you may need to support data in both schemas indefinitely
    - Could get messy over the years…

Practice Quiz: Project Check In

- Working with your project team:
  - Review the Homework 12 and 13 Project Milestones
  - List the items you still need to complete, and make a plan to divide the work

Today you will learn…

- About the tradeoffs of enforcing ACID properties
- How to choose between relational DBs and NoSQL

Related Issues

Review: Distribution Models

- Single server: simplest model, everything on one node
- Sharding: storing different data on different nodes (AKA, fragmentation)
  - *Auto-sharding:* some databases handle the logistics of sharding so that the application does not have to
- Replication: duplicate data on multiple nodes
  - *Primary-standby replication:* primary node is responsible for updates, standby node(s) support reads (AKA, master-follower)
  - *Peer-to-peer replication:*
    - All nodes do reads and writes, and communicate changes to other nodes (AKA, multi-master)
    - Pros: Eliminates any one master as a single point of failure
    - Cons: maintaining consistency can be challenging (e.g., write-write conflicts, when two users update the same data item on separate nodes)

Review: ACID

- **A**tomicity: either all of the transaction completes, or none of it completes
  - If any part of the transaction fails, all effects of it must be removed from the database
- **C**onsistency: database ends the transaction in a consistent state (provided it started that way)
- **I**solation: concurrently executing transactions must be unaware of each other (as if they ran serially)
  - It should look to one as if the other has not started or has already completed
- **D**urability: a transaction's effects must persist in the database after it completes

Two Forms of Consistency

- Update consistency: ensuring serial database changes
- Read consistency: ensuring users read the same value for data at a given time
- Note: These are related to ACID's "consistency," but slightly different

Update Consistency

- Ensuring serial database changes
- *Pessimistic* approach: prevents conflicts from occurring (i.e. locking)
- *Optimistic* approach: detects conflicts and sorts them out (i.e. validation)
  - Conditional update: just before update, check to see if the value has changed since last read
  - Write-write conflict resolution: automatically or manually merge the updates
- Trade-off between safety and "liveness" (responsiveness)

Read Consistency

- Ensuring users read the same value at a given time
- *Logical consistency:* consistent between reads on a single node
- *Replication consistency:* consistency between replicas
  - Eventual consistency, not immediate
  - Challenge: if a user makes a change, they expect to see the change (AKA, *read-your-writes consistency*)
  - Potential solution: *S*ession affinity: assign a user's session to a given database node (AKA, *sticky sessions*)

CAP Theorem

- Pick two of these three:
  - **Consistency:** ensure update consistency (serial database changes), and read consistency (data becomes visible to all readers simultaneously)
  - **Availability:** if you can talk to a node, you can read and write
  - **Partition tolerance:** cluster can continue operating even if a network fault divides it into multiple isolated partitions
- For distributed systems it's "pick two," because *partitions can happen to any distributed system*
  - If you have just one server, it cannot be partitioned, so you can get both consistency and availability

CAP Explained

- To continue operating despite a network partition **(partition tolerance)**, we need to trade off consistency of data vs. availability
  - If we want to support writes (**availability**) while the partitions cannot communicate, their data will become inconsistent
  - If we want to keep the data in both partitions **consistent**, we cannot continue servicing requests

Diluting ACID: Relaxed Consistency

- If a network partition occurs, suppose we choose **availability over consistency**. For example:
  - A user's shopping cart becomes unavailable, so a new cart is created. At checkout, user is prompted to merge carts.
  - Allow two users to book a flight/hotel/etc. at the same time. It might be okay if there is an overbooking if extra seats/rooms/etc. can be made available.
- **Counterpoint**: supporting relaxed consistency requires complex application logic. Fast recovery from partitions (less than 10 seconds) can ensure consistency, while having high availability.
  - Further reading: The Limits of the CAP Theorem

Diluting ACID: Relaxed Durability

- Replication durability: what happens if primary node receives an update, but it cannot communicate with a standby node?
  - Not necessary to communicate with **all** standbys to preserve strong durability; just a large enough quorum
- But what if you are willing to sacrifice durability?
  - You can achieve higher performance by writing to disk less frequently
  - Perhaps a good choice for:
    - Telemetry (e.g., performance statistics): you wouldn't want your webpages to hang if the telemetry server went down anyway
    - Session data: updated very frequently, and a critical determiner of site responsiveness. Only a little annoying to customers if they lose a few minutes of recent browsing history, etc.

Polyglot Persistence

Polyglot Persistence

- Pick the best tool for the job: different databases are designed for different types of data
- Example:
  - Many e-commerce sites run entirely on a relational database
  - Alternatively:
    - Keep order processing data in the RDBMS
    - Session and shopping cart data could be separated into a key-value store
      - More transient data which can be copied to RDBMS once an order is placed
    - Customer social data could reside in a graph database
      - Designed specifically to optimize traversing relationships between data, perhaps for recommendations

Polyglot Persistence Example

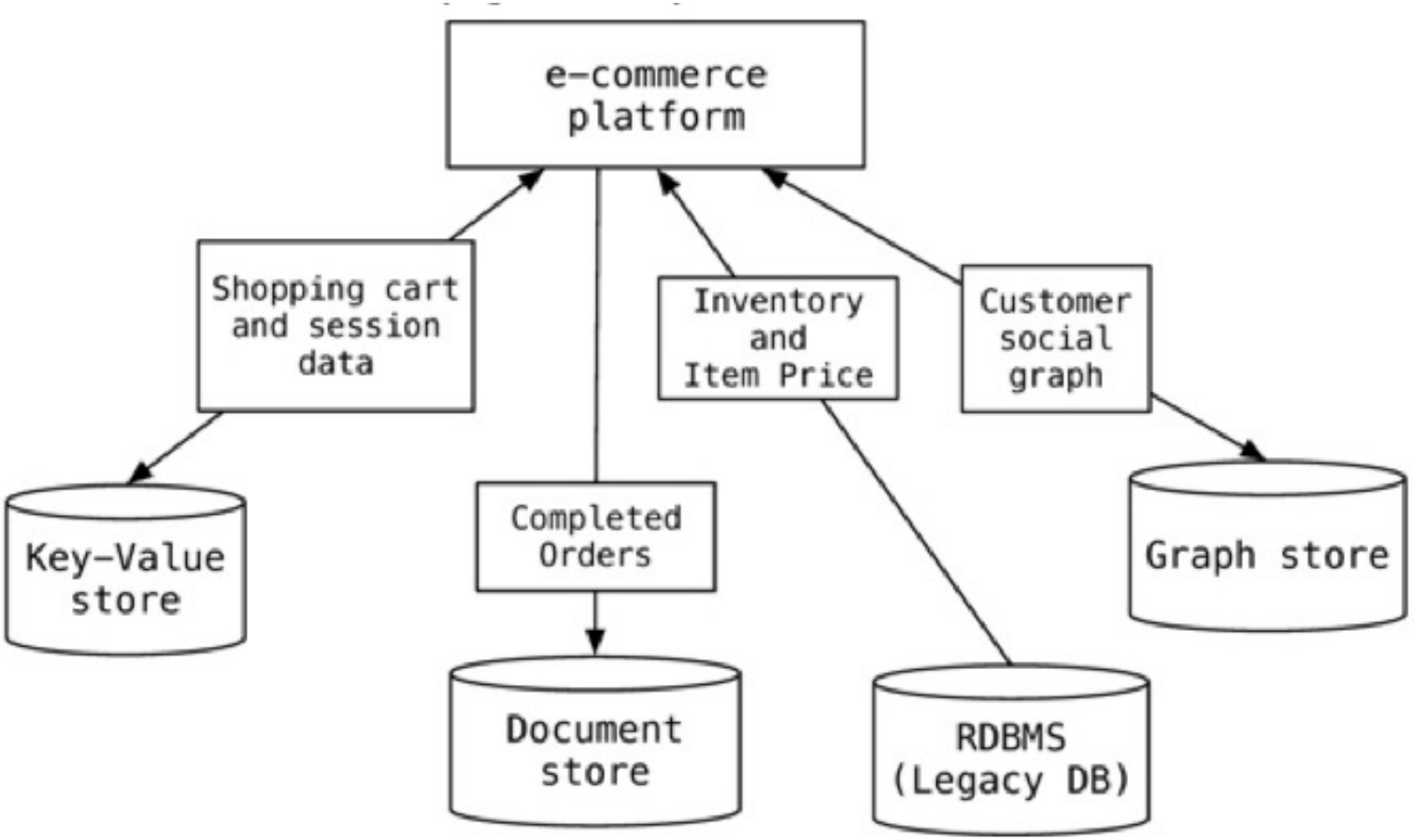


Figure 13.3. Example implementation of polyglot persistence

Web Service Wrappers for Data Stores

- Advantages over direct access to data store:
  - Easier and cleaner to integrate the data store with multiple applications
  - Allows database structure to change without needing to update applications that use it
    - Potentially even change the database itself
- Drawbacks:
  - Overhead of another layer
  - Sometimes modifying a web service still requires changing applications that use it (reduces this likelihood)

Web Service Wrapper Example

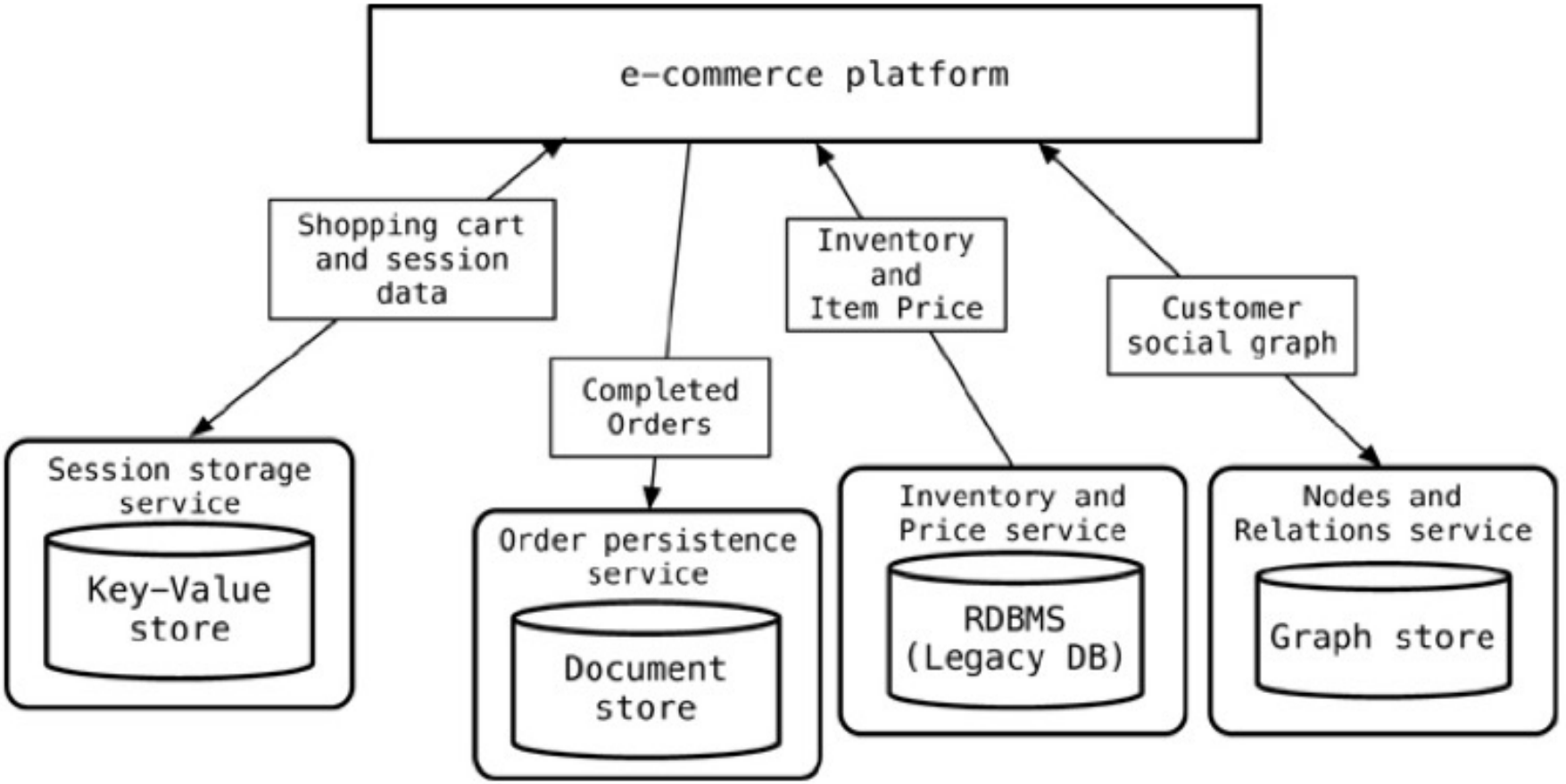


Figure 13.5. Using services instead of talking to databases

Which DBMS should you choose?

For a given purpose, do you use relational or NoSQL?

When You Should Use NoSQL

- When you need to perform complex graph queries, use a graph database
- When you have weak consistency or durability needs
  - For caching, use a key-value store
  - For telemetry, use:
    - InfluxDB for time series data (column oriented)
    - Elasticsearch for log data (supports fast text searches)

When You Might Use NoSQL

- Potentially: when your application only needs to support limited access patterns.
  - This might be the case if your data is mainly collected or displayed in terms of aggregates
- Debatable: if your data includes complex, nested, or hierarchical structures; or if your data is non-uniform
  - You could simply de-normalize or store JSON in a relational DB. Can even get better performance this way.
- Debatable: programmer productivity
  - Easier to prototype with an implicit schema, but much harder to maintain. And again, you could simply use JSON in a relational DB.
- Debatable: scalability
  - Distributed relational databases (e.g., CockroachDB, Google Spanner, Google F1), etc. show you can get scalability and consistency

When *Not* to Use NoSQL

- When you need ACID (e.g., transferring money between accounts)
- When many different applications with different developers/owners will access the data (i.e., integration databases)
  - Relational databases support strong security measures at the database level to protect data
  - An explicit schema serves as documentation
  - However, today web-services are considered a better practice
- If you don't have a good reason to use NoSQL
  - Relational databases are well-known, mature, and have lots of tools (e.g., the Django admin interface)
  - ORM can cut down on impedance mismatch
  - **Much** easier to switch from a well-designed relational schema to an implicit NoSQL schema – potentially **impossible** to do the reverse

My Recommendations

- Relational database for primary data storage
- Key-value stores for caching
- Graph databases for data exploration, recommendations, etc.
- Column-oriented databases for telemetry
- Wide-column store databases for big data
- Generally, avoid document databases
  - Much easier to switch from an explicit schema to an implicit schema than vice versa
  - Perhaps tempting to rapid-prototype with an implicit schema, but all-too-often, prototypes are deployed to production…

Further Reading

- https://www.cockroachlabs.com/docs/stable/cockroachdb-in-comparison.html
- https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/
- Flowchart (linked from schedule)