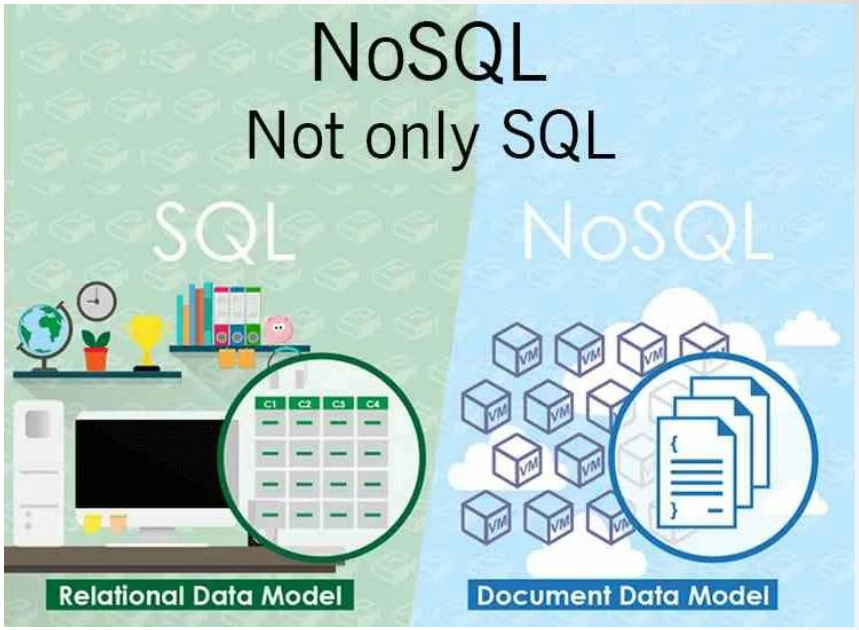


NoSQL for Java developers

June, 26th 2020

Sergiy Morenets, 2020

DEVELOPER 16 YEARS

TRAINER 7 YEARS

WRITER 4 BOOKS

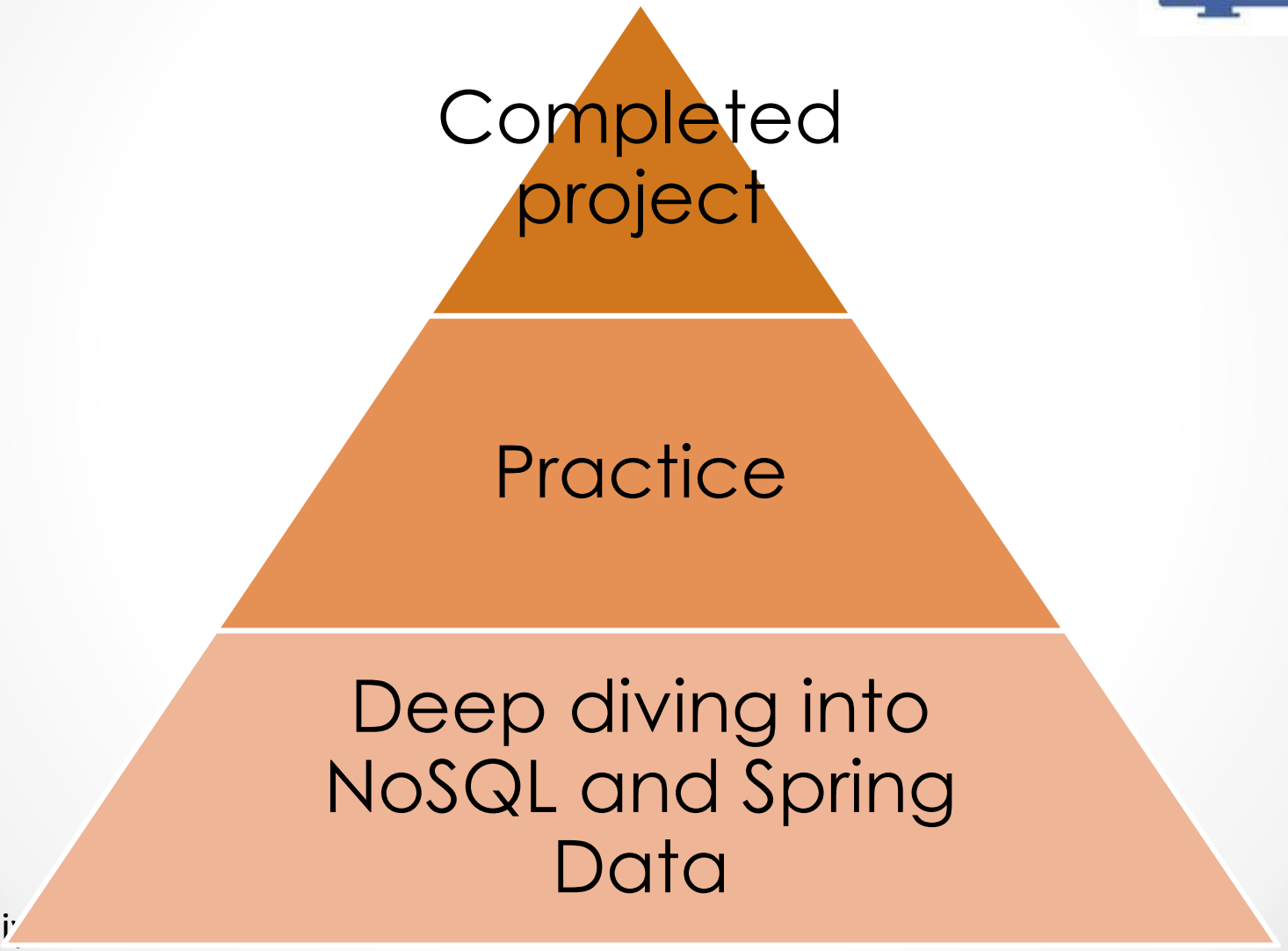
FOUNDER

ITSimulator

SPEAKER

Java
frameworks
day

Goals


Completed project
Practice
Deep diving into NoSQL and Spring Data

Agenda

- ✓ Evolution of database systems
- ✓ Relational databases vs NoSQL
- ✓ Overview of NoSQL databases
- ✓ MongoDB
- ✓ Redis
- ✓ Apache Cassandra
- ✓ Neo4j
- ✓ GUI tools
- ✓ Patterns & Anti-patterns
- ✓ Spring Data projects
- ✓ Integration tests

Database ranking. May 2020

357 systems in ranking, May 2020

Rank			DBMS	Database Model	Score		
May 2020	Apr 2020	May 2019			May 2020	Apr 2020	May 2019
1.	1.	1.	Oracle	Relational, Multi-model	1345.44	+0.02	+59.89
2.	2.	2.	MySQL	Relational, Multi-model	1282.64	+14.29	+63.67
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1078.30	-5.12	+6.12
4.	4.	4.	PostgreSQL	Relational, Multi-model	514.80	+4.95	+35.91
5.	5.	5.	MongoDB	Document, Multi-model	438.99	+0.57	+30.92
6.	6.	6.	IBM Db2	Relational, Multi-model	162.64	-2.99	-11.80
7.	7.	7.	Elasticsearch	Search engine, Multi-model	149.13	+0.22	+0.51
8.	8.	8.	Redis	Key-value, Multi-model	143.48	-1.33	-4.93
9.	9.	↑ 11.	SQLite	Relational	123.03	+0.84	+0.14
10.	10.	↓ 9.	Microsoft Access	Relational	119.90	-2.02	-23.88

Sergiy Morenets, 2020

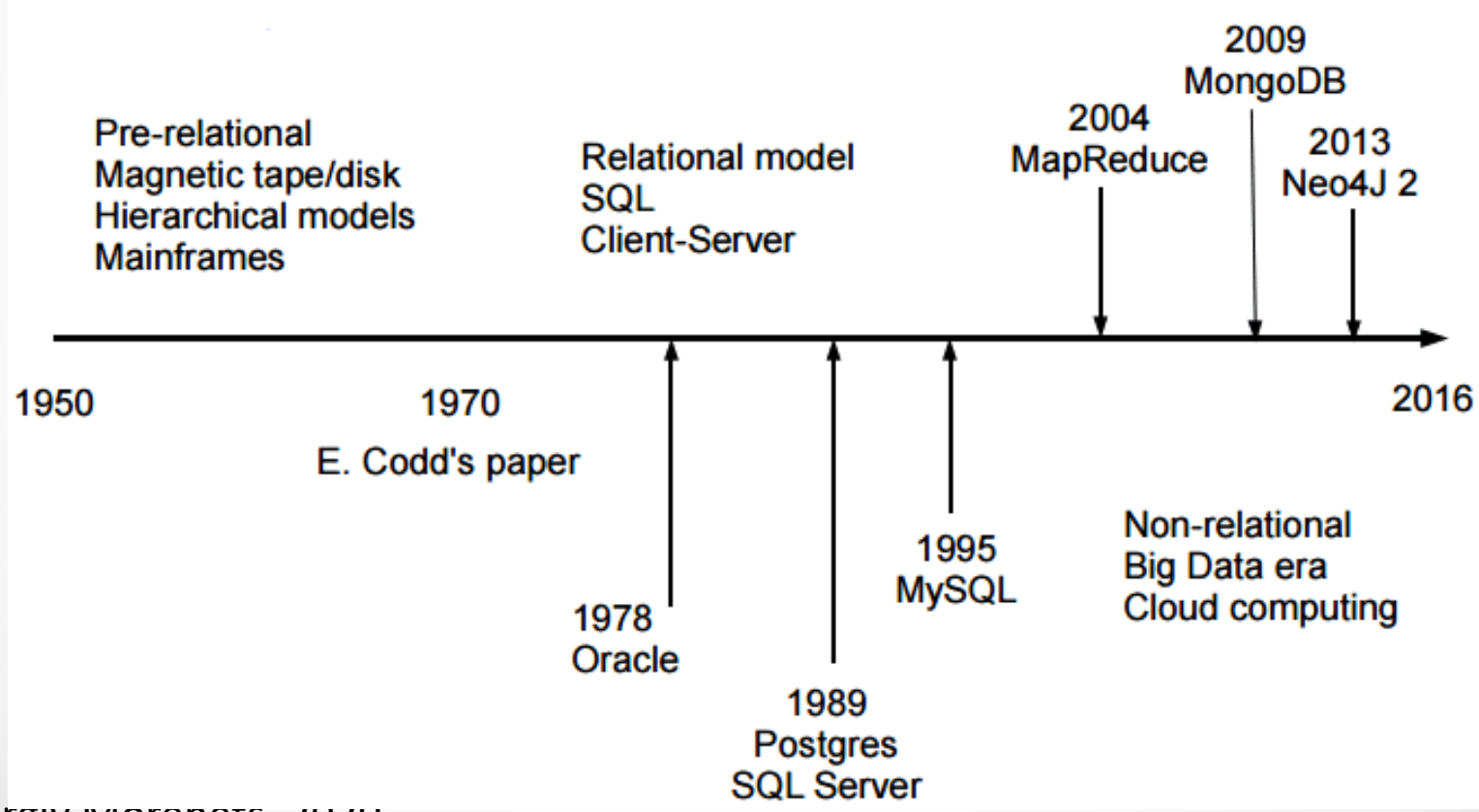
Stackoverflow. 2020 survey

Migration to NoSQL

Transactions

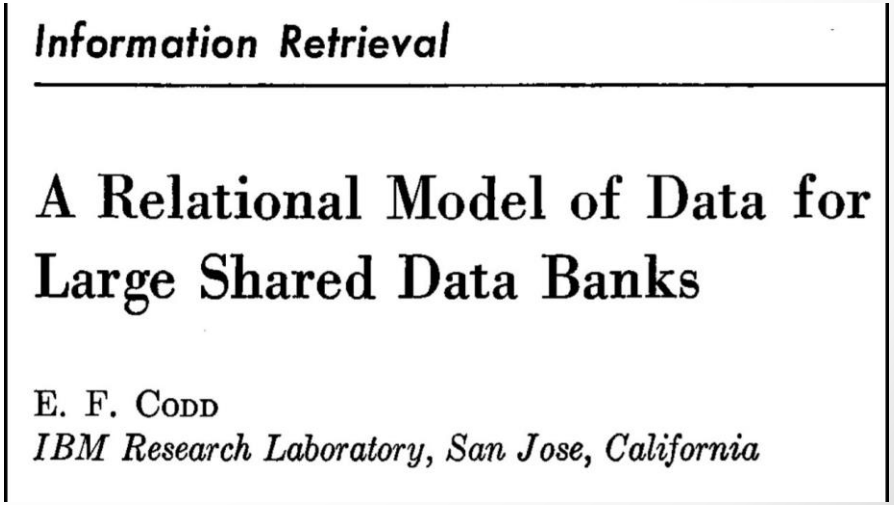
Copy-paste

Database evolution



Relational databases history

- Relational database model firstly appeared in 1970 in Edgar Codd's work
- Tuple is unordered set of attributes (**row**)
- Relation is collection of tuples (**table**)
- **Constraints** enforce consistency of the database
- **Operations** on relations (joins, projections, unions) always return relations.



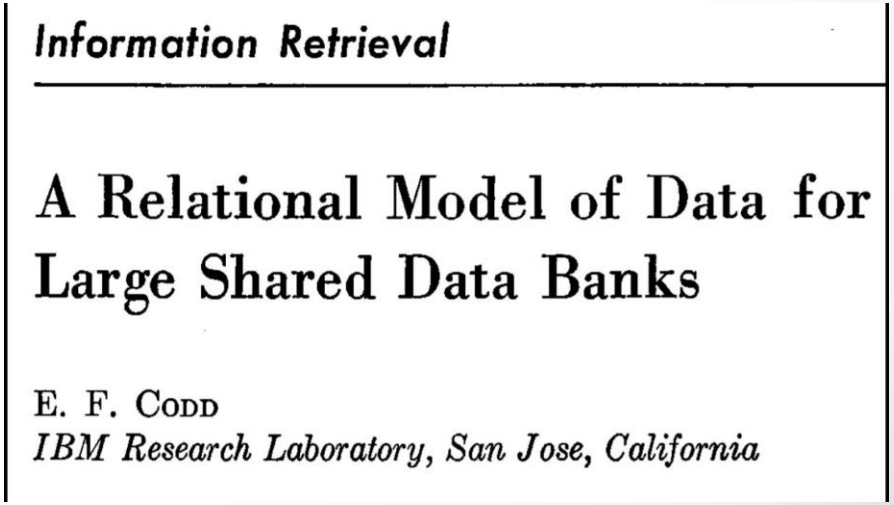
Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Relational databases history

- Each row has unique key (identifier)
- Query language for accessing data
- Arrays and nested structures are not supported
- Normal forms describe “structuration” of the data
- Jim Gray in late 70’s describe transactions as “transformation of state which has the properties of atomicity, durability and consistency”



Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Relational databases history

- Each relational database should support relational data model, ACID transactions and SQL for data access
- First relational DB System R was developed by IBM in 1974 with basic SQL support
- In 1977 Larry Ellison founded Oracle Corporation and released Oracle DB in 1979
- Object-oriented databases were introduced in 90’s but quickly declined in favor of ORM

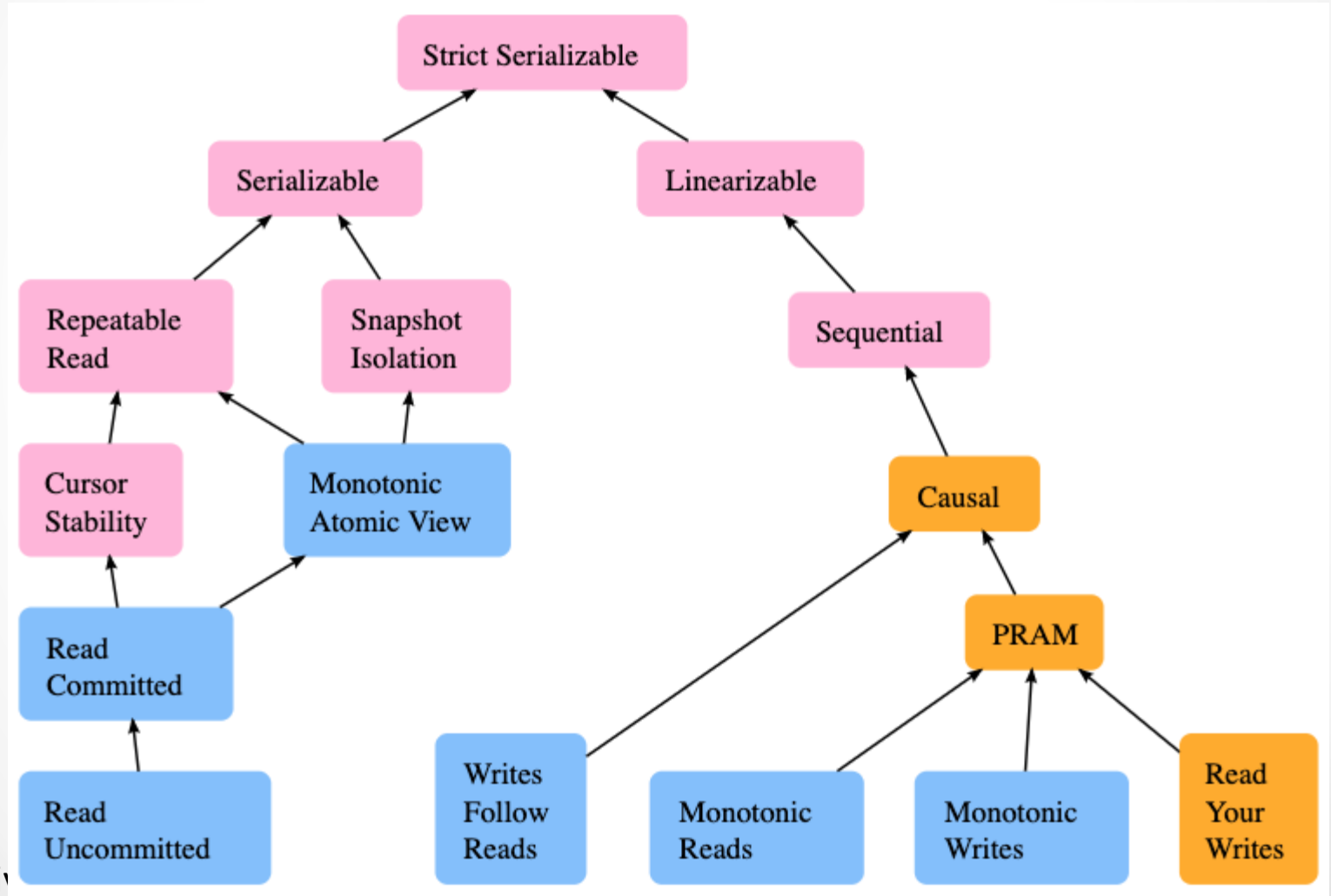
Relational databases

- ✓ Fixed set of columns
- ✓ Atomic fields
- ✓ Normalization
- ✓ Hard to change the schema
- ✓ Transactions (strong consistency)
- ✓ SQL for queries

Relational DB. Issues

- ✓ Auto-incrementing/global lock is hard to achieve for distributed systems
- ✓ Nested transactions
- ✓ Impedance mismatch
- ✓ Large volume of data

Transactions. Isolation levels



Transactions

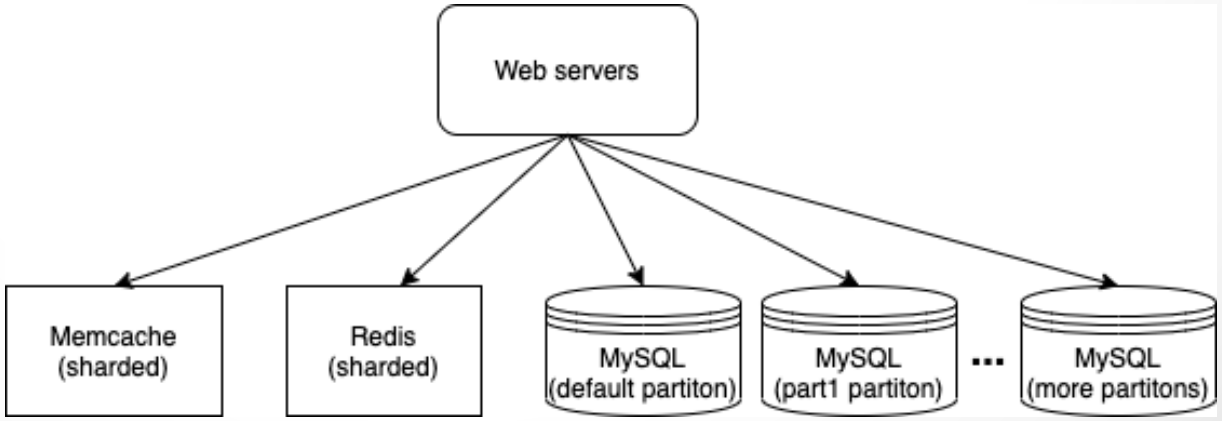
For most people, coming into a transactionless environment is quite a shock. Using transactions is a very accepted part of working with databases. A lot of people, including myself, see transactions as one of the key benefits that databases give you.

The rationale for not using transactions was that they harm performance at the sort of scale that eBay deals with.

Sergiy Morenets, 2020

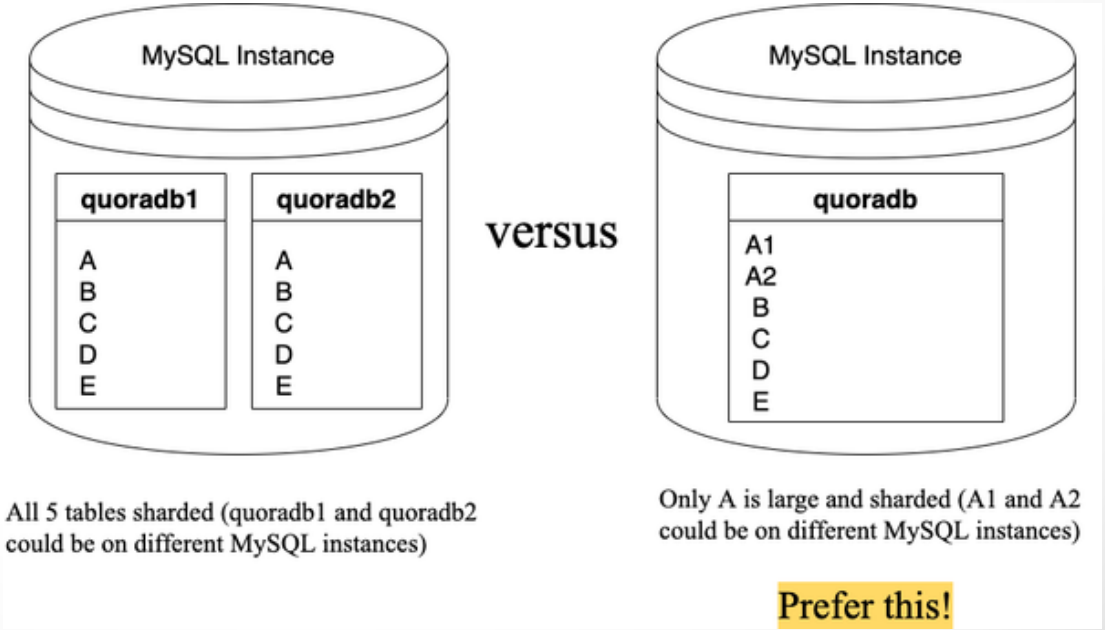
Relational DB. Issues and solutions

- Quora uses MySQL for storage of critical data
- Non-important data are stored in Hbase
- Caching in Redis/Memcache to improve performance
- MySQL sharding to increase QPS (query per second)



Sharding

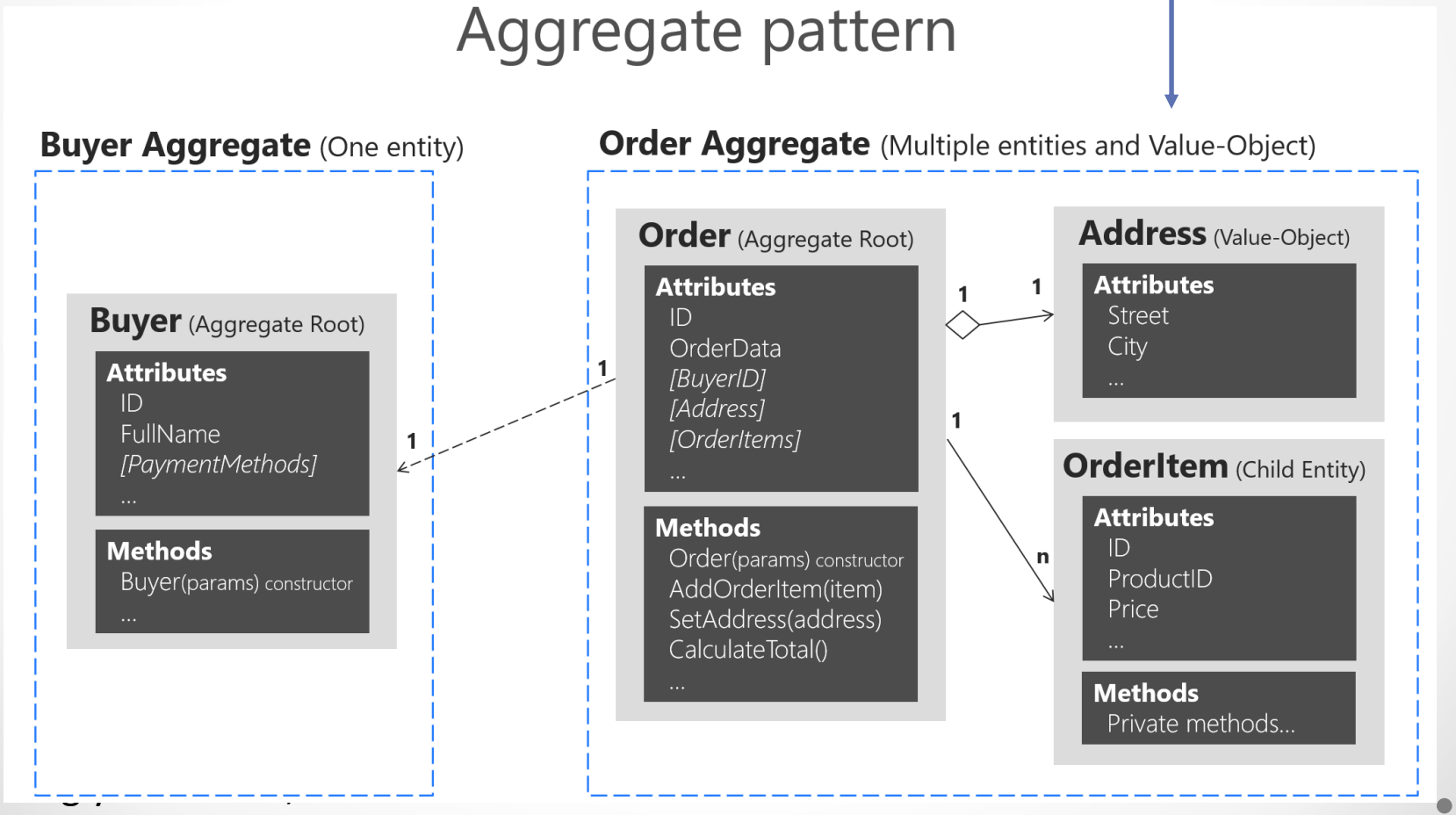
- ✓ Support replication and don’t support joins/transactions between shards
- ✓ Partitions can be geographically distributed
- ✓ In 2011 Facebook has 4000 of MySQL shards



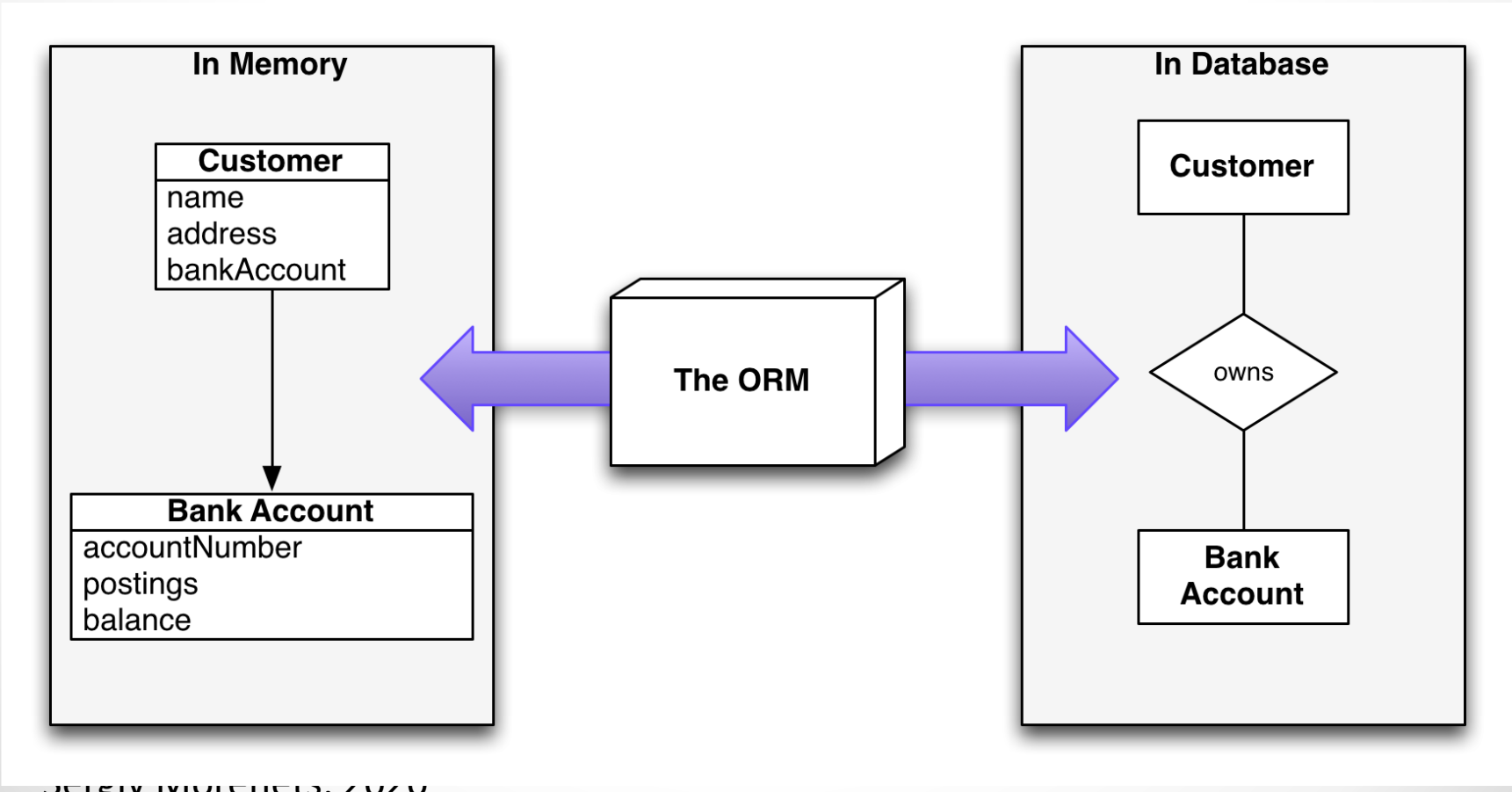
Sergiy Morenets, 2020

Data model. Aggregates

Entities are grouped together because they're used together

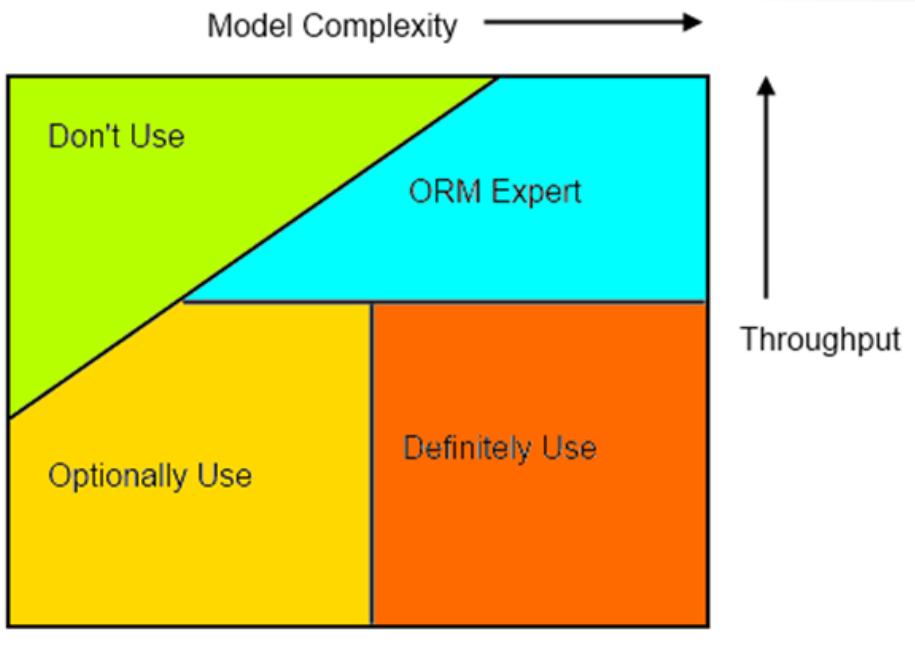


ORM


In Memory
Customer
name
address
bankAccount
Bank Account
accountNumber
postings
balance
The ORM
In Database
Customer
owns
Bank Account

ORM. Challenges

- Database mappings don’t always fit into domain model
- Since each transaction may involve multiple tables/relations it leads to scalability issues
- Additional overhead by replication/sharding requirements



ORM. Challenges

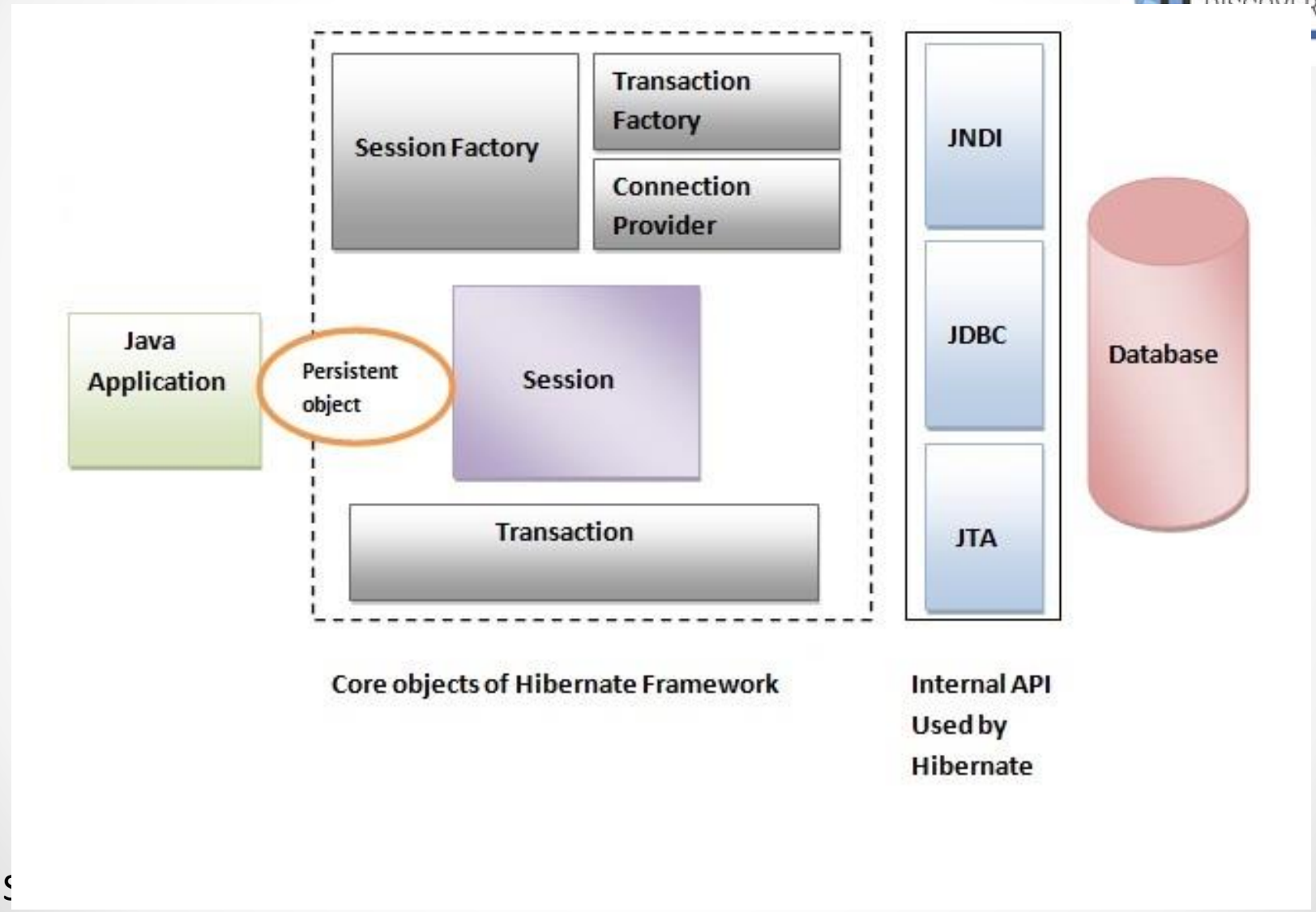

```java
@Entity
@Table
@Getter
@Setter
public class User {

    @Id
    private Integer id;

    private List<String> roles;
```

How to provide ORM mapping?

Requires UserRoles table if roles list is static

Requires additional Roles table if roles list is dynamic

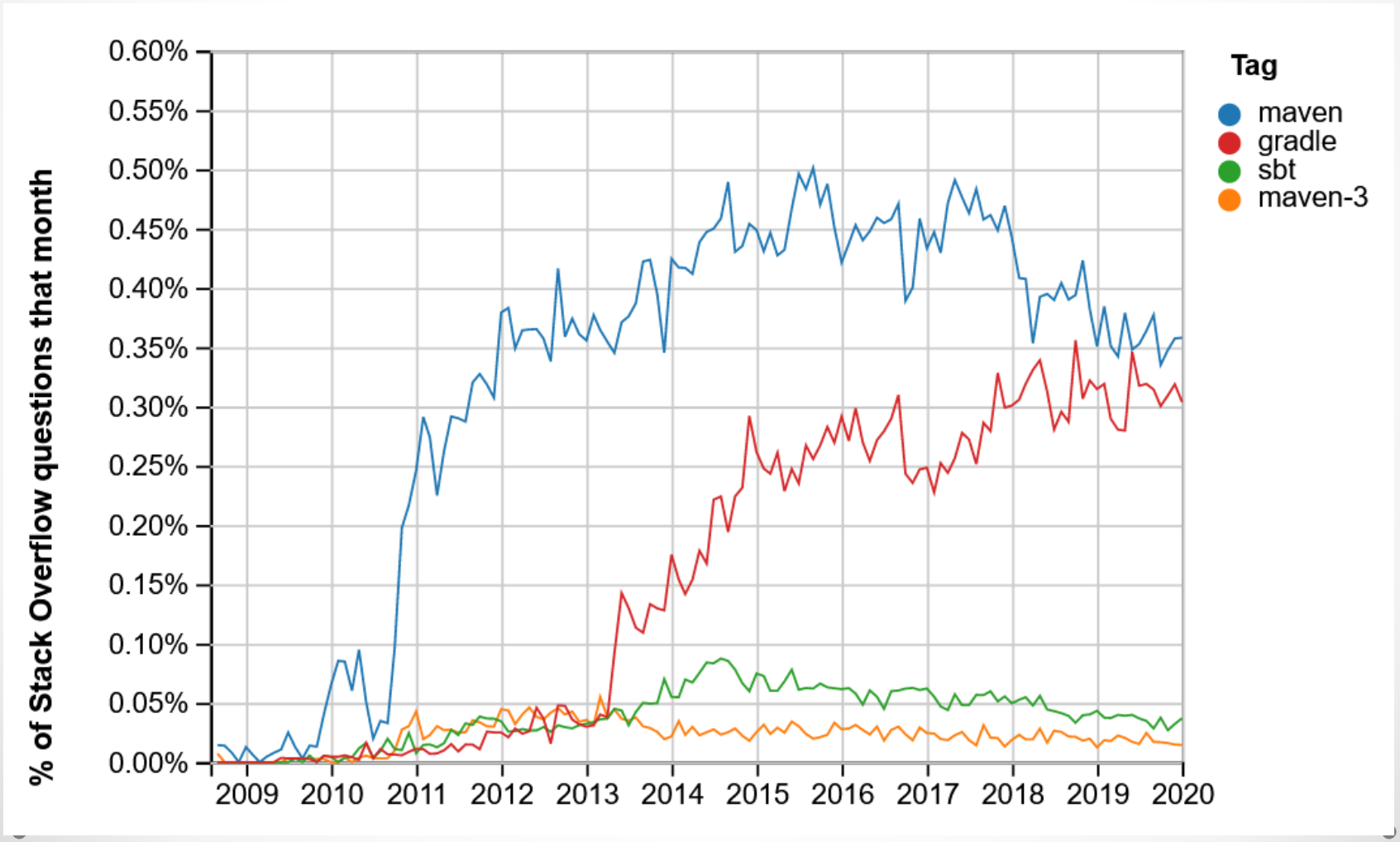
Hibernate


Session Factory
Transaction Factory
Connection Provider
Java Application
Persistent object
Session
Transaction
JNDI
JDBC
JTA
Database
Core objects of Hibernate Framework
Internal API Used by Hibernate

IDE

Build management

Maven vs Gradle vs Sbt



Task 1. Installation & configuration

- You should install Maven 3.6 (or later) or Gradle (6.x or later) on your computer. You should also have the latest installation of JDK 14.
- Import nosql project as **Maven (or Gradle)** projects.
- Review **nosql-task1** project and its domain entities, relationship between them and repositories.
- Review integration tests and run them. Review console output.

Big data revolution

- Soon after Google establishment its developers realized that relational databases are not enough to store indexed data
- Most common issue was scaling and it leads to replication and sharding patterns
- In 2004 MapReduce was introduced
- 2004 started era of JSON usage that was not supported by relational database but supported by document databases
- In 2005 Google introduced Modular data centers
- In 2006 distributed database BigTable (based on Google file system) was announced
- In 2007 Hadoop (based on Nutch project) appeared
- In 2008 Amazon started development of DynamoDB
- In 2010 Hive project proposed SQL-like query language for Hadoop access

NoSQL. History

- NoSQL term was firstly used in 1998 in Strozzi DBMS which was relational but doesn’t use SQL
- Reintroduced in 2009 as “Not Only SQL ”in database event devoted to new non-relational DBMS: DynamoDB and BigTable

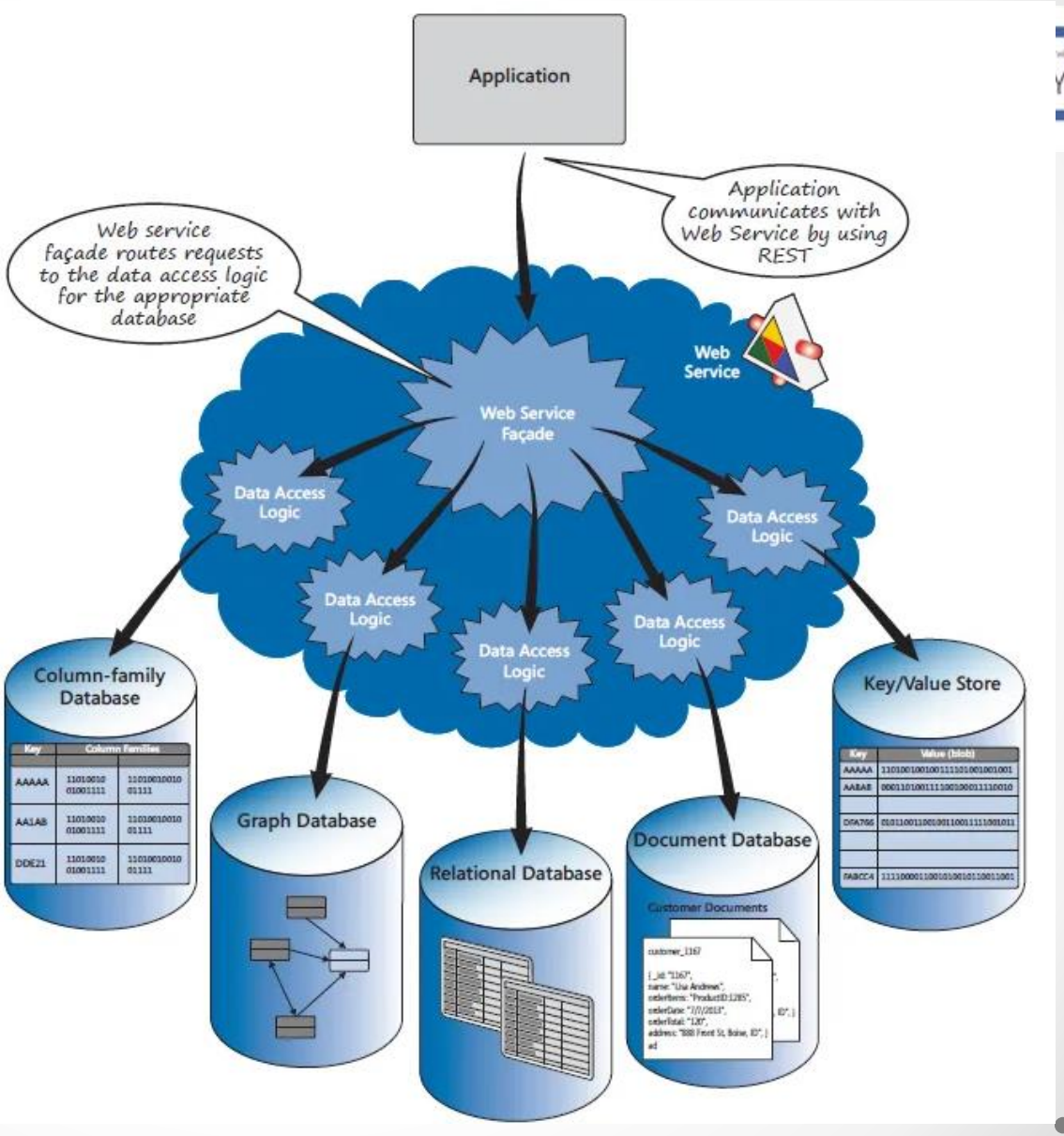
NoSQL database categories

- Key-value
- Document
- Wide column
- Graph

Polyglot persistence

- ✓ Relational database: products/inventories
- ✓ Document store: finalized orders
- ✓ Key-value: sessions, shopping cart
- ✓ Graph database: Customers network
- ✓ Wide columns: Analytics

Polyglot persistence



Sergiy Morenets, 2020

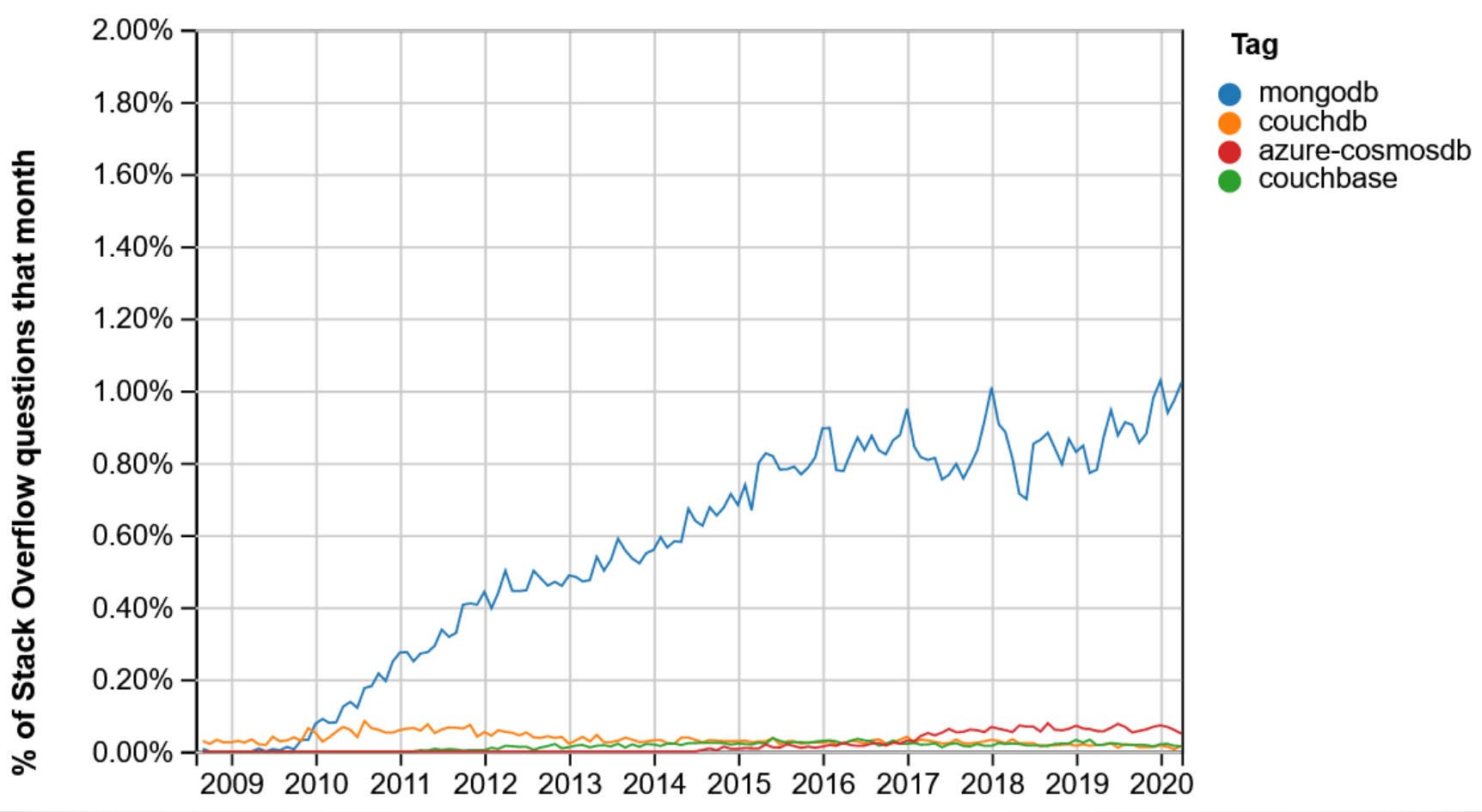
Document store

- Documents (usually JSON objects) similar to rows
- Can be XML-based (eXist, MarkLogic)
- Each document has unique identifier (URI), properties and values
- Properties can be primitive types, arrays, sub-documents or links to other documents
- Good for storing events, CMS-data, web- or real-time analytics or any data with fast-changing structure
- Minimum data migration
- Cloud solutions: Amazon DynamoDB, Microsoft Azure Cosmos DB
- MongoDB, CouchDB (first document db in 2005), Couchbase

Document vs relational database

Relational database	Document
Schema	Database
Table	Collection
Row	Document
Id	_id
Join	DBRef

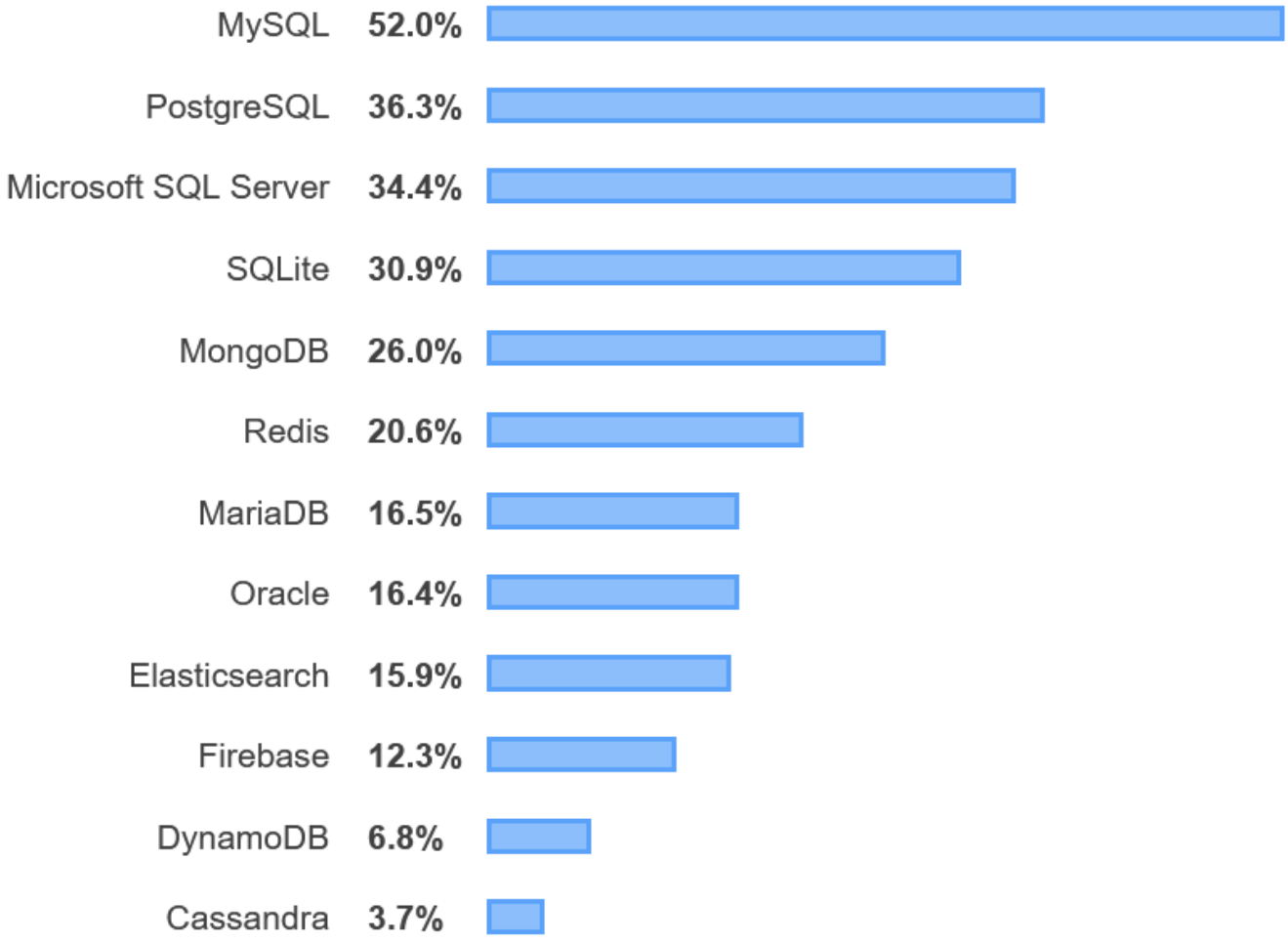
Document store



Sergiy Morenets, 2020

Databases

All Respondents | Professional Developers



Stackoverflow survey 2019

ACID 2.0. BASE

- The BASE acronym was defined by Eric Brewer
- Basically available (admits partial node failures but availability of the whole system)
- Soft state (state may change without external inputs due to data replication)
- Eventual consistency

Sergiy Morenets, 2020

NoSQL databases

- Unstructured data (easy to change the data format)
- No ACID
- Denormalized
- Designed for web-scaled applications (replication & partitioning)
- Geographic distribution
- Eventual consistency
- No standard

amazon

NoSQL vs relational

- Business intelligence applications use models and models rely on schemas
- Business applications require atomic operations
- Relational: transaction, business models, complex data structures
- NoSQL: Web consumers

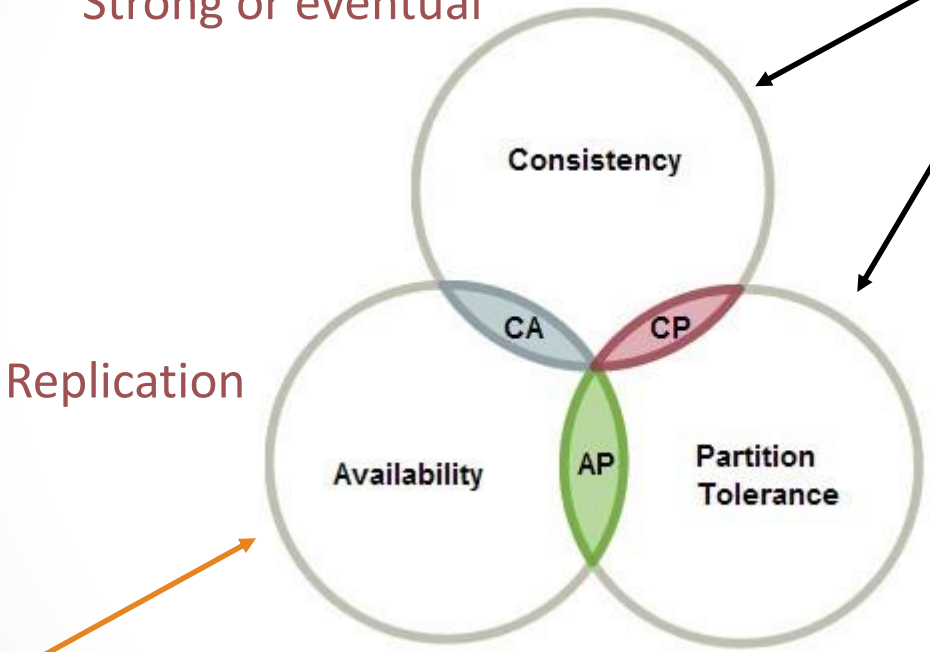
NoSQL data modeling

- NoSQL data model is based on the access patterns
- Data that is loaded together should be stored together
- Analyze search criteria (by key or complex search)
- Write patterns (single or batch updates)

NoSQL data modeling

- ✓ Identity entities and attributes
- ✓ Identifiers and key attributes
- ✓ Define how data is used
- ✓ Group entities
- ✓ Create data model

CAP Theorem



Relational databases

Strong or eventual

Replication

Fault-tolerant

NoSQL

Distribution models

- Single server
- Sharding (horizontal scalability) including auto-sharding
- Master-slave replication, mostly for read-oriented operations
- Peer-to-peer replication where all nodes are able to read/write
- Combination of sharding & replication

Mongo

Sergiy Morenets, 2020

MongoDB

- Development started in 2007 by 10gen company (later renamed to MongoDB Inc)
- First version released in 2009 by Dwight Merriman
- Mongo means humongous (huge + monstrous)
- Community Server, Enterprise Server, Atlas and Stitch editions
- Atlas runs on AWS, Google Cloud and Azure

MongoDB

- Open-source **NoSQL** document database
- Written in C++
- Uses **JSON** documents(in binary-encoded format BSON)
- BSON allows partial (in-place) document update and extra data types (double, date, byte array, JS code, regexp)
- No schemas for document content
- Indexing
- High availability with replications
- Load balancing through sharding
- Aggregation (MapReduce)
- JavaScript–based query language

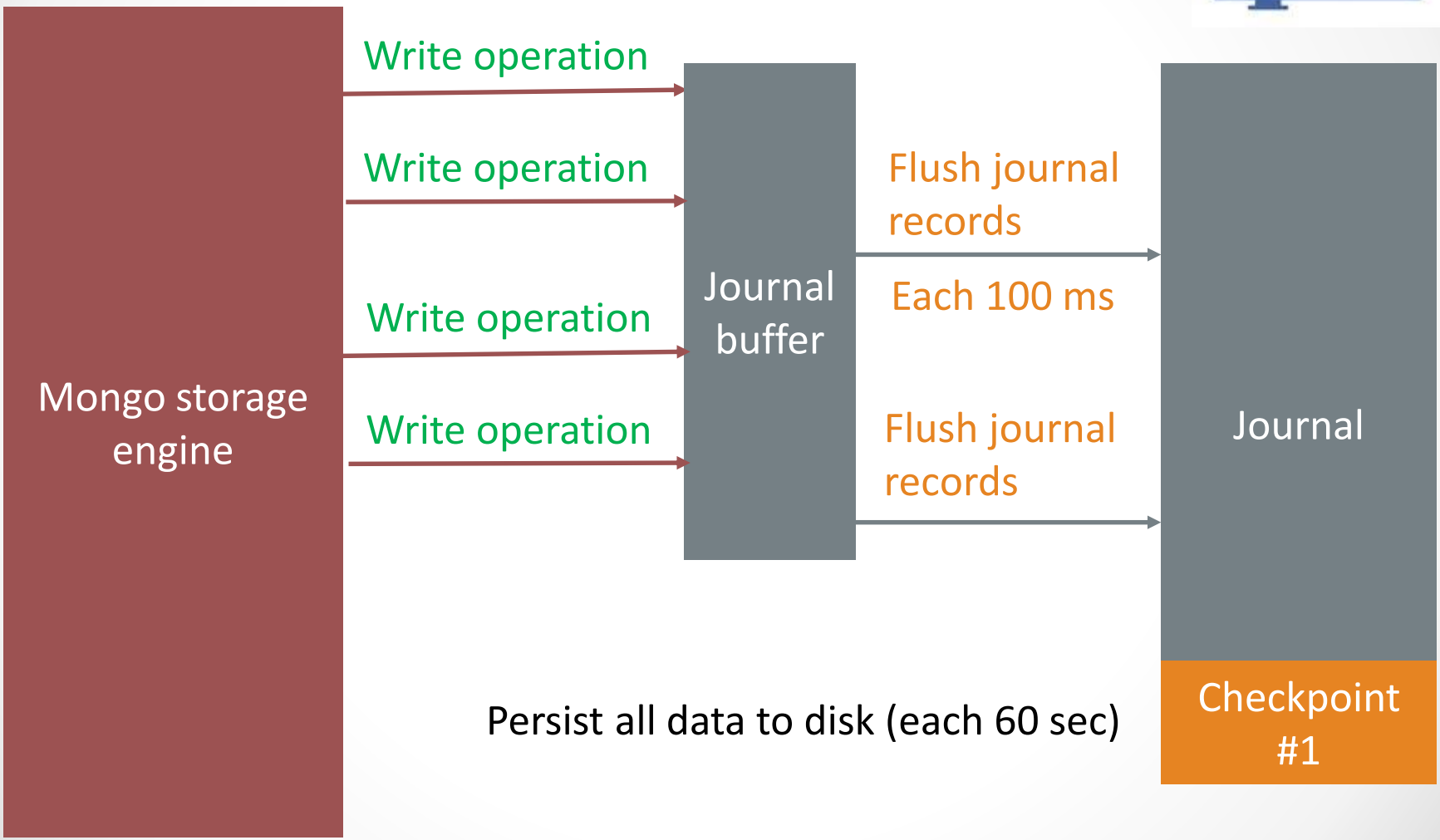
Mongo. Storage & persistence

- Mongo can use three storage engines: in-memory, MMAP and WiredTiger
- Little overhead & good efficiency in marshaling/unmarshalling to C++ types
- Easy traversing the document
- No schema restrictions
- Maximum size of the document 16 megabytes
- GridFS API for extra-large documents

Mongo. Journaling

- Journal (write-ahead log) keeps track of all write operations between checkpoints
- Checkpoints are created each 60 seconds (since 3.6)
- Each write operation is one journal record (minimum size 128 bytes)
- In-memory buffering is used to store journal records
- Journal file size limit is 100 Mb
- Journal records are flushed each 100 ms or if write operation include write concern
- If crash occurred then all the write operations after last checkpoint are restored

Mongo. Journaling & checkpoints



Mongo. MMAP storage engine

- From the beginning Mongo used MMAP-v1 storage engine (memory-mapped files)
- However alternative engines were also supported (like Rocks DB)
- Database-level exclusive write lock and since 3.0 collection-level write lock
- Doesn’t support compression/encryption
- Used all free memory as internal cache
- Supports only 64-bit architecture
- In 4.0 MMAP-v1 was deprecated

Mongo. WiredTiger storage engine

- Introduced in 3.0
- Default engine since 3.2 is WiredTiger
- It enabled multi-document transaction, document-level locking and data compression (collection + indexes)
- Snappy is default compression algorithm. Zlib offers higher compression rate but higher CPU utilization
- Mongo can use internal cache and filesystem cache
- Create checkpoints for write operations and store in the journal files
- Encryption available in Enterprise edition

MongoDB. JSON document


```
{
    '_id' : 1,
    'name' : { 'first' : 'John', 'last' : 'Backus' },
    'contribs' : [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
    'awards' : [
        {
            'award' : 'W.W. McDowell Award',
            'year' : 1967,
            'by' : 'IEEE Computer Society'
        }, {
            'award' : 'Draper Prize',
            'year' : 1993,
            'by' : 'National Academy of Engineering'
        }
    ]
}
```

Mongo. Utilities

- Mongo shell uses SpiderMonkey JavaScript engine written for Mozilla Firefox
- Can use external editor (managed by EDITOR env variable)

Utility	Description
mongod	Mongo daemon (server)
mongo	Mongo shell
mongodump	Binary export of the whole Mongo database
mongoexport	Exports collection data to CSV, TSV, or JSON files
mongostat	Quick overview of Mongo server status
mongos	Utility for managing Mongo sharding instances
mongotop	Per-collection statistics of Mongo I/O usage
mongofiles	Allow to manage GridFS objects

MongoDB. Database methods

Method	Description
adminDatabase()	Runs a command against the admin database
createView()	Creates a view
dropDatabase()	Removes the current database
getCollection()	Returns a collection or view object
getCollectionInfos()	Returns collection information for all collections and views in the current database
hostInfo()	Returns a document with information about the system MongoDB runs on
runCommand()	Runs a database command
shutdownServer()	Shuts down the current Mongo server
stats()	Returns state of the current database.

db.dropDatabase()

Task 2. Database utilities

1. Run Mongo Docker container
2. Create locally new text file data.js
3. Copy data.js to Mongo container and execute this script
4. Run mongo command-line utility in interactive mode
5. Run *db.getMongo()* command to view current database connection
6. Switch to **sample** database

MongoDB. Query methods

SQL	Mongo query syntax
SELECT * FROM Product	db.products.find()
SELECT * FROM Product WHERE name='PC'	db.products.find({name:'PC'})
SELECT id, name FROM Product	db.products.find({}, {id:1,name:1})
SELECT count(*) FROM Product	db.products.count()
SELECT p.* FROM Product p inner join Category c ON c.id=p.category_id WHERE c.name='Computer'	db.products.find({'category.name': 'Computer'})

MongoDB. Document identifiers

- Document must have **_id** field (auto-generated as ObjectId on demand)
- _id can be of text, numeric, date format or complex type
- **ObjectId** is 96-bit number (4 bytes are number of seconds, 3 bytes is machine id, 2 bytes is process id and 3 byte random counter)
- ObjectId is considered globally unique
- UUID can also be used as id (128-bit number) and generated on application-level

MongoDB. Query methods

Method	Description
find({ “name” : “test” })	Executes query to return collection
findOne()	Returns single document
drop()	Remove collection
insert({ “name” : “Microservices”})	Inserts new document
update({ name : “Phone”}, { name: “PC” })	Modifies document in the collection
count()	Calculates number of the documents in the collection
remove()	Clears collection
renameCollection()	Changes name of the collection

db.users.find({})

users is collection name

MongoDB. Create a collection


```
db.createCollection('products', {
        collation: {
            locale: 'en_US',
            caseFirst: 'lower'
        },
        validator: {
           $jsonSchema: {
              bsonType: "object",
              required: [ "name", "year", "major", "address" ]
           }
        },
        validationLevel: 'strict',
        validationAction: 'warn',
        writeConcern: {
          w: 1
        }
    })
```

Collation for indexing
(if omitted then binary collation used)

Validation document schema

Keep track of all inserts/updates

Log invalid documents

Sync up journal records immediately

MongoDB. Capped collections

- ✓ Fixed-sized collections
- ✓ Prevent insertion order (no indexes needed)
- ✓ Similar to circular buffer (new documents will override old documents)
- ✓ It's not possible to delete single document, use write-operation transactions or shard capped collection

```
db.createCollection('products', {
        capped: true,
        size: 100000,
        max: 1000
    })
```

Maximum size in bytes

Maximum number of documents

MongoDB. Operators

Operator	Description
$eq	Specifies equality condition
$gt	Allows to find document(s) where field value is greater than argument
$in	Specifies set of matching values for a field
$lt	Allows to find document(s) where field value is less than argument
$ne	Specifies not equality condition
$and	Logical AND operation on an array of one or more expressions
$not	Logical NOT operation on the specified expression
$or	Logical OR operation on an array of two or more expressions

MongoDB. Operators

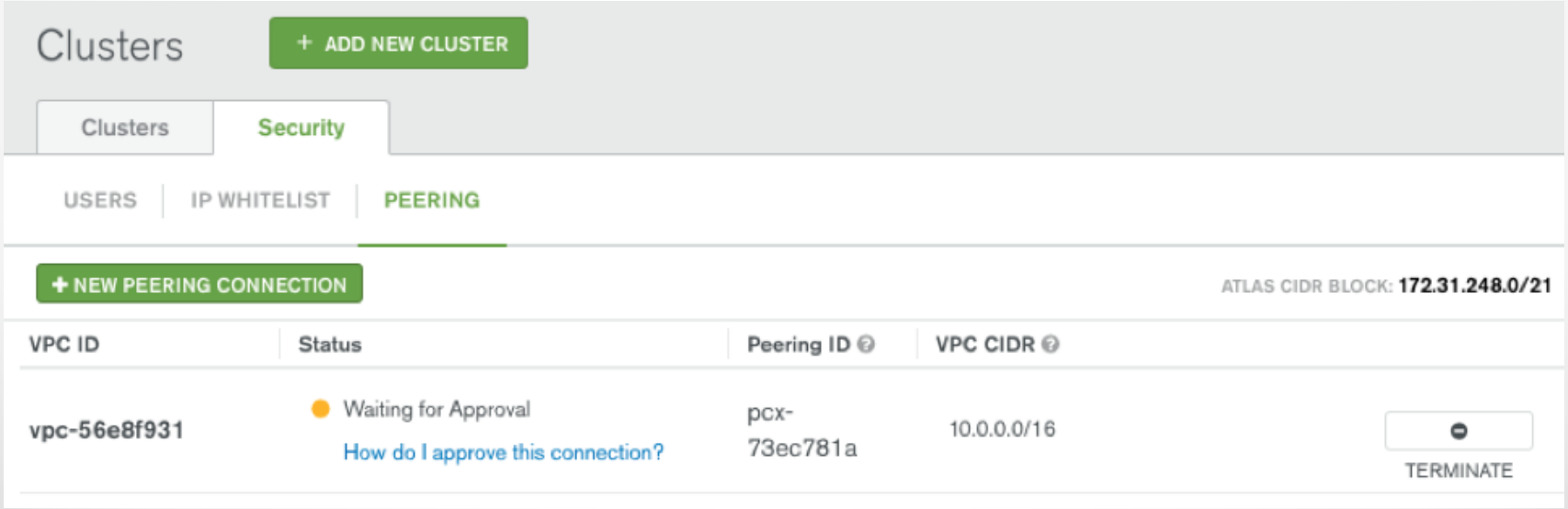
Operator	Description
$exists	Matches the documents that contain the field, including documents where the field value is null
$regex	Provides regular expression capabilities for pattern matching strings in queries
$all	Selects the documents where the value of a field is an array that contains all the specified elements
$elemMatch	Matches documents that contain an array field with at least one element that matches all the specified query criteria.
$size	Matches any array with the number of elements specified by the argument

Task 3. Mongo shell

1. Run mongo command-line utility in interactive mode
2. Insert new book document
3. Print all the existing books. What fields are returned from the database? Use pretty printing
4. Try to search books by name or find books where title is not “Microservices”.Is searching by properties case-sensitive?
5. Try to find books by **publisher** name
6. Print the books number

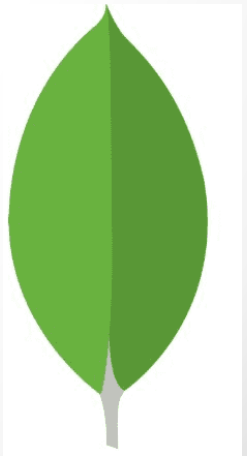
MongoDB Atlas

- ✓ Database-as-a-service
- ✓ Contains MongoDB, Realm backend service and UI tools (charts, search)
- ✓ Supports AWS, Azure and Google Cloud

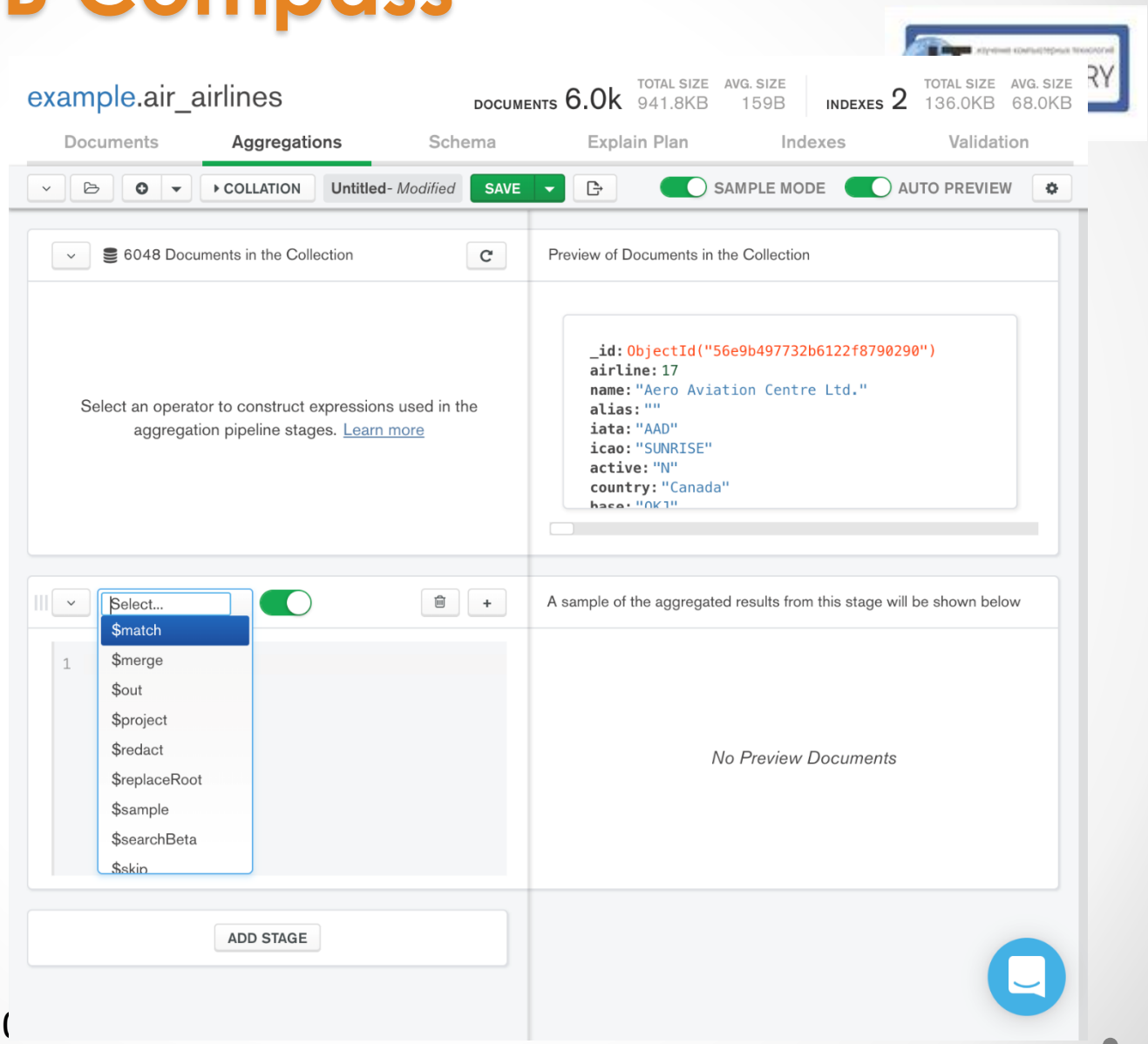


MongoDB Compass

- Compass, Compass Readonly and Compass Isolated editions
- MongoDB data explorer
- No need to know Mongo query syntax
- Allows to optimize query performance, manage indexes, and implement document validation
- Supports data export/import (CSV or JSON)
- Aggregation pipeline builder



MongoDB Compass

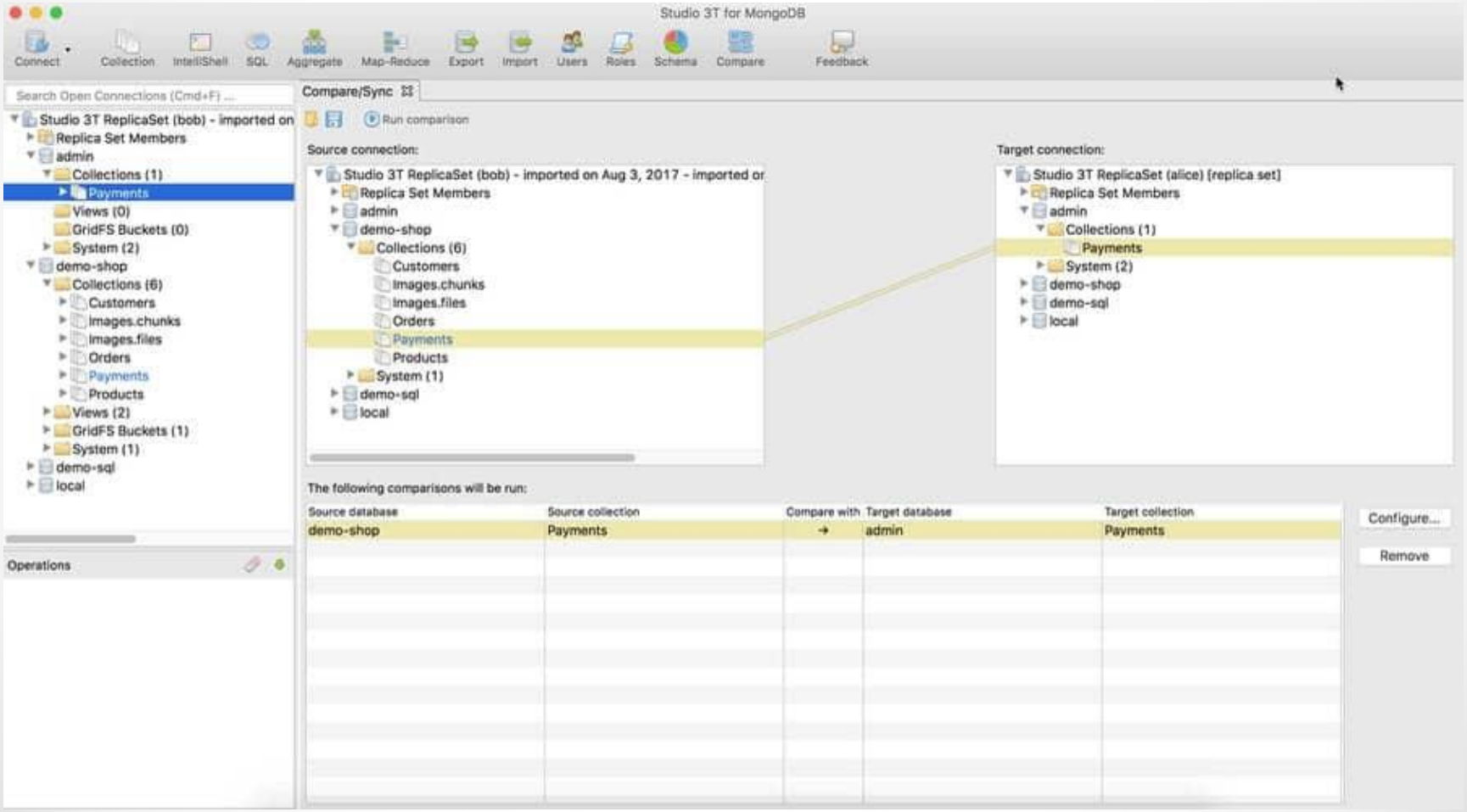


Sergiy Morenets, 2(

Robo 3T

- Merging of Robomongo and 3T Software Labs companies
- Commercial Studio 3T and free Robo 3T versions
- Visual Query Builder
- IntelliShell
- SQL to MongoDB migrations
- Import/Export wizard
- Aggregation Editor
- Schema explorer/validator
- GridFS support
- Map-Reduce support

Robo 3T/Studio 3T



Sergiy Morenets, 2020

Mongify

- Data translator system for moving your SQL data to MongoDB
- Written in Ruby
- Supports MySQL, PostgreSQL, SQLite, Oracle, SQLServer, and DB2
- Supports any version of MongoDB

mongify

Sergiy Morenets, 2020

Mongify


```
sql_connection do
  adapter    "mysql"
  host       "localhost"
  username   "root"
  password   "passw0rd"
  database   "my_database"
end

mongodb_connection do
  host       "localhost"
  database   "my_database"
end
```

database.config

```
table "users" do
  column "id", :key
  column "first_name", :string
  column "last_name", :string
  column "created_at", :datetime
  column "updated_at", :datetime
end
```

translation.rb

mongify check database.config

mongify translation database.config

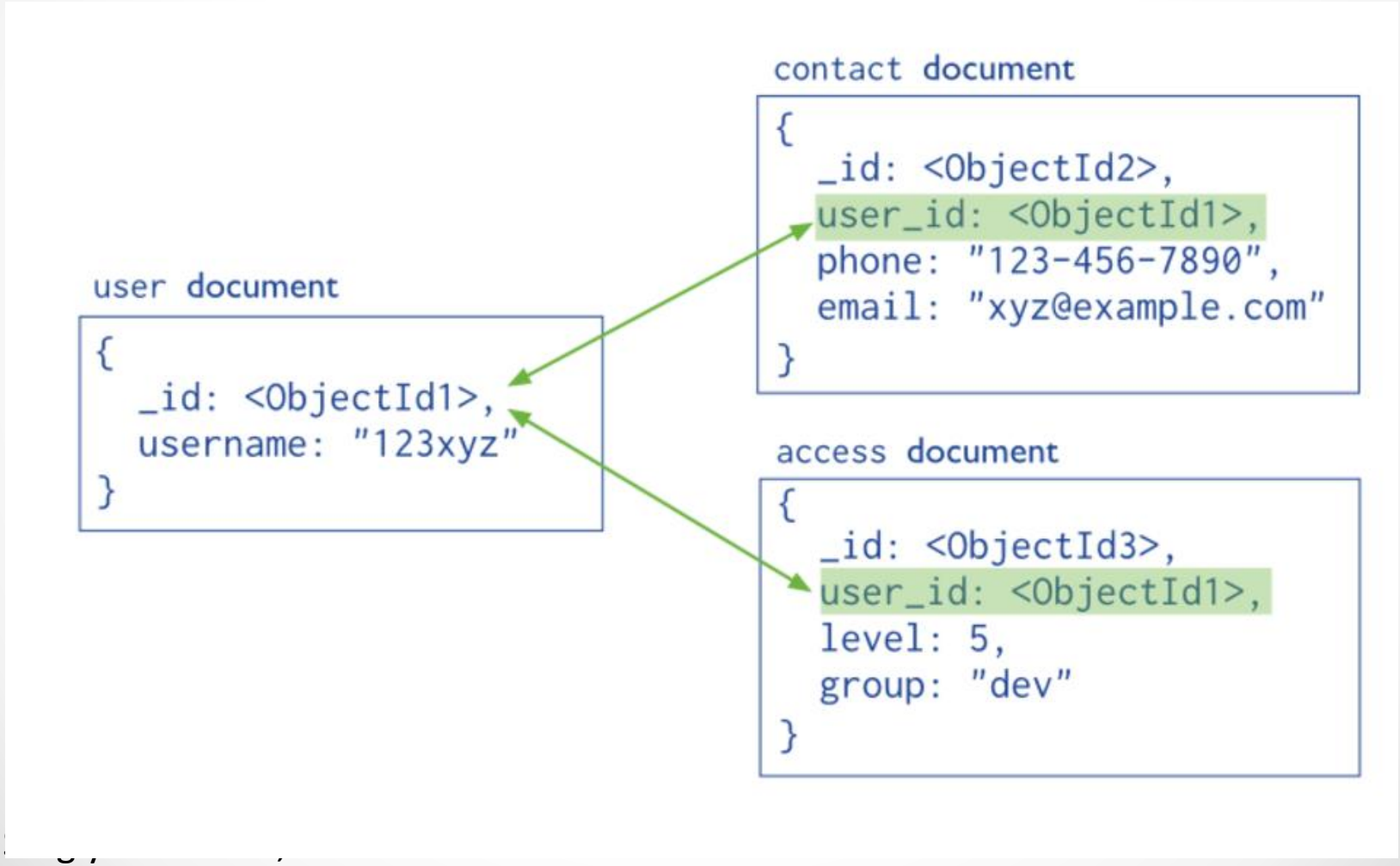
mongify process database.config translation.rb

Sergiy Morenets, 2020

Task 4. Mongo GUI

1. Download MongoDB Compass
2. Connect to local Mongo database. Try to view sample database structure, add/view documents in books collection.
3. Download and install Robo 3T
4. Run Robo 3T and connect to local Mongo database. Try to view/insert/delete documents in books collection.

Mongo. Embedding vs Referencing



Mongo. Embedded vs Referencing

- ✓ Data that references (loaded) together should be embedded
- ✓ Dependent entities (for example, one-to-one) can be embedded
- ✓ Entities updated at the same time should be embedded
- ✓ Independent entities should be referenced

Mongo. Embedding vs Referencing

Embedding

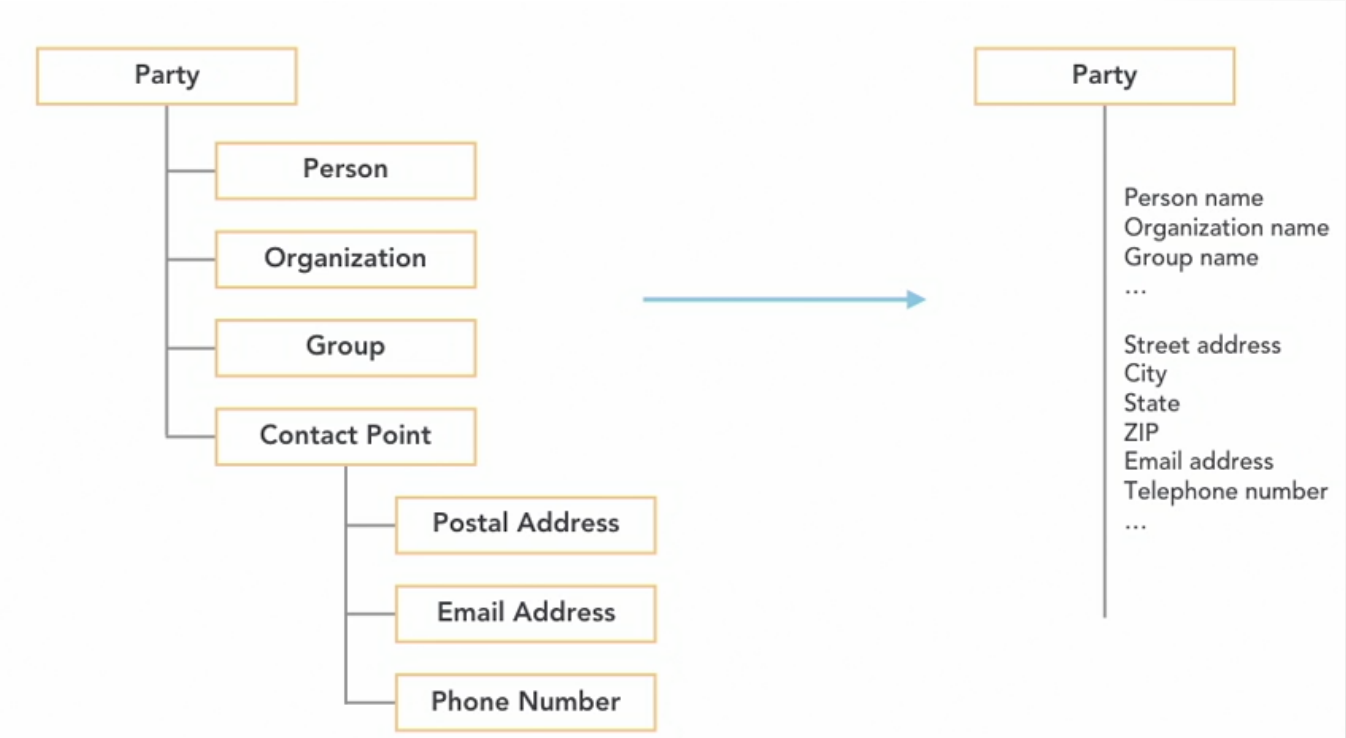
- Single query to load all data
- No lookups or joins
- Atomicity when updating all data

Referencing

- Small documents (16MB limit)
- No duplication
- Avoid loading frequently-used data

Document flattening

- ✓ Remove object structure and expose attributes directly
- ✓ Turn arrays into fixed set of attributes
- ✓ Simplify the document



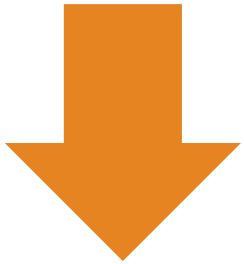
Anti-patterns

- ✓ Can cause performance issues
- ✓ Use one-to-many relationship when parent document has huge number of child documents
- ✓ High-load data transfer when reading the document
- ✓ Resolution – invert relationship

Invert relationship


```
public class Bank {
    private String name;

    private List<Account> accounts;
}
```

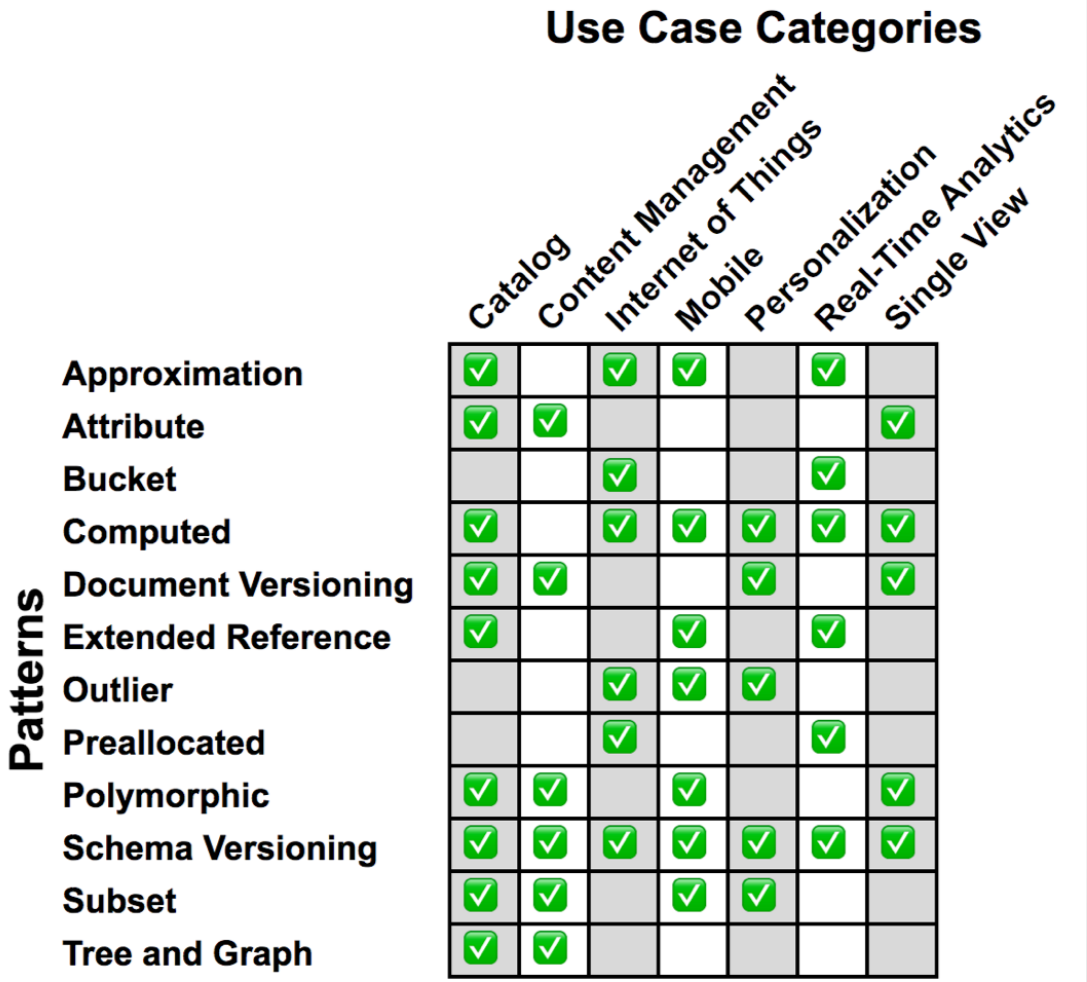
Thousands of entries



```
public class Account {
    private String id;

    private double amount;

    private Bank bank;
}
```

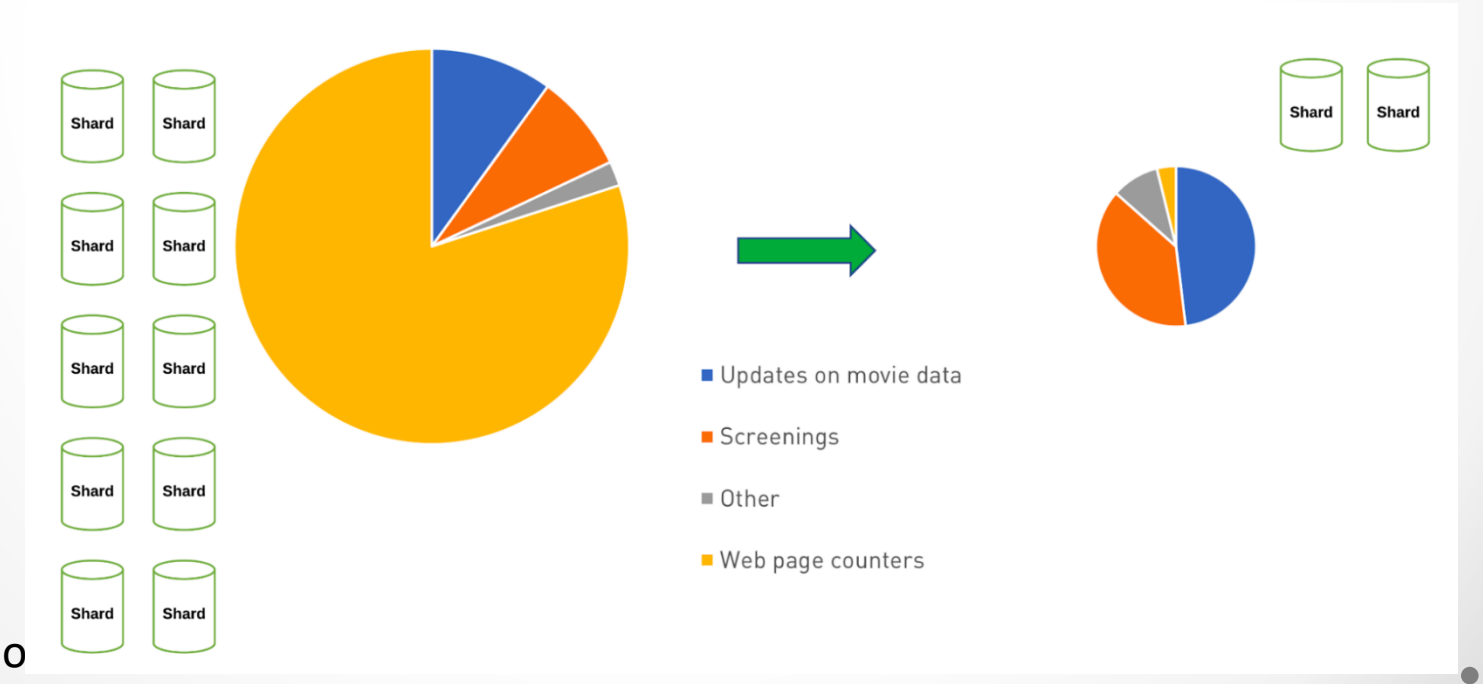
Separate collection

Mongo. Design patterns



Approximation pattern

- Used for resource-expensive calculations when precision is not the top-priority
- When we hit approximation factor(threshold) we issue calculate(write) request



Attribute pattern

- We have to query/sort documents that have similar subset of fields
- Creating indexes on all needed fields can affect performance

```
public class Movie {
    private String titleEn;

    private String titleFr;

    private String titleCh;
}
```

We need to search/sort by all title fields

Require indexes (can impact performance)

Attribute pattern


```
class Translation {
    private String text;

    private String locale;
}
```

```
public class Movie {

    private List<Translation> titles;
```

Require single index

Bucket pattern

- ✓ Useful for real-time analytics events
- ✓ Group data by time intervals

```
public class Temperature {
    private String locationId;

    private double value;

    private LocalDateTime created;
}
```

Require created/movieId indexes

Hard to scale

Bucket pattern


```
class Bucket {
    private String locationId;

    private LocalDateTime startDate;

    private LocalDateTime endDate;

    private int valueCount;

    private double totalValue;

    private List<BucketValue> values;
}
```

Measurement interval (1 hour/day/month)

Easy to get average value

```
class BucketValue {
    private double value;

    private LocalDateTime created;
}
```

Computed pattern

- ✓ Some information is rarely updated but frequently read
- ✓ Financial data, event sourcing, product catalogs

```
public class Rating {
    private String movieId;

    private String userId;

    private int rate;

    private LocalDateTime created;
}
```

We need to show average rating for the movie page(s)

```
public class Movie {
    private double rating;
```

Re-calculate each hour(day)

Document version pattern

- ✓ Keep history of the documents
- ✓ Suppose each document has small amount of revisions
- ✓ Most queries require current revision of the document

```
public class Agreement {
    private String text;
```

```
public class User {
    private Agreement agreement;
```

```
public class Agreement {
    private String text;

    private int revision;

    private LocalDateTime updated;
```

Up-to-date agreement version

History of agreements should be maintained

Incremented agreement revision

Document version pattern


```
public class User {
    private Agreement agreement;

    private List<Agreement> agreements;
```

Alternate option is
to keep it in separate collection

Extended reference pattern

- ✓ Keep copy of external data instead of keeping the reference
- ✓ Keep frequently accessed data

```
public class Movie {
    private List<String> actorIds;
```

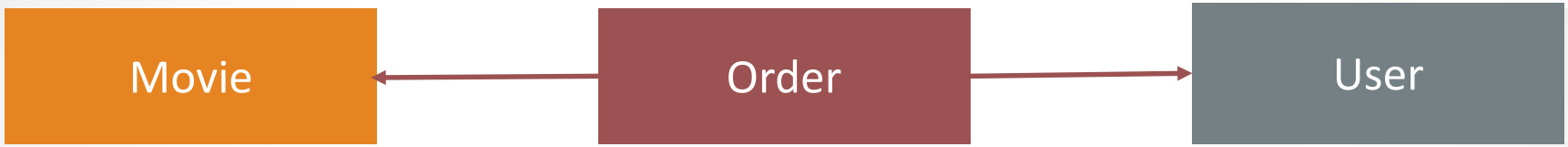
Require additional request(s) to Actor collection

```
class ActorInfo {
    private String id;

    private String name;
}
```

```
public class Movie {
    private List<ActorInfo> actorIds;
```

Needs update if actor name has changed

Outlier pattern




```
public class Movie {
    private List<String> userIds;
```

List of users who purchased this movie

Suitable for recommendations

For top movies list can reach millions of elements

```
public class Movie {
    private List<String> userIds;

    private boolean hasExtras;
```

Flag that information is partially stored in difference place

Preallocation pattern

- Used when structure of the document is fixed and can be preallocated
- Relocating growing documents is pretty expensive (till v 4.0) because it requires index re-build

```
public class Trip {
    private int maxSeats;

    private List<Integer> seats;
}
```

List of occupied seats (seat numbers)

```
private List<Integer> seats;

public Trip() {
    seats = Stream.of(0, maxSeats)
            .collect(Collectors.toList());
}
```

Initialize all the seats when trip is created. Now seats can contain ticket ids

Polymorphic pattern

- ✓ Documents have similar structure but differences as well
- ✓ Instead of grouping documents by collections we can combine them in single collection

```
public abstract class Event {
    private String id;

    private LocalDateTime startDate;

    private LocalDateTime endDate;
}
```

```
class Workshop extends Event {
    private double price;

    private String promoCode;
}
```

```
class Webinar extends Event {
    private String onlineUrl;
}
```

Subset pattern

- Some large documents can contain data which are not fully used
- In that case it's too expensive to store (cache) these documents in memory

```
public class User {
    private List<Page> viewedPages;
```

Can contain hundreds (or thousands) of entries

```
public class User {
    private List<Page> recentPages;
```

Store only most recent (most important) page views

```
public class Page {
    private String id;

    private String userId;
```

Store all page views

Tree pattern

- ✓ Suitable for hierarchical data
- ✓ Allows partial copy-paste to speed up the search

```
public class Category {
    private String id;

    private String name;

    private String parentId;
```

Standard tree data model

```
public class Category {
    private String id;

    private String name;

    private String parentCategory;

    private List<String> parentCategories;
```

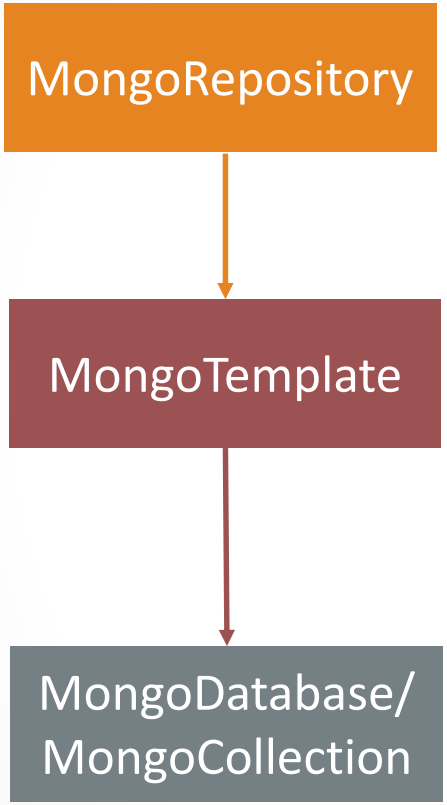
Store names of all parent Categories (speed up the Search if we use indexes)

Needs to be updated if we change the hierarchy

Task 5. Data model

1. Open **nosql-task5** project and review project structure/configuration and data model.
2. Which document store features can simplify data model? Which features of other NoSQL database categories would you use?
3. Change the data model based on MongoDB patterns/anti-patterns

MongoDB Access API



Spring Data MongoDB.
High-level abstraction

Spring Data MongoDB.
Low-level abstraction

MongoDB Java driver

MongoDB. Drivers

Driver	Description
BSON Library	A standalone BSON library, with a new Codec infrastructure that you can use to build high-performance encoders and decoders without requiring an intermediate Map instance
MongoDB Driver	An updated Java driver that includes the legacy API as well as a new generic MongoCollection interface that complies with a new cross-driver CRUD specification
MongoDB Reactive Streams Driver	Providing asynchronous stream processing with non-blocking back pressure for MongoDB. Fully implements the Reactive Streams API for providing interop with other reactive streams within the JVM ecosystem.

Spring Data Modules

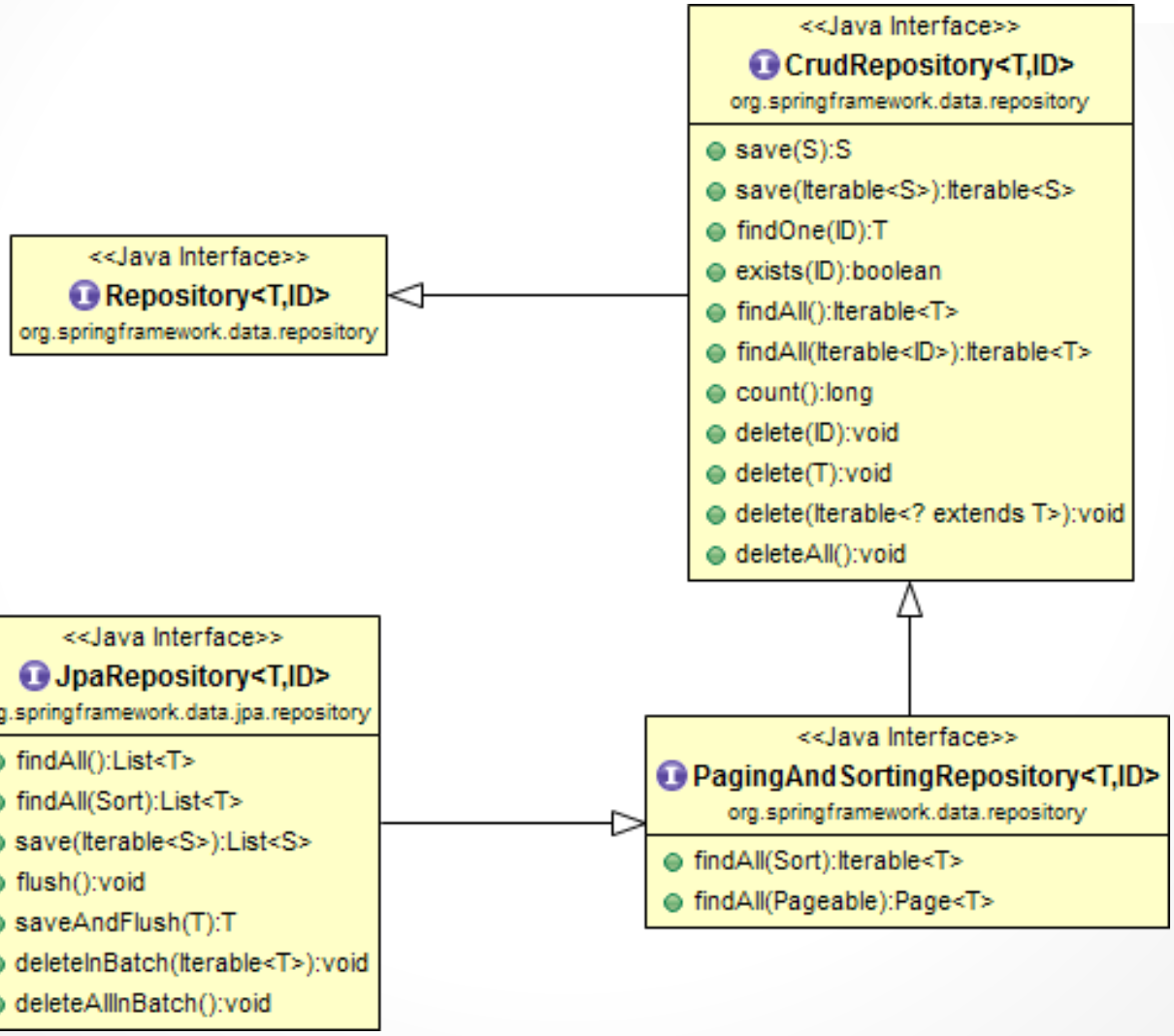
Spring Data

- ✓ Started in 2008
- ✓ Reduces programming effort and avoids boilerplate code
- ✓ Dynamic query construction
- ✓ Easy Spring integration
- ✓ Includes integration with JDBC/Mongo/Redis/REST/ Cassandra/Couchbase/ElasticSearch/Hadoop/Neo4j
- ✓ Current version 2.3.0

Spring Data for Apache Solr Discontinued

ENGINEERING | CHRISTOPH STROBL | APRIL 07, 2020 1 COMMENT

Spring Data JPA. Hierarchy



CrudRepository. Methods

Method	Description
save	Saves given entities/entity
findById	Retrieves an optional entity by its id
findAll	Returns all instances of the type
count	Returns the number of entities available
delete	Deletes the entity with the given id
existsById	Returns whether an entity with the given id exists
deleteAll	Deletes all entities managed by the repository
deleteById	Deletes an entity by its id

Spring Data JPA. Repositories


```
public interface ProductRepository extends
    CrudRepository<Product, Integer> {

    public List<Product> findByName(String name);

}
```

SELECT … FROM Product WHERE name= …

JPQL parameter

Run-time query (built dynamically based on input parameters)

```
Product product = new Product();
product.setName("phone1");
Example<Product> example = Example.of(product);

List<Product> items = productRepository.findAll(example);
```

Use all non-null properties

Spring Data MongoDB

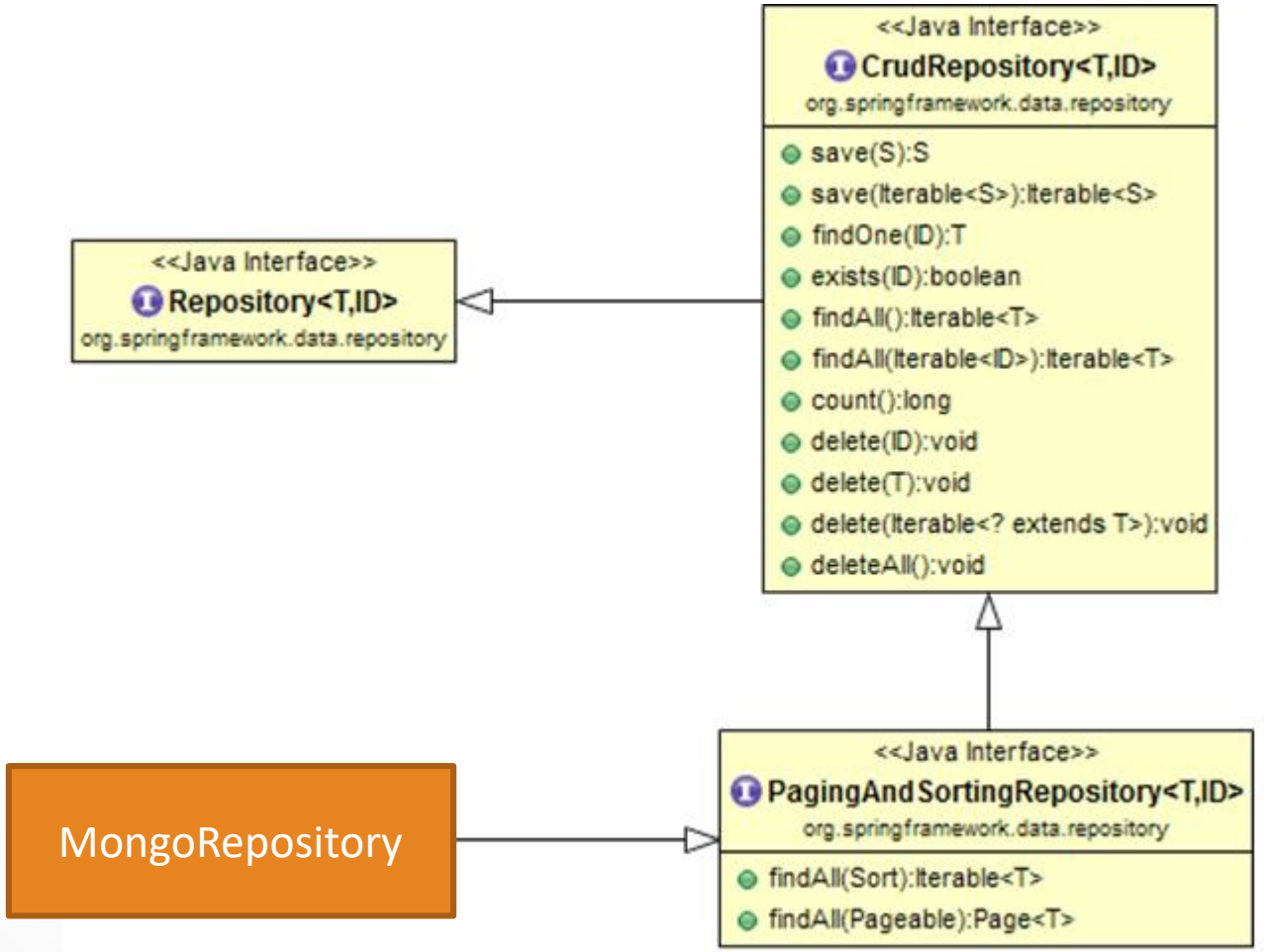
- ✓ Spring Data sub-project
- ✓ MongoTemplate for operations
- ✓ Java-based Query/Criteria
- ✓ Automatic repository implementation
- ✓ Cross-store persistence
- ✓ Map-Reduce integration
- ✓ Uses MongoDB Java driver (sync, async or reactive-streams)

Spring Data MongoDB. Maven


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
    <version>2.3.0.RELEASE</version>
</dependency>
```

Depends on Spring Data MongoDB 3.x and MongoDB driver 4.x

Spring Data. Hierarchy



Spring Data MongoDB. Repositories


```
@Document(collection = "products")
public class Product {
    @Id
    private String id;

    private String name;

    private double price;
}
```

Also supports ObjectId, BigInteger

```
public interface ProductRepository extends
            MongoRepository<Product, String> {

}
```

Spring Data MongoDB. Repositories


```
@RestController
@RequestMapping("products")
@RequiredArgsConstructor
public class ProductController {

    private final ProductRepository productRepository;

    @GetMapping
    public List<Product> findAll() {
        return productRepository.findAll();
    }
}
```

Spring Data MongoDB. Repositories


```
public interface ProductRepository extends
            MongoRepository<Product, String> {

    List<Product> findByName(String name);
```

Logging

find using query: { "name" : “PC"}

```
@Query("{name: ?0}")
List<Product> find(String name);
```

JavaScript Mongo query

```
@Query("{ratings: {$exists: false}}")
List<Product> findWithoutRatings();
```

Field is missing

```
@Query("{ratings: {$exists: true, $ne: []}}")
List<Product> findWithRatings();
```

ratings is not null and not empty

Spring Data MongoDB. Audit


```java
@CreatedDate
private LocalDateTime createdAt;

@LastModifiedDate
private LocalDateTime modifiedAt;
```

```java
@SpringBootApplication
@EnableMongoAuditing
public class MongoApplication {
```

Enable auditing fields

Spring Data MongoDB. Configuration


```yaml
spring:
  data:
    mongodb:
      database: warehouse
      host: localhost
      username: user
      password: p@wd
      port: 27017
      auto-index-creation: true
```

application.yml

Task #6. Spring Data MongoDB

1. Open **application.yml** file in src/main/properties folder and update Mongo connection settings
2. Review data model and add necessary annotations
3. Add **@EnableMongoAuditing** annotation on Spring Boot configuration class
4. Update repository interfaces and extend them **MongoRepository** interface.

JUnit 5

- JUnit 5 = Platform + Jupiter + Vintage
- Separation of concerns
- API improvements, test extension model
- Supports repeated, parametrized and dynamic tests
- New assumptions concept
- Java 8 - based

JUnit 5

A first test case

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

    @Test
    void myFirstTest() {
        assertEquals(2, 1 + 1);
    }

}
```

Spring Boot Test Starter

- ✓ Includes Spring Test Framework
- ✓ Junit 5 API (annotations, assertions, assumptions)
- ✓ Junit 5 Jupiter Engine
- ✓ Junit 5 parametrized tests
- ✓ Junit 5 Vintage (Junit 4)
- ✓ Mockito + integration with Junit
- ✓ AssertJ + Hamcrest

Upgrade spring boot starter test to JUnit 5 #14736

Closed **philwebb** opened this issue on Oct 9, 2018 · 16 comments

Spring Boot Starter Test


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>${spring.boot.version}</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Spring Data Mongo. Integration tests


```
@DataMongoTest
public class ProductRepositoryTest {

    @Autowired
    ProductRepository productRepository;

    @Test
    void saveProduct_validDocument_success() {
        Product product = productRepository.save(
                new Product("Phone", 100));
        assertNotNull(product.getId());
        assertEquals(100, product.getPrice());
    }
}
```

Mongo. Usage in integration tests

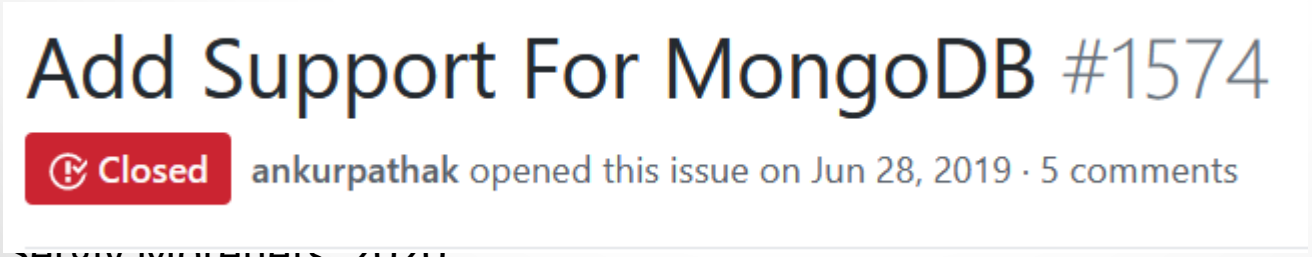
- **Embedded Mongo** project is not officially supported by MongoDB
- Required Java8+ source compatibility
- Hasn’t maintained since 2018 (Mongo 4.x is not supported)
- Natively integrates with Spring Boot/Spring Test
- **Fongo** mocks (catches) all Mongo Drivers calls (not maintained now)
- **MongoDB Java server** is in-memory fake implementation of MongoDB
- Full-text search or map/reduce are not supported

Test containers

- Java library that is wrapper around existing Docker client
- Written in Java 8
- Support Junit 4/5, Spock
- Requires Docker 1.12 and later
- Allows to simplify DAO/application integration tests, UI acceptance tests

Test containers

- ✓ Supports generic containers
- ✓ Relational databases (MySQL, Postgres, SQL Server, Oracle,DB2)
- ✓ **NoSQL** databases (Cassandra, OrientDB, Couchbase, Neo4j)
- ✓ Cloud solution support (ElasticSearch, LocalStack, Kafka, RabbitMQ)
- ✓ Selenuim/nginx
- ✓ Docker-compose support



Test containers. Build support


```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.14.1</version>
    <scope>test</scope>
</dependency>
```

Generic containers usage

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.14.1</version>
    <scope>test</scope>
</dependency>
```

Junit 5 support

Test containers. Mongo usage


```
@DataMongoTest
@Testcontainers
public class ProductRepositoryTest {

    @Container
    GenericContainer mongo = new GenericContainer<>("mongo:4")
            .withExposedPorts(27017);
```

Mapped port is random

Container is started before tests and stopped after tests

Make use of @DynamicPropertySource in our integration tests #16886

Closed **wilkinsona** opened this issue on May 17, 2019 · 18 comments

Test containers. Mongo usage


```
@Container
static GenericContainer mongo = new
        GenericContainer<>("mongo:4")
        .withExposedPorts(27017);
```

```
@DynamicPropertySource
static void mongoProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.data.mongodb.port",
            () -> mongo.getMappedPort(27017));
}
```

Override Spring Data MongoDB port based on mapped port

Test containers. Docker Compose support


```
@DataMongoTest
@Testcontainers
public class ProductRepositoryTest {

    @Container
    static DockerComposeContainer environment =
            new DockerComposeContainer<>(
            new File("src/test/resources/docker-compose.yml"))
            .withLocalCompose(true)
            .waitingFor("mongo1",
                new DockerHealthcheckWaitStrategy());
```

If Docker Compose installed locally

Wait until container becomes healthy

Task 7. Integration tests

1. Review integration tests. Add **@DataMongoTest** and @TestContainers annotations to the test classes
2. Add new static field **mongo** of GenericContainer type that should refer to Mongo 4.x Docker image
3. Update MongoDB port property for Spring Data using @DynamicPropertySource annotation
4. Uncomment all commented-out code in in the integration tests

MongoTemplate

Method	Description
findOne	Find one document by criteria query
findAll	Query for a list of objects from the collection
Insert	Insert the object into the collection
updateFirst	Updates the first object that is found in the collection that matches the query document
updateMulti	Updates all objects that are found in the collection for that matches the query document criteria
save	Save the object to the collection
upsert	If no document is found that matches the query, a new document is created and inserted
count	Returns the number of documents of the entity class
exists	Define if result of given query contains at least one element

MongoTemplate. Examples


```
MongoOperations operation = new
        MongoTemplate(MongoClients.create(),
        "shop");
```

Mongo driver API

Database name

```
operation.insert(new Product("PC", 100));
```

'{"insert": "products", "ordered": true, "$db": "shop", "documents": [{"_id": {"$oid": "5ebee63d92543f6acd623f97"}, "name": "PC", "price": 100.0, "_class": "it.discovery.model.Product"}]}'

```
Product product = operation.findOne(new Query(
        Criteria.where("name").is("PC")),
        Product.class);
```

{"find": "products", "filter": {"name": "PC"}, "limit": 1, "singleBatch": true, "$db": "shop"}'

MongoTemplate. Examples


```
@RestController
@RequestMapping("products")
@RequiredArgsConstructor
public class ProductController {

    private final MongoOperations mongoOperations;

    @GetMapping
    public List<Product> findAll() {
        return mongoOperations.findAll(Product.class);
    }
```

Created automatically
based on application properties

Mongo. Transactions

- ✓ Originally single-document operations (including embedded documents/arrays) were atomic
- ✓ Single 4.0 version Mongo support multi-document transactions
- ✓ Since 4.2 version Mongo supports multi-document transactions on shared clusters (distributed transactions)
- ✓ Until a transaction commits, the data changes made in the transaction are not visible outside the transaction
- ✓ Operations that affect the database catalog, such as creating or dropping a collection or an index, are not allowed in transactions
- ✓ At any given time, you can have at most one open transaction for a session

Spring Data Mongo. Transactions


```
@Configuration
public class MongoDBConfiguration extends
            AbstractMongoClientConfiguration {

    @Bean
    public MongoTransactionManager transactionManager(
            MongoDatabaseFactory dbFactory) {
        return new MongoTransactionManager(dbFactory);
    }

    @Override
    protected String getDatabaseName() {
        return "shop";
    }
}
```

Manual configuration

Required to enable transactions

Explicitly specify database name

Spring Data Mongo. Transactions


```
@Service
@RequiredArgsConstructor
public class ProductService {

    private final MongoOperations mongoOperations;

    @Transactional
    public void save(Product product) {
        mongoOperations.save(product);

        mongoOperations.save(new Log("Product " + product.getId()
            + " saved"));
    }
```

com.mongodb.MongoClientException: Sessions are not supported by the MongoDB cluster to which this client is connected

Multi-document transactions are not supported for a non-replica set deployment

Spring Data Mongo. Transactions

com.mongodb.MongoWriteException: Cannot create namespace shop.products in multi-document transaction.

db.createCollection('products')
db.createCollection('logs')

Task 8. MongoTemplate and transactions

1. Review **BookService** class. Replace BookRepository field with MongoTemplate field. Update service implementation.
2. Run integration tests for BookService and make sure they work properly
3. Which operations should include multi-document transactions? Add @Transactional annotation to these operations

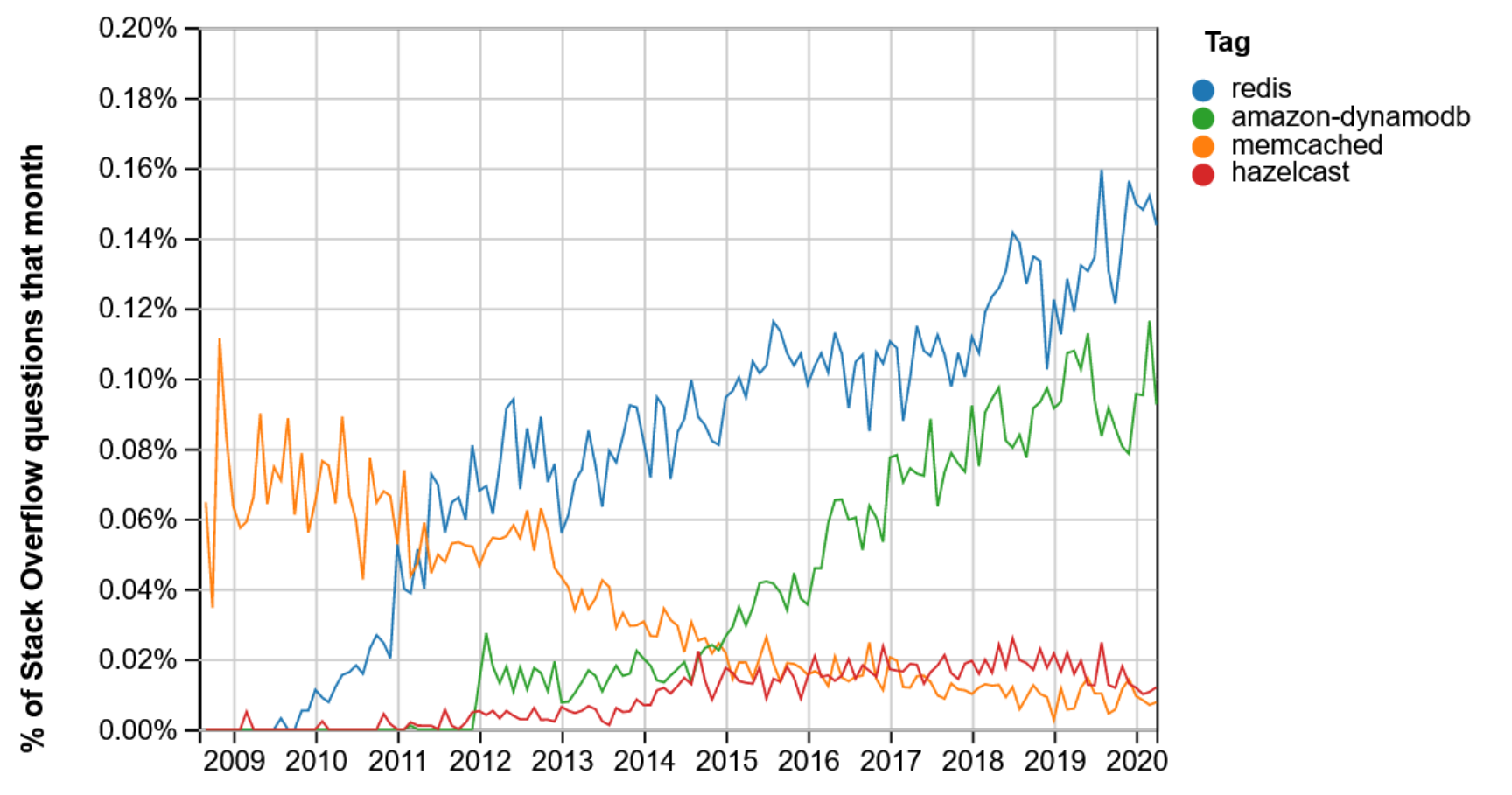
Key-value store

- Big dictionary/associated array
- Schemas can differ between rows
- Cloud: Amazon Simple DB, Azure Table Storage, Amazon DynamoDB
- MemcacheDB, Voldemort, Redis, Berkley DB, Hazelcast, Tarantool
- Good for storing metadata/cache data/temporary information
- Allows querying only by key (identifier)
- Value can be any type (XML, JSON, blob, text)
- Should be not used when different type of values have relationship

Tarantool

- Started in 2009
- Combines Lua application server and DBMS
- Can use Memtx engine (in-memory db) or Vinyl engine (disk storage)
- Can work with big data volumes because of LSM tree usage
- Supports Lua scripting and
- Supports sharding and replication
- Persistence with write-ahead-log file or snapshots

Key-value store



Sergiy Morenets, 2020

Redis

- Created in 2009 by Salvatore Sanfillipo as **Re**mote **D**isctionary **S**erver
- In-memory database, message broker and cache provider
- Super-lightweight and easy to use
- 6 data types, 180+ commands
- Supports monitoring and metrics
- Atomic, isolated and consistent
- Uses Lua scripting (since 2.6)
- Eviction algorithms are **LRU** (Last Recently Used) and **LFU** (Last frequently used)

redis

Redis. Popularity

- Competes with **Memcached**(multi-threaded vs single-threaded)
- Sometimes called "**Memcached on steroids**"
- No memory limit for 64-bit systems
- Supports up to **512M** for key/value size (Memcached supports 250 bytes)
- Used in Twitter, Github, Instagram, Flickr and Pinterest
- #1 key-value database
- #4 NoSQL database

redis

Redis. Persistence and RDB

- Data is stored in the memory for fast access
- RDB (Redis database file) option makes snapshots at specified intervals
- RDB binary format optimized for fast read/write and can use LZF compression
- All the Redis instance data can be stored in 1 RDB file
- Sufficient for backup/recovery purposes
- Can be synchronous or asynchronous

http://oldblog.antirez.com/post/redis-persistence-demystified.html

Redis. Persistence and AOF

- AOF (Append-only file) option stores every write operation (command) in the order they're received by server
- When Redis is started it reads/executes all commands from AOF
- AOF is text human-readable format
- Redis is able to rewrite AOF if it reaches maximum size
- You can combine RDB and AOF options or disable persistence
- AOF minimize chances of data loss but RDB gives best performance (if the keys are often updated)

Redis. High Availability

- **Redis Cluster** automatically split data across multiple servers (partitions)
- Requires Cluster bus for node-to-node communication channel
- **Redis Replication** creates a set of replicate nodes which are copies of master server
- Master periodically sends updates (client commands, keys expired) to the replicas
- Replica can connect to other replicas
- By default replication is asynchronous process

redis

Redis. High Availability

- **Redis Sentinel** is high-availability solution which supports monitoring (health check), automatic failover and configuration provider
- If master server fails then Sentinel promotes some replica server to the master role
- Client connects to Sentinel server to acquire master address
- At least three Sentinel servers (in different availability zones) are recommended

redis

Redis. Transactions

- All commands inside transaction are queued on the client and sent to the server when transaction commits
- It’s possible to roll back (discard) active transaction
- If some command produces failures then transactions is not roll backed
- Optimistic lock on selected keys with WATCH command

redis

Task 9. Redis basic commands

1. Run Redis 6 as Docker container
2. Run Redis CLI
3. Open new terminal and start another Redis console using *docker exec -it redis redis-cl monitor* command. It should print all the commands entered in any console.
4. Try to print all the configuration settings using *CONFIG GET ** command (or *CONFIG GET dir* for single setting).

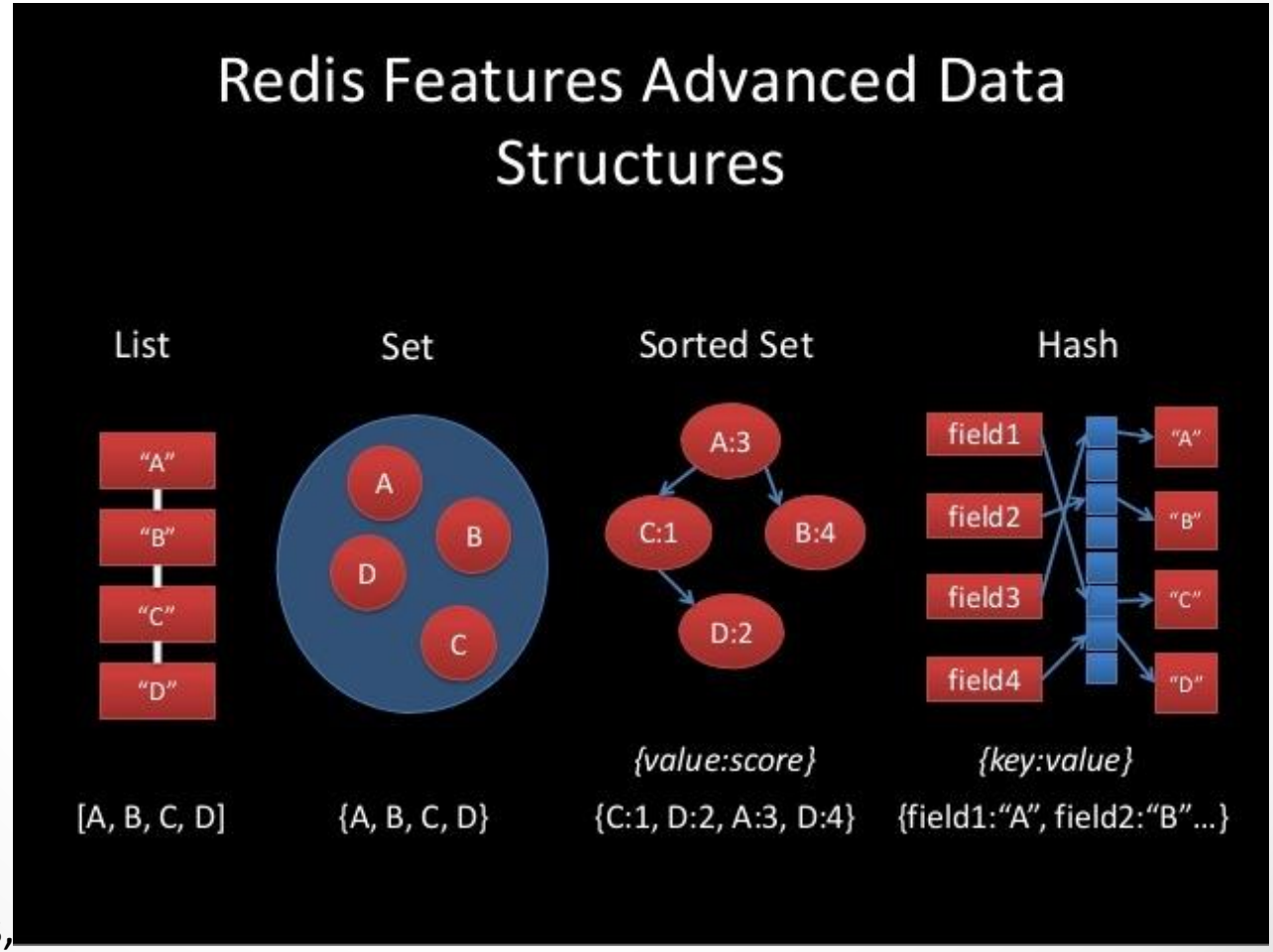
Redis data types

- **String** can be used as numeric value, text or bitmap (value can’t exceed 512M)
- Supports expiration and counting
- **Lists** can behave instead of collection, stack of queue and based on linked lists
- Maximum number of elements is 4 billions
- **Hash** data type allows to store objects (properties and values)
- Property name/value should be String
- **Set** is unordered collection of unique strings
- Maximum number of elements is 4 billions

Redis data types

- **Sorted set** contains strings where each elements is assigned a score
- If two elements have the same score then they’re sorted in alphabetical order
- Bitmap is bit array (sequence of bits) implemented as string
- Each bitmap can store up to 4 billions of bit

Redis data types



Task 10. Redis data types

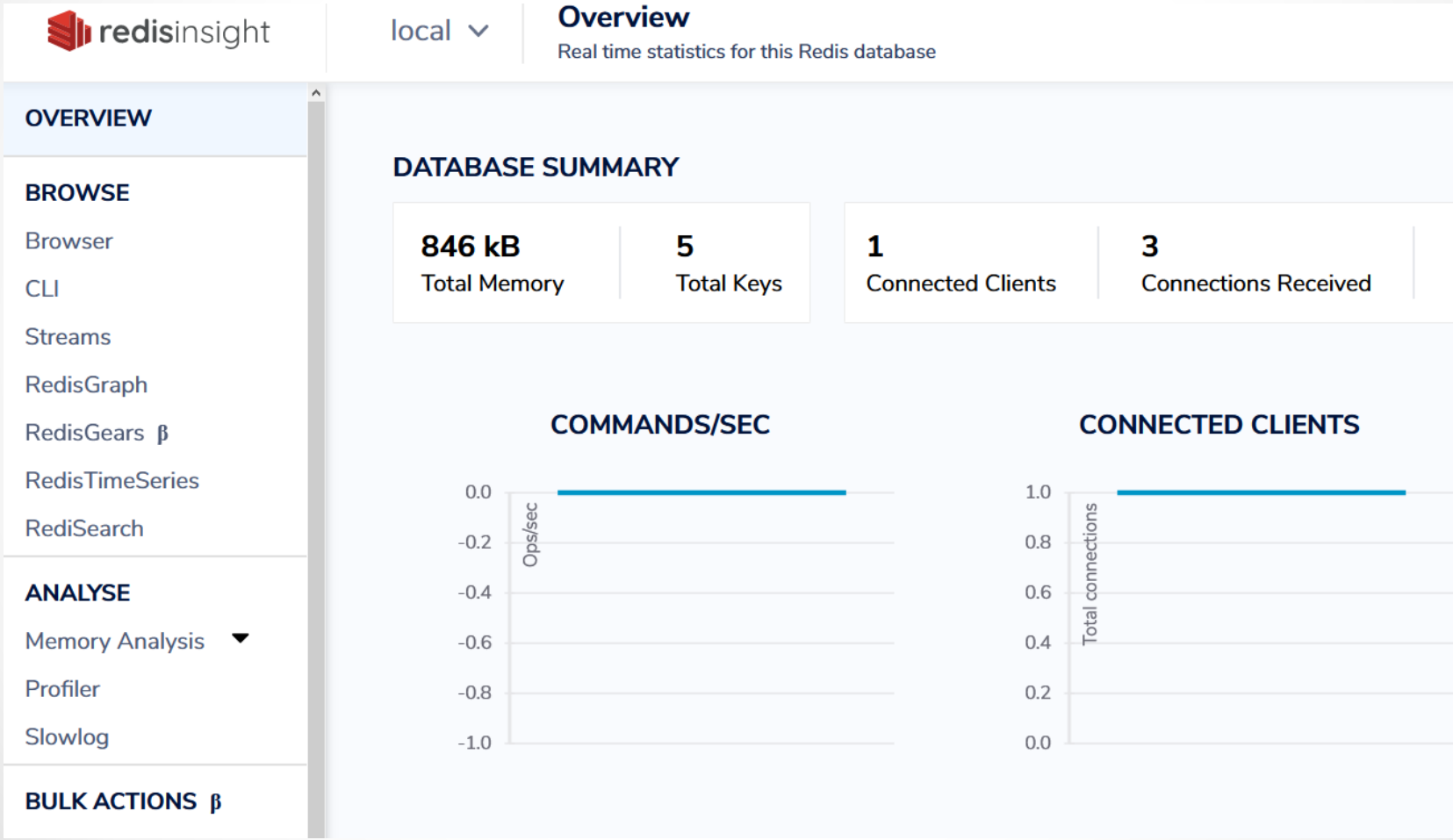
1. Create new hash key: *hmset book:1 id 1 title Redis* and try to view key value: *hgetall book:1* or single hash item: *hget book:1 id*
2. How can you find all the books where title is Redis?
3. Print all the keys using *SCAN 0* command. Print all keys using *KEYS ** command. You can try filtering keys using regular expression, for example, *KEYS *t** or *KEYS tex?*

Redis Insight

- ✓ Browser-based management interface
- ✓ Data inspection, health monitoring and runtime server configuration
- ✓ Cluster management
- ✓ Memory usage analysis

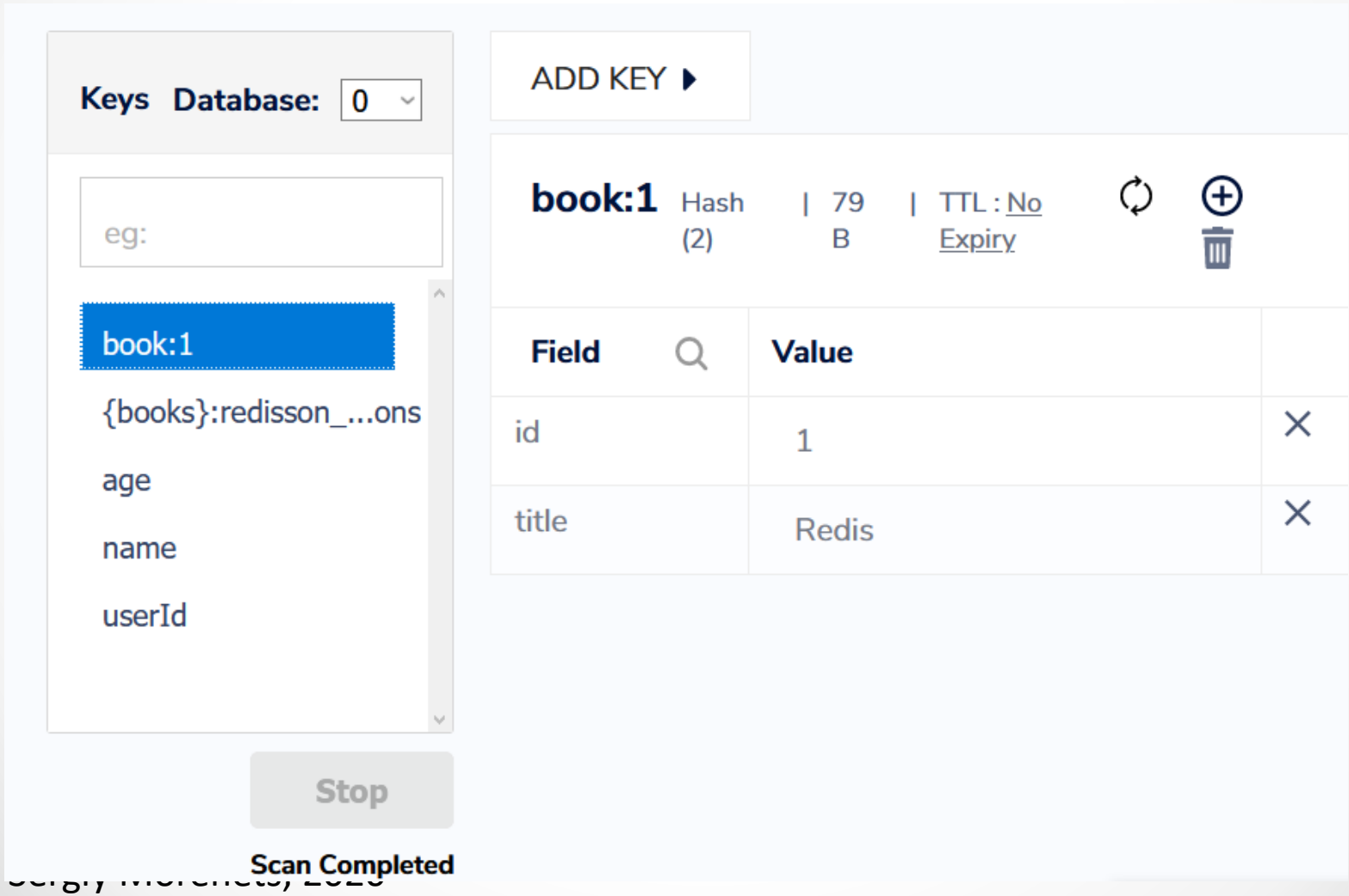
redis

Redis Insight

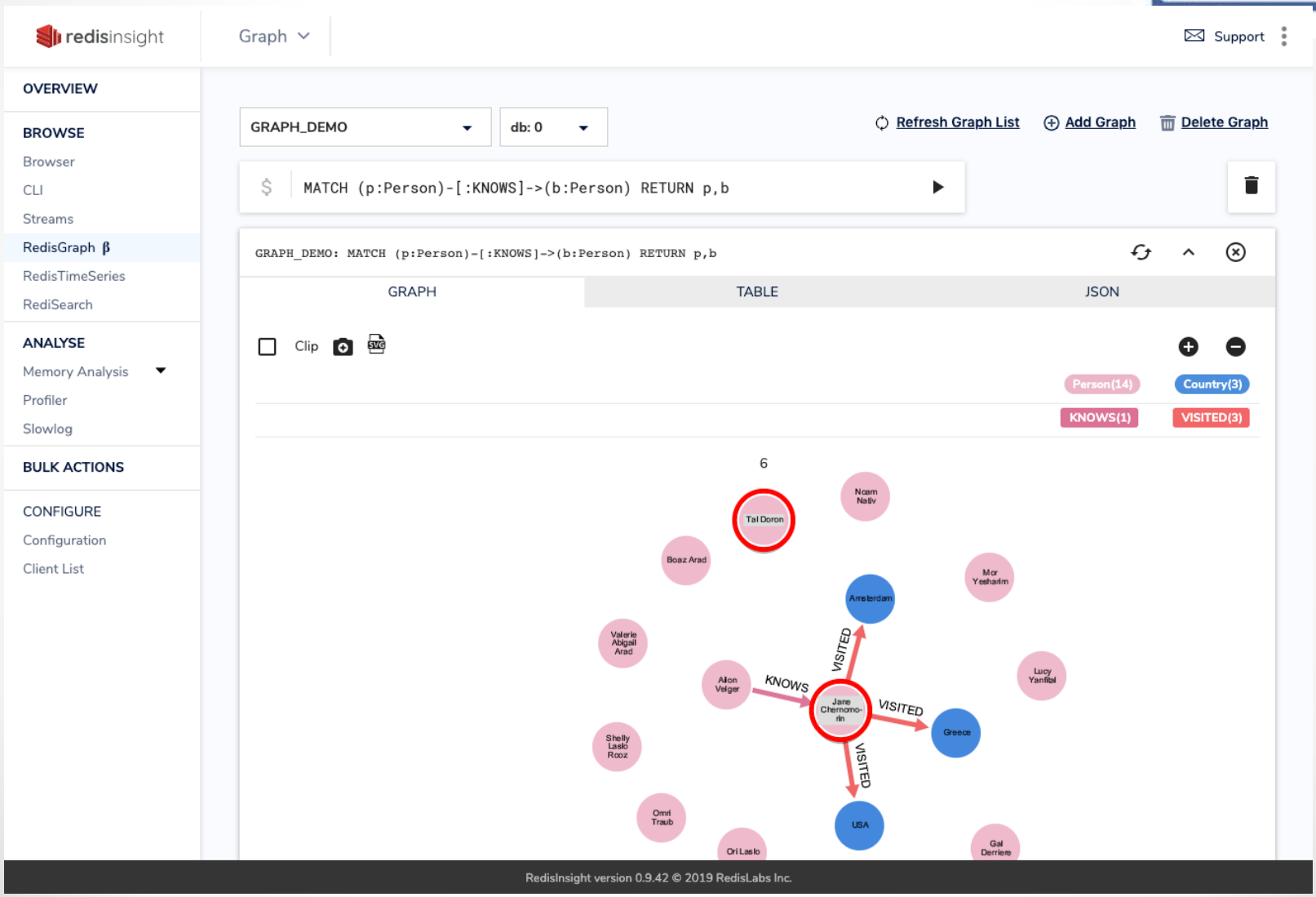


Sergiy Morenets, 2020

Redis Insight. Browser



Redis Insight. Graphs



Task 11. Redis Insight

1. Download and install Redis Insight
2. Start Redis Insight and open GUI
3. Click ‘Browser’ menu and review database content
4. Click ‘CLI’ and try to run CLI commands
5. Click ‘Memory Analysis’. What is server memory usage?
6. Click ‘Configuration’. Which persistence mode is currently used?

Spring Data Redis

- ✓ Jedis (lightweight implementation)
- ✓ Lettuce (default connector)
- ✓ Doesn't support building dynamic queries

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>${spring.boot.version}</version>
</dependency>
```

Lettuce

- Netty-based connectors
- Support sync/async/reactive approaches
- Supports Redis Standalone/Master-Slave/Cluster/Sentinel
- JDK8+
- Spring/CDI integration
- Used by Spring Data Redis
- Supports AWS ElasticCache and Azure Redis Cluster

Lettuce. Basic API


```
RedisClient redisClient = RedisClient.create(RedisURI
        .create("redis://localhost:6379/0"));
try (StatefulRedisConnection<String, String> connection
        = redisClient.connect()) {
    RedisCommands<String, String> cmd = connection.sync();

    cmd.set("userId", "1");
    String value = cmd.get("userId");
    cmd.incr("userId");
    cmd.set("users:1", userName);
}
redisClient.shutdown();
```

Always use namespace for multi-data

Spring Data Redis


```
@RedisHash("sessions")
public class UserSession {
    @Id
    private int userId;

    private String login;
```

```
public interface UserSessionRepository extends
        KeyValueRepository<UserSession, Integer> {
```

```
private final UserSessionRepository sessionRepository;

@PostMapping("{userId}")
public void login(@PathVariable int userId) {
    sessionRepository.save(new UserSession(userId));
}
```

Spring Data Redis. Queries

findOne → HGETALL “sessions:1”

save → DEL “sessions:1”
HMSET “sessions:1” <fields>
SADD “sessions” “1”

findAll → SMEMBERS “sessions”
HGETALL “sessions:1”

deleteById → DEL “sessions:1”
SREM “sessions” “1”

count → SCARD “sessions”

findByLogin → SINTER “sessions:login:1”

Spring Data Redis. Examples & probes


```
public interface UserSessionRepository extends
        KeyValueRepository<UserSession, Integer>,
        QueryByExampleExecutor<UserSession>{
```

```
UserSession probe = new UserSession();
probe.setLogin(login);

Example<UserSession> example = Example.of(probe);

Iterable<UserSession> sessions = sessionRepository
        .findAll(example);
sessions.forEach(session -> System.out.println(
        session.getUserId()));
```

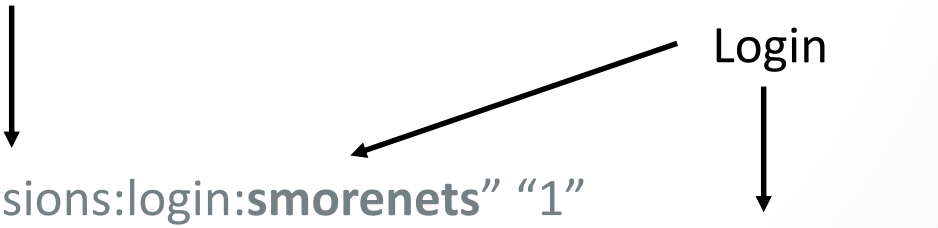
Only exact match

Prints all sessions

Spring Data Redis. Examples & probes

```
@RedisHash("sessions")
public class UserSession {
    @Id
    private Integer userId;

    @Indexed
    private String login;
```

```
Iterable<UserSession> sessions = sessionRepository
        .findAll(example);
```



Redis. Expiration


```
@RedisHash(value = "sessions", timeToLive = 15)
public class UserSession {
```

Expiration time in seconds

```
@RedisHash("sessions")
public class UserSession {
    @Id
    private int userId;

    @TimeToLive(unit = TimeUnit.SECONDS)
    private int expiration;
```

Redis Template


```
@RestController
@RequestMapping("users")
@RequiredArgsConstructor
public class AuthenticationController {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    @PostMapping("{userId}")
    public void login(@PathVariable String userId) {
        redisTemplate.opsForSet().add("users", userId);
    }
```

Add new element to set

Spring Data Redis. Persistence

- Redis can store objects as binary data (byte arrays)
- RedisSerializer is generic interface to marshal/unmarshal data
- By default Java (default) and Jackson serializers are supported
- Transactions are supported with RedisTemplate but not supported for Redis repositories

Task 12. Spring Data Redis

1. Add Spring Data Redis dependency
2. Create new entity **ShoppingCart** that will contain user information and purchased books. Add new ShoppingCartRepository interface that will extend CrudRepository interface
3. Assign expiration time using @RedisHash or @TimeToLive annotations

✓ Sergiy Morenets, sergey.morenets@gmail.com