Polyglot persistence for Java developers - moving out of the relational comfort zone

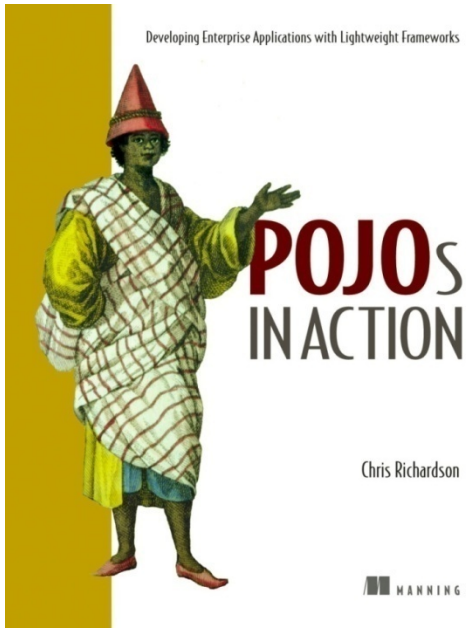
Chris Richardson

Author of POJOs in Action
Founder of CloudFoundry.com
Chris.Richardson@SpringSource.Com
@crichardson

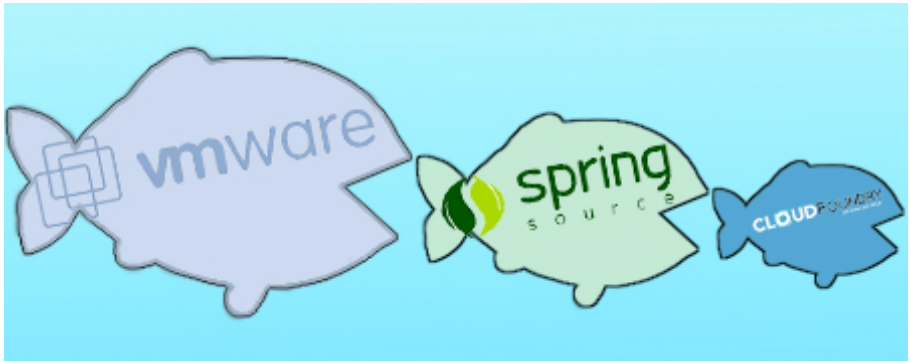
Overall presentation goal

The joy and pain of building Java applications that use NoSQL

About Chris



- Grew up in England and live in Oakland, CA
- Over 25+ years of software development experience including 14 years of Java
- Speaker at JavaOne, SpringOne, NFJS, JavaPolis, Spring Experience, etc.
- Organize the Oakland JUG and the Groovy Grails meetup



http://www.theregister.co.uk/2009/08/19/springsource_cloud_foundry/

Agenda

The trouble with relational databases

Overview of NoSQL databases

Introduction to Spring Data

NoSQL case study: POJOs in Action

Relational databases are great

SQL = Rich, declarative query language

Database enforces referential integrity

ACID semantics

Well understood by developers

Well supported by frameworks and tools, e.g. Spring JDBC, Hibernate, JPA

Well understood by operations

- Configuration
- Care and feeding
- Backups
- Tuning
- Failure and recovery
- Performance characteristics

But….

The trouble with relational databases

Object/relational impedance mismatch

- Complicated to map rich domain model to relational schema

Relational schema is rigid

- Difficult to handle semi-structured data, e.g. varying attributes
- Schema changes = downtime or $$

Extremely difficult/impossible to scale writes:

- Vertical scaling is limited/requires $$
- Horizontal scaling is limited or requires $$

Performance can be suboptimal for some use cases

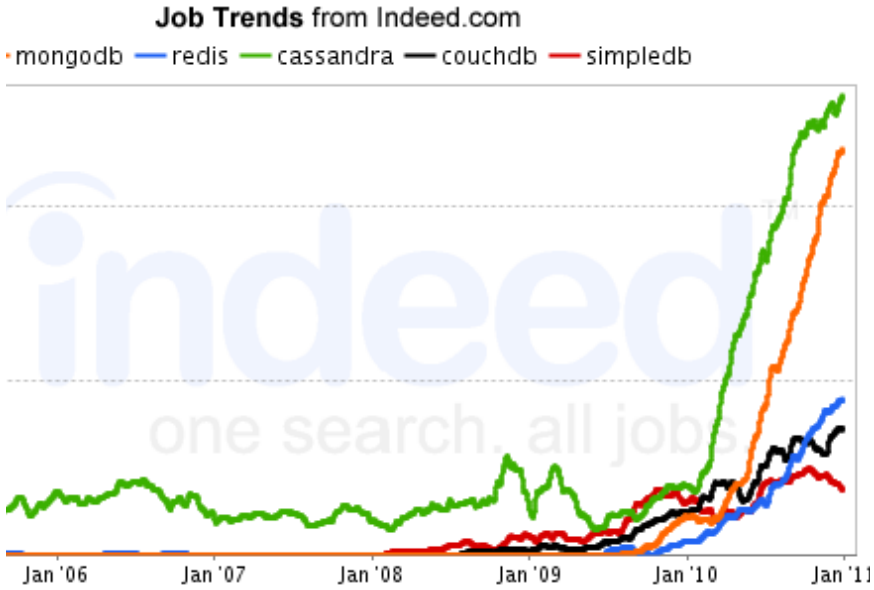
NoSQL databases have emerged…

Each one offers some combination of:

- High performance
- High scalability
- Rich data-model
- Schema less

In return for:

- Limited transactions
- Relaxed consistency
- …



… but there are few commonalities

Everyone and their dog has written one

Different data models

- Key-value
- Column
- Document
- Graph

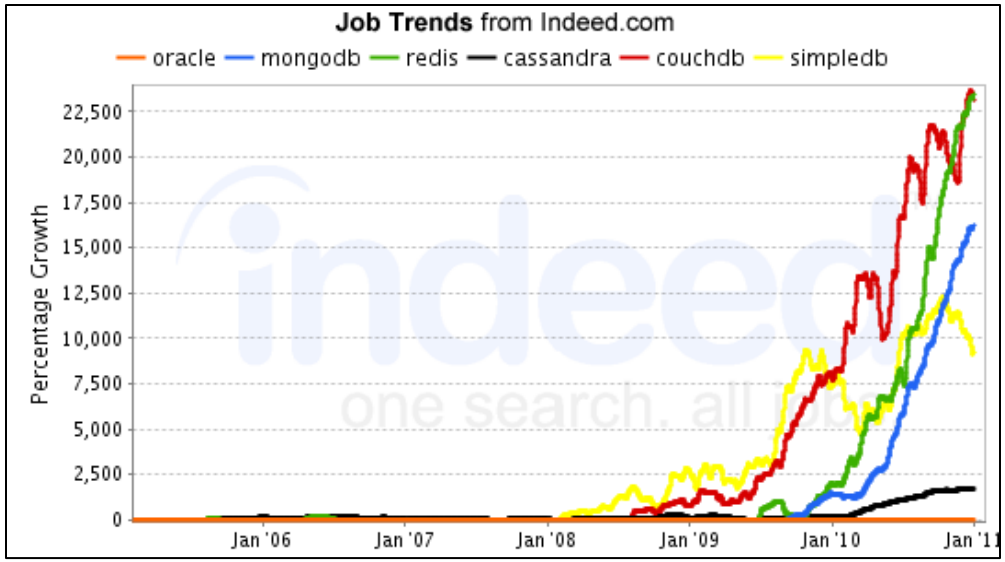
Different APIs – No JDBC, Hibernate, JPA (generally)

"Same sorry state as the database market in the 1970s before SQL was invented" http://queue.acm.org/detail.cfm?id=1961297

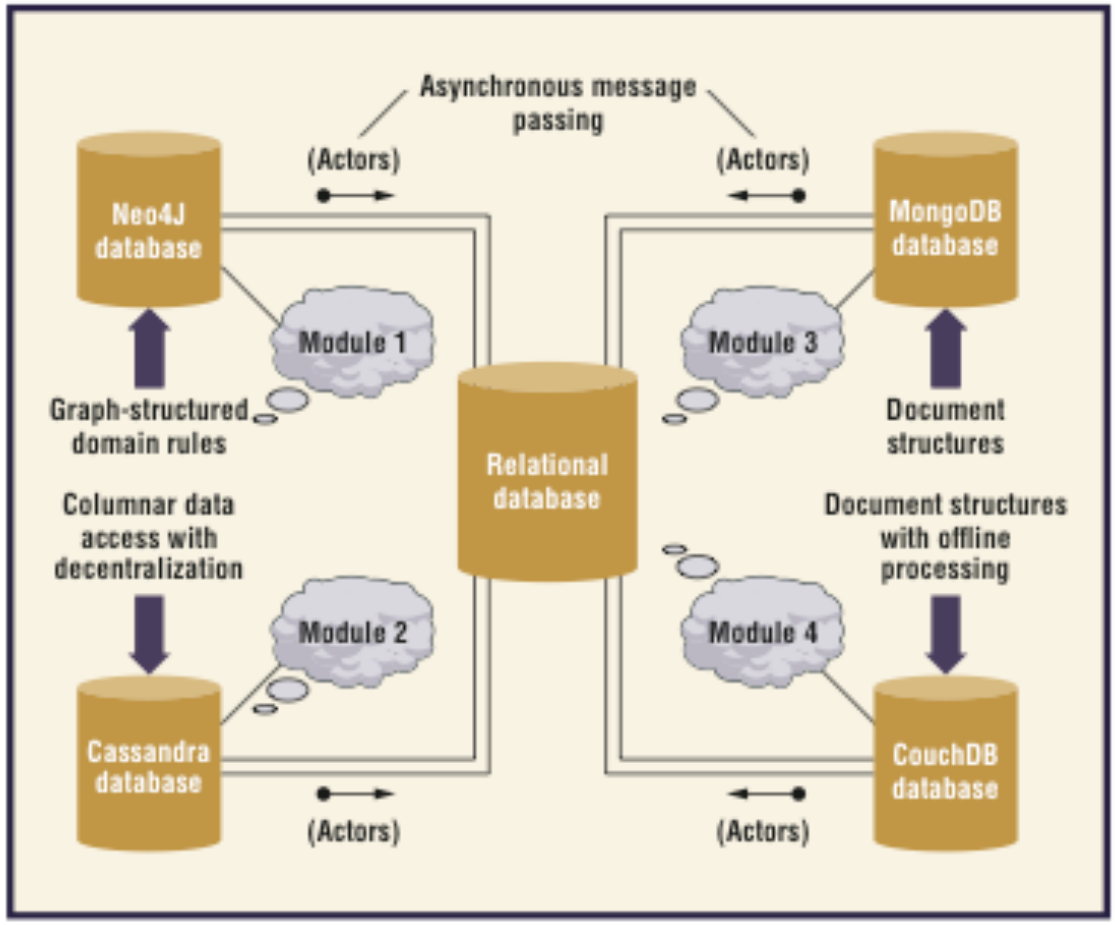
How to I access my data?

Reality Check - Relational DBs are not going away

- NoSQL usage small by comparison …
- But growing…



Future = multi-paradigm data storage for enterprise applications



IEEE Software Sept/October 2010 - Debasish Ghosh / Twitter @debasishg

Agenda

The trouble with relational databases

Overview of NoSQL databases

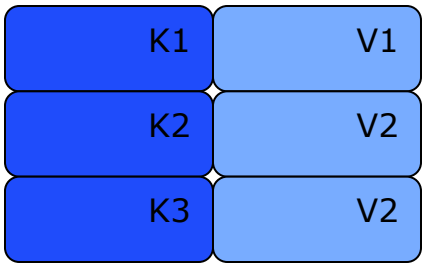
Introduction to Spring Data

NoSQL case study: POJOs in Action

Redis

Advanced key-value store

- Think memcached on steroids (the good kind)
- Values can be binary strings, Lists, Sets, Ordered Sets, Hash maps, ..
- Operations for each data type, e.g. appending to a list, adding to a set, retrieving a slice of a list, …



Very fast:

- In-memory operations
- ~100K operations/second on entry-level hardware

Persistent

- Periodic snapshots of memory OR append commands to log file
- Limits are size of keys retained in memory.

Has “transactions”

- Commands can be batched and executed atomically

Redis CLI

```
redis> sadd myset a
(integer) 1
redis> sadd myset b
(integer) 1
redis> smembers myset
1. "a"
2. "b"
redis> srem myset a
(integer) 1
redis> smembers myset
1. "b"
```

Scaling Redis

Master/slave replication

- Tree of Redis servers
- Non-persistent master can replicate to a persistent slave
- Use slaves for read-only queries

Sharding

- Client-side only – consistent hashing based on key
- Server-side sharding – coming one day

Run multiple servers per physical host

- Server is single threaded => Leverage multiple CPUs
- 32 bit more efficient than 64 bit

Optional "virtual memory"

- Ideally data should fit in RAM
- Values (not keys) written to disc

Redis use cases

Use in conjunction with another database as the SOR

Drop-in replacement for Memcached

- Session state
- Cache of data retrieved from SOR
- Denormalized datastore for high-performance queries

Hit counts using INCR command

Randomly selecting an item – SRANDMEMBER

Queuing – Lists with LPOP, RPUSH, ….

High score tables – Sorted sets

Notable users: github, guardian.co.uk, ….

Cassandra

An Apache open-source project originally developed by Facebook for inbox search

Extremely scalable

Fast writes = append to a log

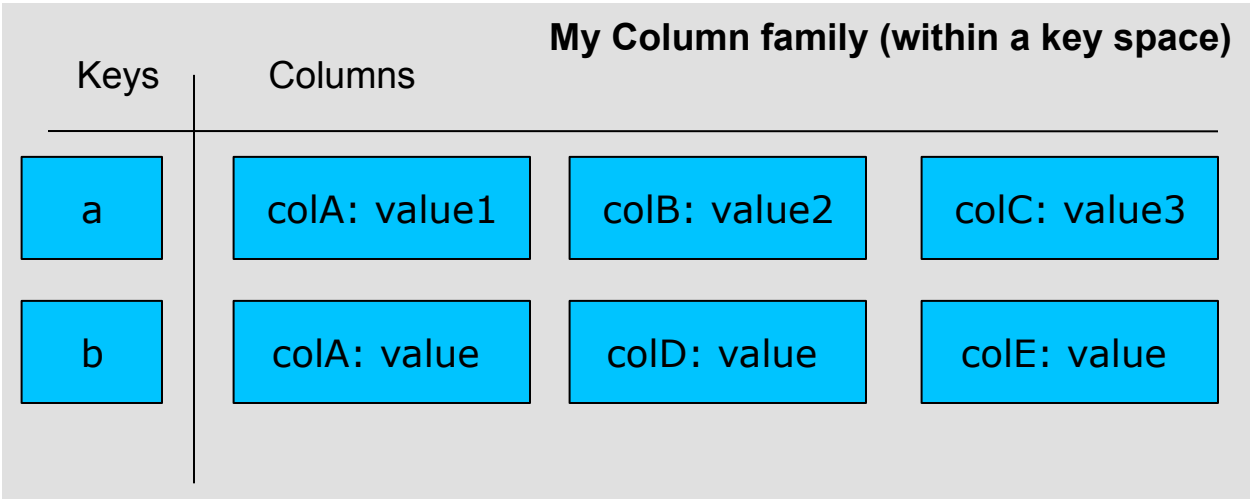
Data is replicated and sharded

Rack and datacenter aware

Column-oriented database

- The data model will hurt your brain
- 4 or 5-dimensional hash map

Cassandra data model



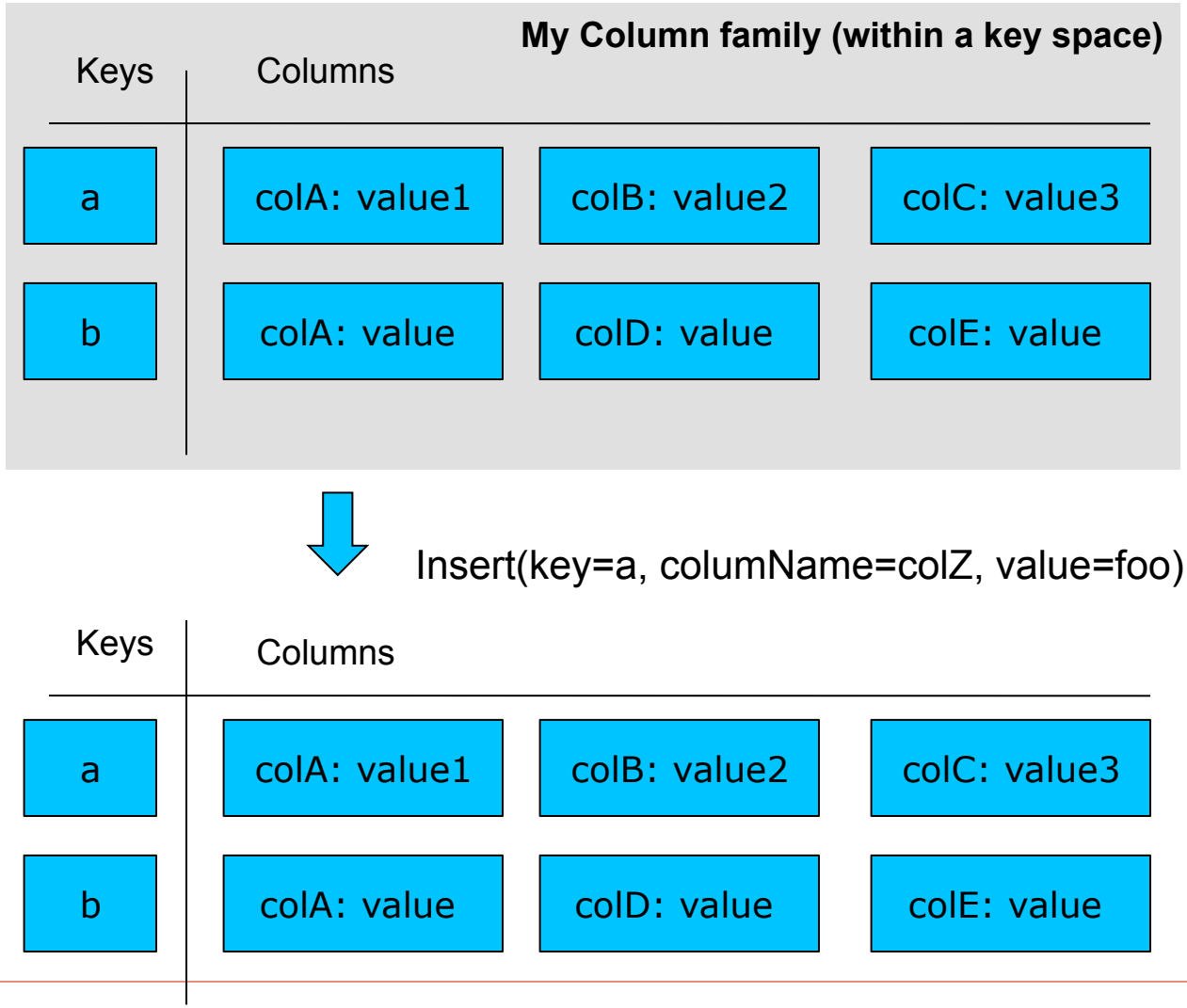
4-D map: keySpace x key x columnFamily x column ➔ value

Column names are dynamic; can contain data

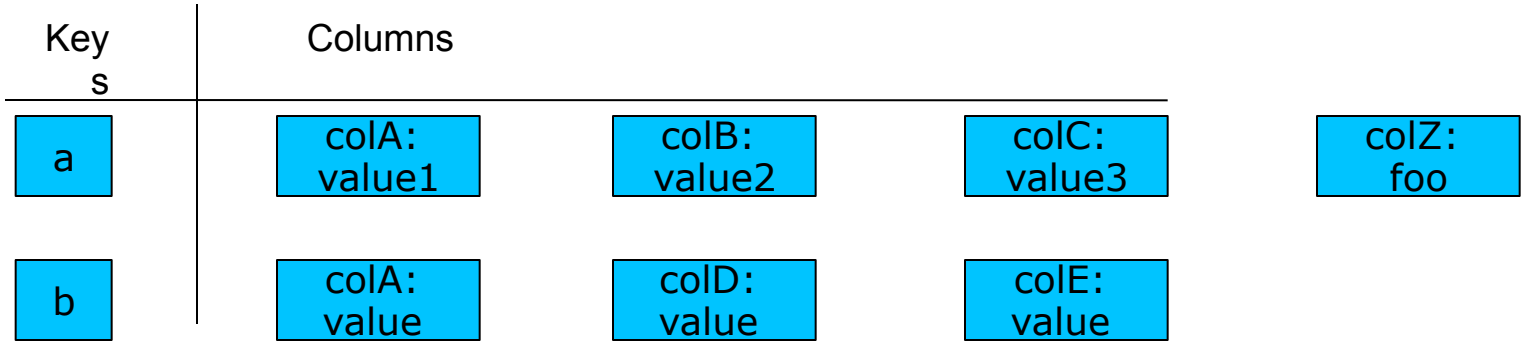
Arbitrary number of columns

One CF row = one DDD aggregate

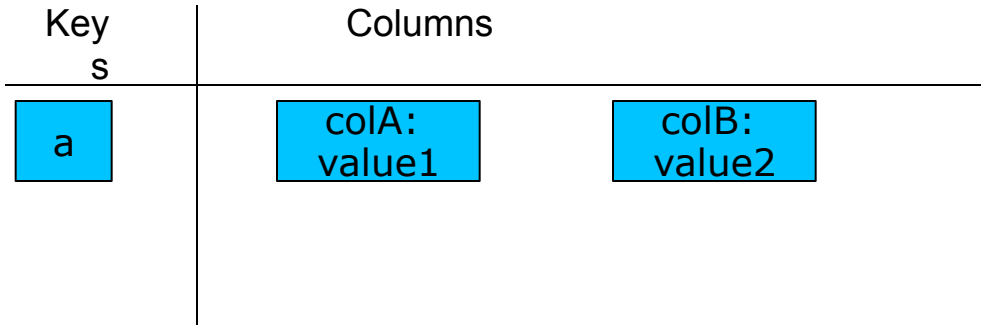
Cassandra data model – insert/update



Cassandra query example – slice

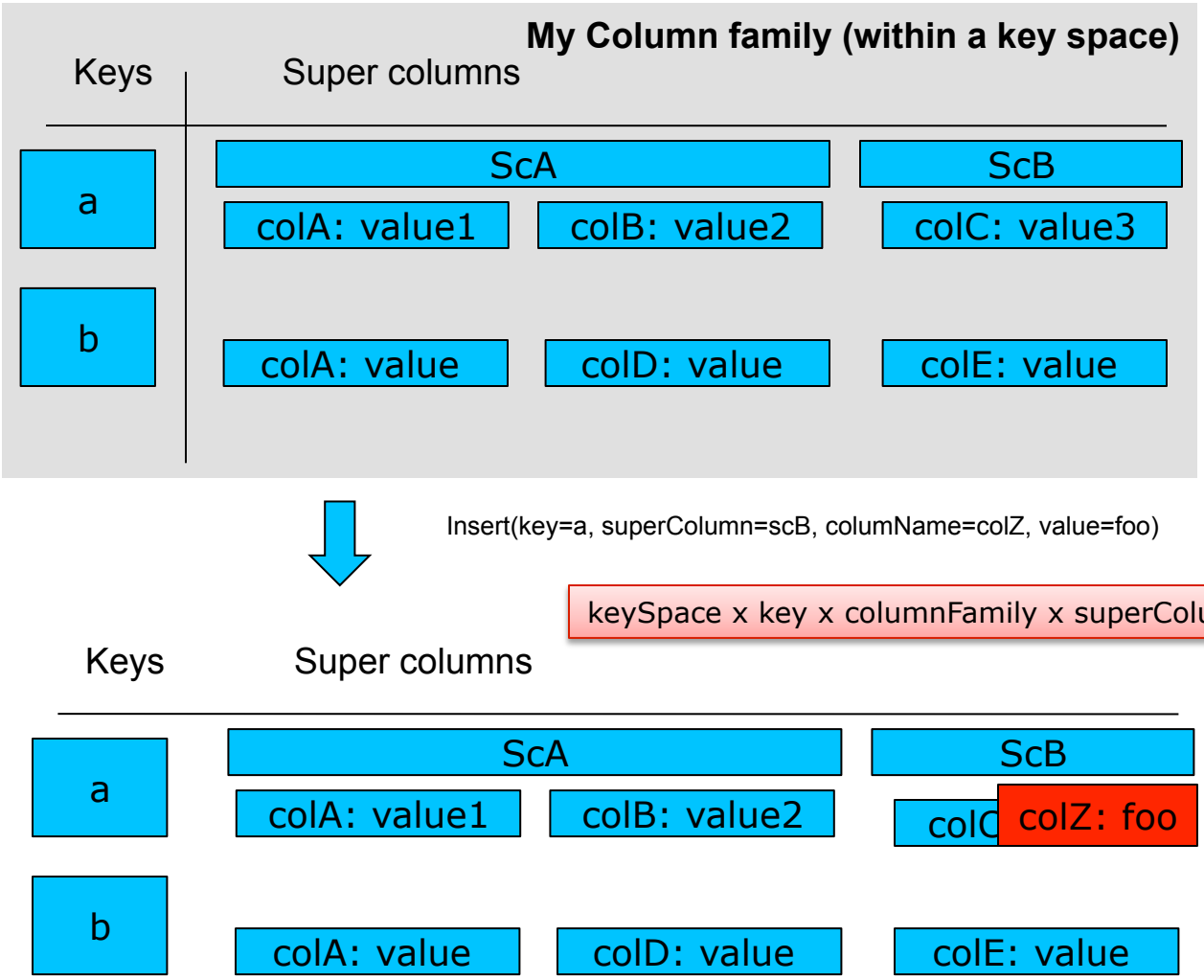


slice(key=a, startColumn=colA, endColumnName=colC)

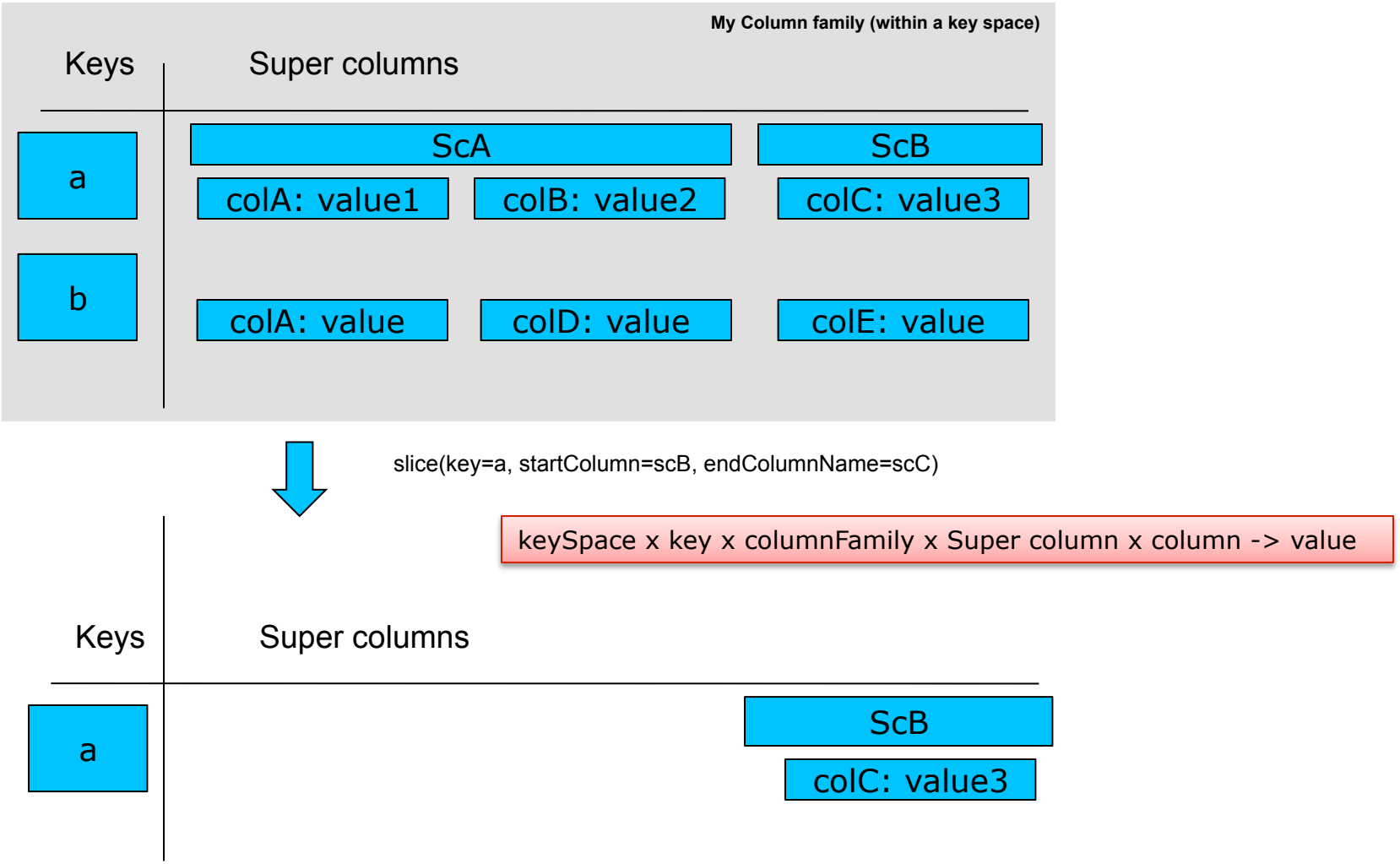


You can also do a rangeSlice which returns a range of keys – less efficient

Super Column Families – one more dimension



Getting data with super slice

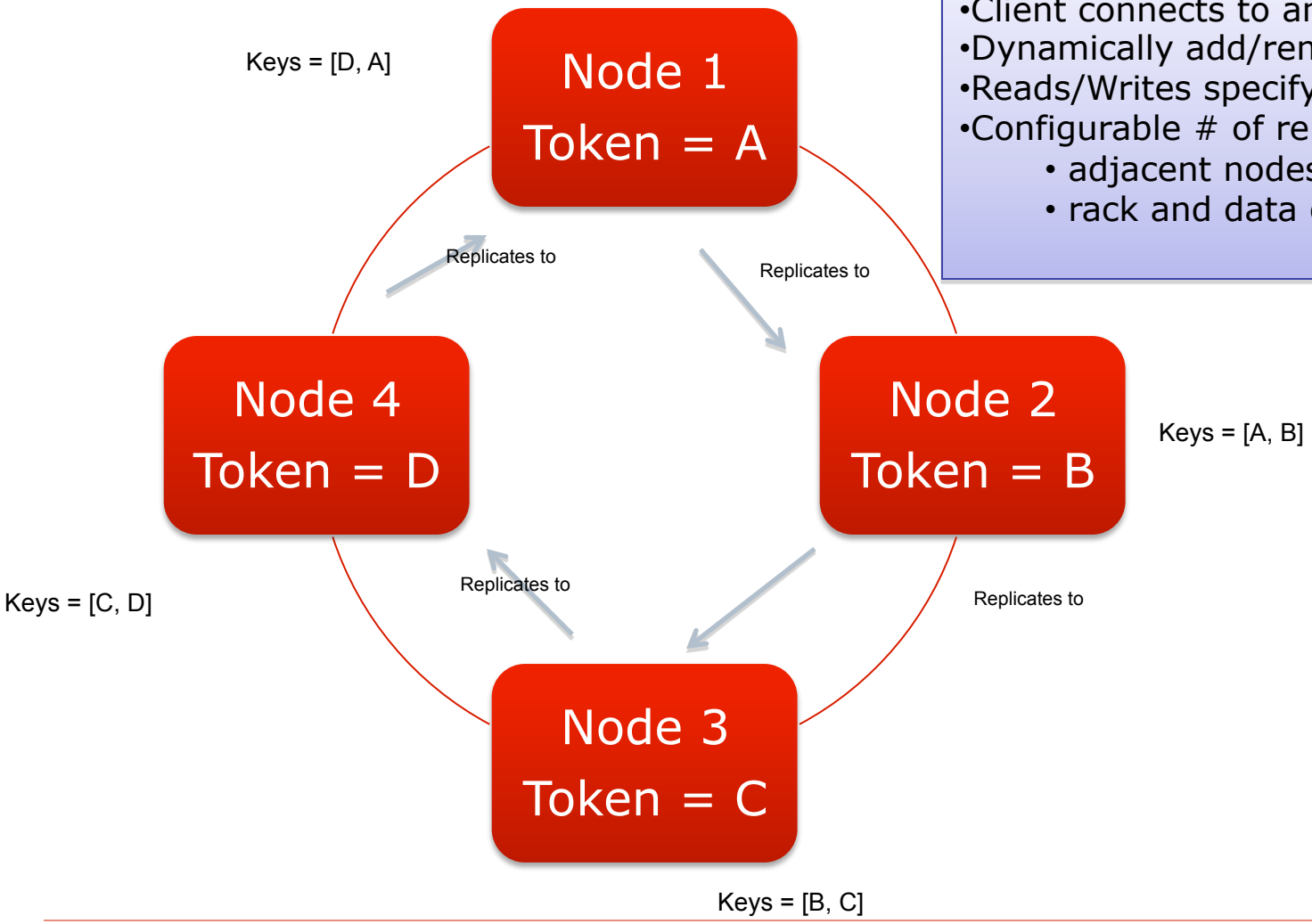


Cassandra CLI

```
$ bin/cassandra-cli -h localhost
Connected to: "Test Cluster" on localhost/9160
Welcome to cassandra CLI.
[default@unknown] use Keyspace1;
Authenticated to keyspace: Keyspace1
[default@Keyspace1] list restaurantDetails;
Using default limit of 100
-------------------
RowKey: 1
=> (super_column=attributes,
      (column=json, value={"id":
    1,"name":"Ajanta","menuItems"....

[default@Keyspace1] get restaurantDetails['1']['attributes'];
=> (column=json, value={"id":1,"name":"Ajanta","menuItems"....
```

Scaling Cassandra



- Client connects to any node
- Dynamically add/remove nodes
- Reads/Writes specify how many nodes
- Configurable # of replicas
  - adjacent nodes
  - rack and data center aware

Cassandra use cases

Use cases

- Big data
- Persistent cache
- (Write intensive) Logging

Who is using it

- Digg, Facebook, Twitter, Reddit, Rackspace
- Cloudkick, Cisco, SimpleGeo, Ooyala, OpenX
- “The largest production cluster has over 100 TB of data in over 150 machines.“ – Casssandra web site

MongoDB

Document-oriented database

- JSON-style documents: Lists, Maps, primitives
- Documents organized into collections (~table)

Full or partial document updates

- Transactional update in place on one document
- Atomic Modifiers

Rich query language for dynamic queries

Index support – secondary and compound

GridFS for efficiently storing large files

Map/Reduce

Data Model = Binary JSON documents

```
{
   "name" : "Ajanta",
   "type" : "Indian",
   "serviceArea" : [
      "94619",
      "94618"
   ],
   "openingHours" : [
      {
         "dayOfWeek" : Monday,
         "open" : 1730,
         "close" : 2130
      }
   ],
   "_id" : ObjectId("4bddc2f49d1505567c6220a0")
}
```

One document
=
one DDD aggregate

Sequence of bytes on disk = fast I/O

- No joins/seeks
- In-place updates when possible => no index updates

Transaction = update of single document

MongoDB CLI

```
$ bin/mongo
> use mydb
> r1 = {name: 'Ajanta'}
{name: 'Ajanta'}
> r2 = {name: 'Montclair Egg Shop'}
{name: 'Montclair Egg Shop'}
> db.restaurants.save(r1)
> r1
{ _id: ObjectId("98…"), name: "Ajanta"}
> db.restaurants.save(r2)
> r2
{ _id: ObjectId("66…"), name: "Montclair Egg Shop"}
> db.restaurants.find({name: /^A/})
{ _id: ObjectId("98…"), name: "Ajanta"}
> db.restaurants.update({name: "Ajanta"},
                        {name: "Ajanta Restaurant"})
```

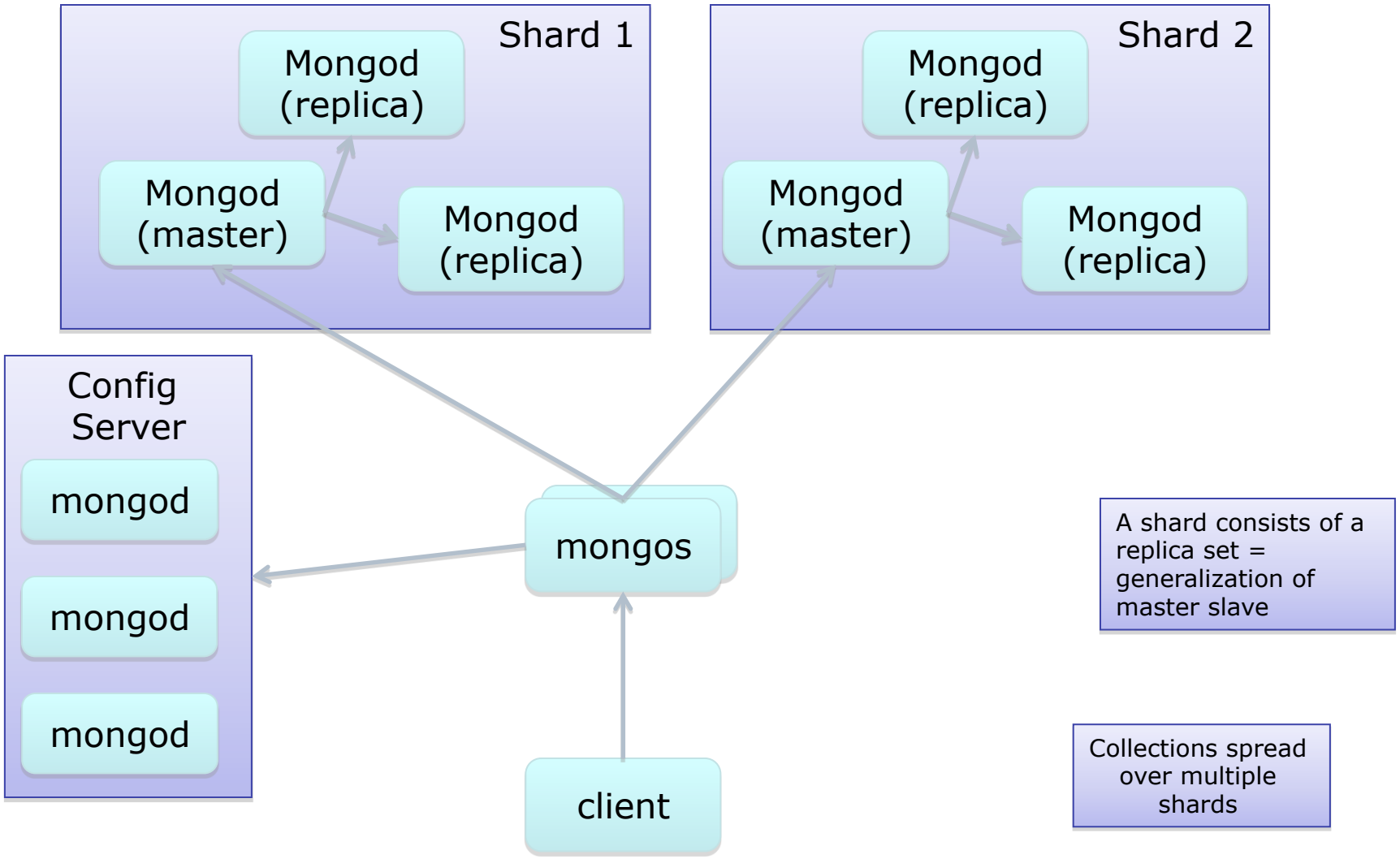
MongoDB query by example

Find a restaurant that serves the 94619 zip code and is open at 6pm on a Monday

```
{
  serviceArea:"94619",
  openingHours: {
    $elemMatch :  {
          "dayOfWeek" : "Monday",
          "open": {$lte: 1800},
          "close": {$gte: 1800}
     }
  }
}
```

```
DBCursor cursor = collection.find(qbeObject);
while (cursor.hasNext()) {
    DBObject o = cursor.next();
    …
 }
```

Scaling MongoDB



MongoDB use cases

Use cases

- Real-time analytics
- Content management systems
- Single document partial update
- Caching
- High volume writes

Who is using it?

- Shutterfly, Foursquare
- Bit.ly Intuit
- SourceForge, NY Times
- GILT Groupe, Evite,
- SugarCRM

Other NoSQL databases

SimpleDB – "key-value"

Neo4J – graph database

CouchDB – document-oriented

Membase – key-value

Riak – key-value + links

Hbase – column-oriented

…

http://nosql-database.org/ has a list of 122 NoSQL databases

Agenda

The trouble with relational databases

Overview of NoSQL databases

Introduction to Spring Data

NoSQL case study: POJOs in Action

NoSQL Java APIs

Database	Libraries
Redis	**Jedis**, JRedis, JDBC-Redis, RJC
Cassandra	Raw Thrift if you are a masochist **Hector**, …
MongoDB	MongoDB provides a Java driver

Spring Data Project Goals

Bring classic Spring value propositions to a wide range of NoSQL databases:

- Productivity
- Programming model consistency: E.g. <NoSQL>Template classes
- “Portability”

Many entry points to use

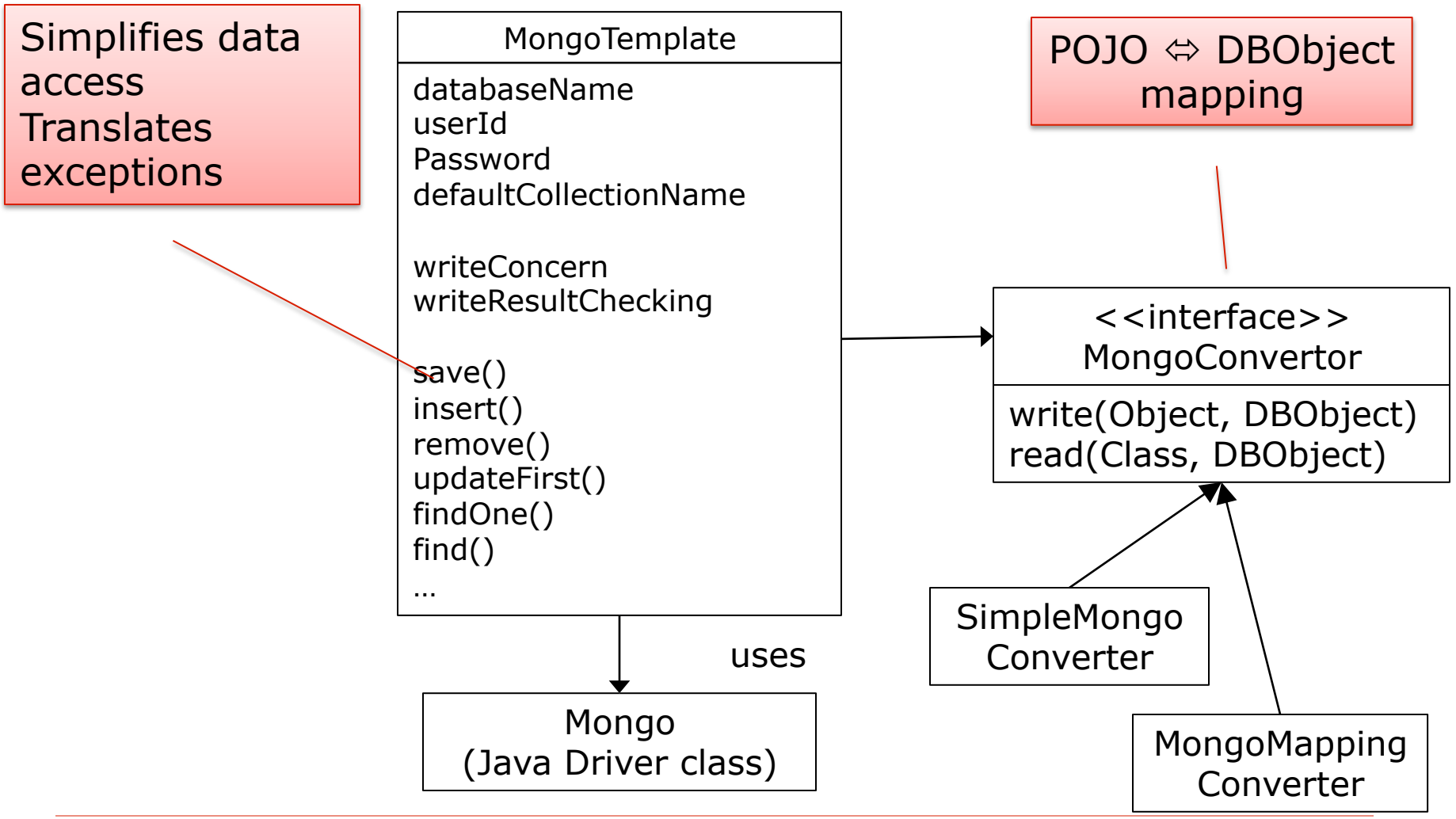
- Auto-generated repository implementations
- Opinionated APIs (Think JdbcTemplate)
- Object Mapping (Java and GORM)
- Cross Store Persistence Programming model
- Productivity support in Roo and Grails

Spring Data sub-projects

- Commons: Polyglot persistence
- Key-Value: Redis, Riak
- Document: MongoDB, CouchDB
- Graph: Neo4j
- GORM for NoSQL
- Various milestone releases
  - Key Value 1.0.0.M3 (Apr 6, 2011)
  - Document 1.0.0.M2 (April 9, 2011)
  - Graph - Neo4j Support 1.0.0 (April 19, 2011)
  - …

http://www.springsource.org/spring-data

MongoTemplate



Richer mapping

```
@Document
public class Person {

  @Id
  private ObjectId id;
  private String firstname;

  @Indexed
  private String lastname;

  @PersistenceConstructor
  public Person(String firstname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

....
}
```

Annotations define mapping: @Document, @Id, @Indexed, @PersistanceConstructor, @CompoundIndex, @DBRef, @GeoSpatialIndexed, @Value

Map fields instead of properties
➔ no getters or setters required

Non-default constructor

Index generation

Generic Mongo Repositories

```
interface PersonRepository extends MongoRepository<Person, ObjectId> {
    List<Person> findByLastname(String lastName);
}
```

```
<bean>
  <mongo:repositories
    base-package="net.chrisrichardson.mongodb.example.mongorepository"
        mongo-template-ref="mongoTemplate" />
</beans>
```

```
Person p = new Person("John", "Doe");
personRepository.save(p);

Person p2 = personRepository.findOne(p.getId());

 List<Person> johnDoes = personRepository.findByLastname("Doe");
 assertEquals(1, johnDoes.size());
```

Support for the QueryDSL project

Generated from domain model class

Type-safe composable queries

```
QPerson person = QPerson.person;

Predicate predicate =
        person.homeAddress.street1.eq("1 High Street")
                .and(person.firstname.eq("John"))

List<Person> people = personRepository.findAll(predicate);

assertEquals(1, people.size());
assertPersonEquals(p, people.get(0));
```

Cross-store/polyglot persistence

```
@Entity
public class Person {
   // In Database
  @Id private Long id;
  private String firstname;
  private String lastname;

 // In MongoDB
 @RelatedDocument  private Address address;
```

```
Person person = new Person(…);
entityManager.persist(person);

Person p2 = entityManager.find(…)
```

```
{ "_id" : ObjectId(”…..”),
 "_entity_id" : NumberLong(1),
  "_entity_class" : "net.. Person",
"_entity_field_name" : "address",
   "zip" : "94611", "street1" : "1 High Street", …}
```

Agenda

The trouble with relational databases

Overview of NoSQL databases

Introduction to Spring Data

NoSQL case study: POJOs in Action

Food to Go

Customer enters delivery address and delivery time

System displays available restaurants

- = restaurants that serve the zip code of the delivery address AND are open at the delivery time

```
class Restaurant {
  long id;
  String name;
  Set<String> serviceArea;
  Set<TimeRange> openingHours;
  List<MenuItem> menuItems;
}
```

```
class TimeRange {
  long id;
  int dayOfWeek;
  int openingTime;
  int closingTime;
}
```

```
class MenuItem {
  String name;
  double price;
}
```

Database schema

ID	Name	...
1	Ajanta	
2	Montclair Eggshop	

RESTAURANT table

Restaurant_id	zipcode
1	94707
1	94619
2	94611
2	94619

RESTAURANT_ZIPCODE table

RESTAURANT_TIME_RANGE table

Restaurant_id	dayOfWeek	openTime	closeTime
1	Monday	1130	1430
1	Monday	1730	2130
2	Tuesday	1130	…

SQL for finding available restaurants

Straightforward three-way join

```
select r.*
from restaurant r
 inner join restaurant_time_range tr
   on r.id =tr.restaurant_id
 inner join restaurant_zipcode sa
   on r.id = sa.restaurant_id
where “94619’ = sa.zip_code
and tr.day_of_week=“monday”
and tr.openingtime <= 1930
and 1930 <=tr.closingtime
```

Redis - Persisting restaurants is "easy"

rest:1:details	[name: "Ajanta", …]
rest:1:serviceArea	["94619", "94611", …]
rest:1:openingHours	[10, 11]
timerange:10	["dayOfWeek": "Monday", ..]
timerange:11	["dayOfWeek": "Tuesday", ..]

OR

rest:1	[name: "Ajanta", "serviceArea:0" : "94611", "serviceArea:1" : "94619", "menuItem:0:name", "Chicken Vindaloo", …]

OR

rest:1	{ .. A BIG STRING/BYTE ARRAY, E.G. JSON }

BUT…

… we can only retrieve them via primary key

➔ Queries instead of data model drives NoSQL database design

➔ We need to implement indexes

But how can a key-value store support a query that has

- A 3-way join
- Multiple =
- > and <

?

Denormalization eliminates joins

Restaurant_id	Day_of_week	Open_time	Close_time	Zip_code
1	Monday	1130	1430	94707
1	Monday	1130	1430	94619
1	Monday	1730	2130	94707
1	Monday	1730	2130	94619
2	Monday	0700	1430	94619
…				

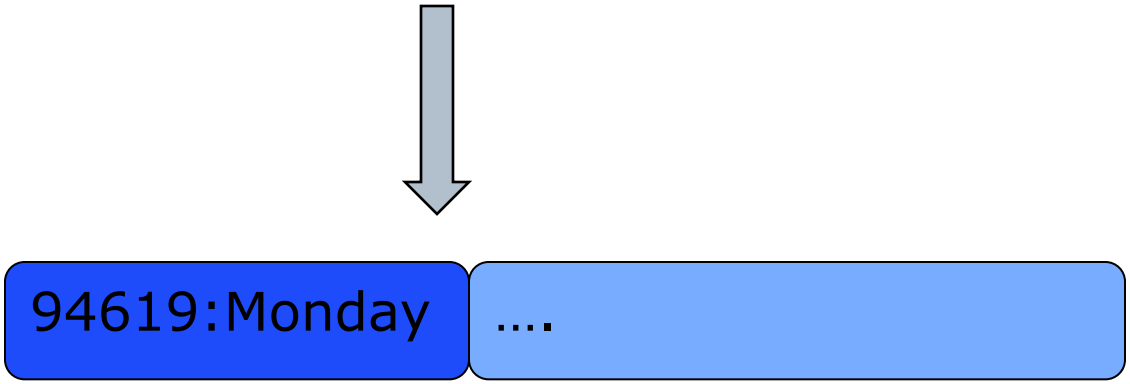
```
SELECT restaurant_id, open_time
 FROM time_range_zip_code
 WHERE day_of_week = 'Monday'
     AND zip_code = 94619
     AND 1815 < close_time
     AND open_time < 1815
```

One simple query
No joins
Two = and one <

Application filters out opening times after delivery time

Eliminate multiple =’s with concatenation

```
SELECT restaurant_id, open_time
 FROM time_range_zip_code
 WHERE day_of_week = ‘Monday’
    AND zip_code = 94619
    AND 1815 < close_time
```



GET 94619:Monday

Sorted sets support range queries

Key	Sorted Set [Entry:Score, …]
closingTimes:94707:Monday	[1130_1:1430, 1730_1:2130]
closingTimes:94619:Monday	[0700_2:1430, 1130_1:1430, 1730_2:2130]

```
ZRANGEBYSCORE
 closingTimes:94619:Monday
 1815 2359
->
{1730_1:2130}
```

Member: OpeningTime_RestaurantId
Score: ClosingTime

1730 is before 1815 => Ajanta is open

Querying my data

RedisTemplate-based code

```
@Repository
public class AvailableRestaurantRepositoryRedisImpl implements AvailableRestaurantRepository {

 @Autowired private final StringRedisTemplate redisTemplate;

 private BoundZSetOperations<String, String> closingTimes(int dayOfWeek, String zipCode) {
   return redisTemplate.boundZSetOps(AvailableRestaurantKeys.closingTimesKey(dayOfWeek, zipCode));
  }

 public List<AvailableRestaurant> findAvailableRestaurants(Address deliveryAddress, Date deliveryTime) {
   String zipCode = deliveryAddress.getZip();
   int timeOfDay = timeOfDay(deliveryTime);
   int dayOfWeek = dayOfWeek(deliveryTime);

   Set<String> closingTrs = closingTimes(dayOfWeek, zipCode).rangeByScore(timeOfDay, 2359);
   Set<String> restaurantIds = new HashSet<String>();
   String paddedTimeOfDay = FormattingUtil.format4(timeOfDay);
   for (String trId : closingTrs) {
     if (trId.substring(0, 4).compareTo(paddedTimeOfDay) <= 0)
       restaurantIds.add(StringUtils.substringAfterLast(trId, "_"));
   }

  Collection<String> jsonForRestaurants = redisTemplate.opsForValue().multiGet(
                                              AvailableRestaurantKeys.timeRangeRestaurantInfoKeys(restaurantIds ));
   List<AvailableRestaurant> restaurants = new ArrayList<AvailableRestaurant>();
   for (String json : jsonForRestaurants) {
     restaurants.add(AvailableRestaurant.fromJson(json));
   }
   return restaurants;
 }
```

Redis – Spring configuration

```
@Configuration
public class RedisConfiguration extends AbstractDatabaseConfig {

  @Bean
  public RedisConnectionFactory jedisConnectionFactory() {
    JedisConnectionFactory factory = new JedisConnectionFactory();
    factory.setHostName(databaseHostName);
    factory.setPort(6379);
    factory.setUsePool(true);
    JedisPoolConfig poolConfig = new JedisPoolConfig();
    poolConfig.setMaxActive(1000);
    factory.setPoolConfig(poolConfig);
    return factory;
  }

  @Bean
  public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory factory) {
    StringRedisTemplate template = new StringRedisTemplate();
    template.setConnectionFactory(factory);
    return template;
  }
}
```

Deleting/Updating a restaurant

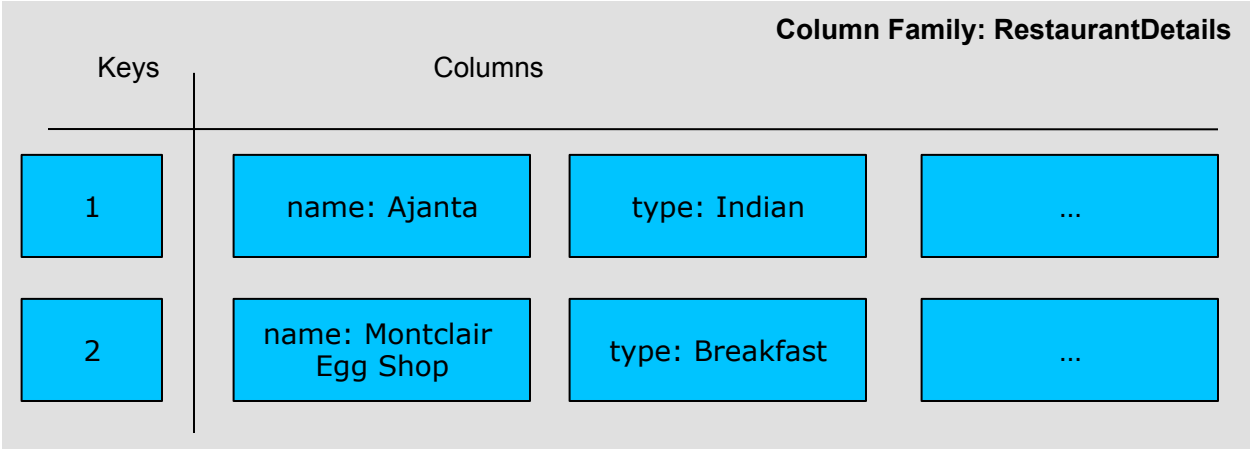
Need to delete members of the sorted sets

But we can't "find by a foreign key"

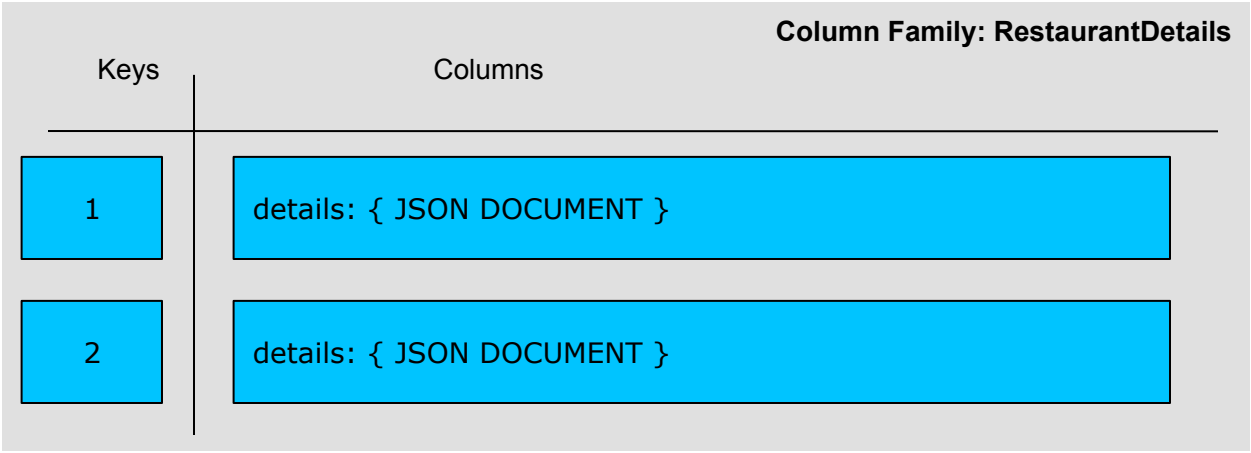
To delete a restaurant:

- GET JSON details of Restaurant (incl. openingHours + serviceArea)
- Re-compute sorted set keys and members and delete them
- Delete the JSON

Cassandra: Easy to store restaurants



OR



But we can't query this

Similar challenges to using Redis

No joins ➔ denormalize

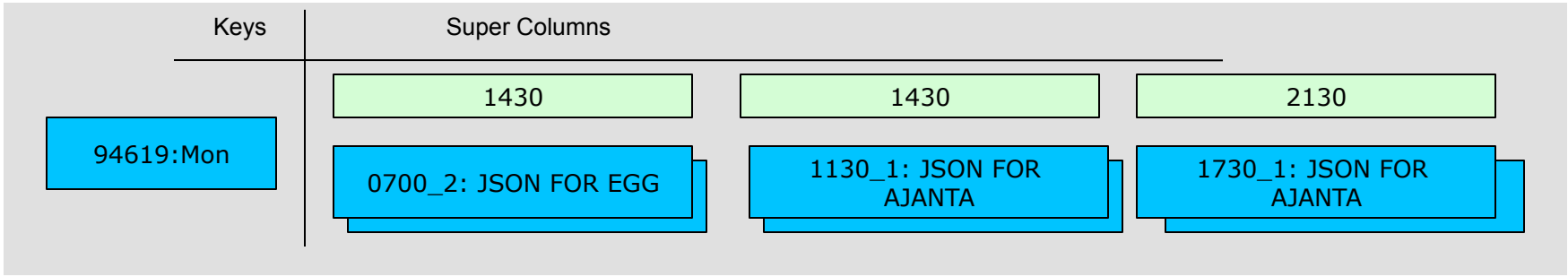
Can use composite/concatenated keys

- Prefix - equality match
- Suffix - can be range scan

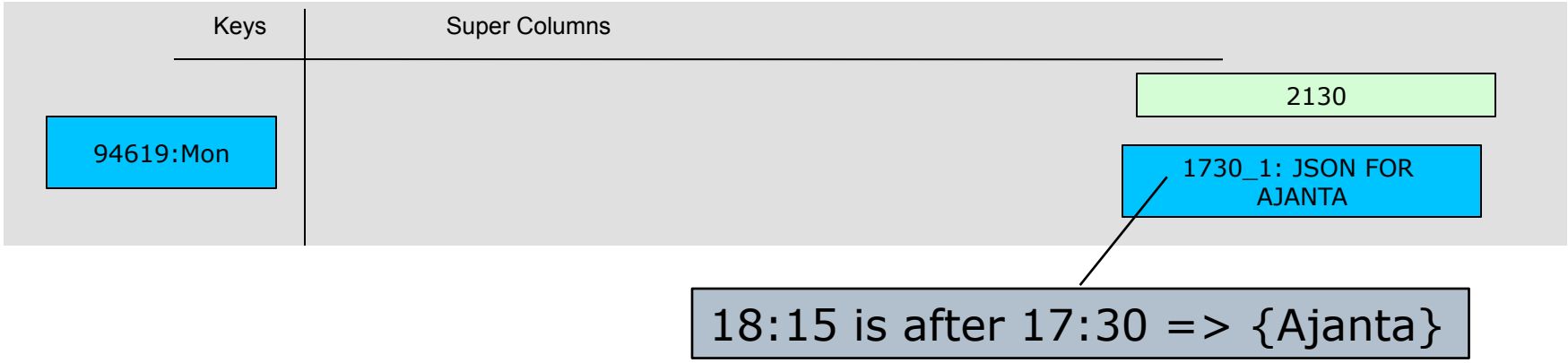
Some limited querying options

- Row key – exact or range
- Column name – exact or range

Cassandra: Find restaurants that close after the delivery time and then filter



SuperSlice
key= 94619:Mon
SliceStart = 1815
SliceEnd = 2359



Cassandra/Hector code

```
import me.prettyprint.hector.api.Cluster;

public class CassandraHelper {
   @Autowired  private final Cluster cluster;

  public <T> List<T> getSuperSlice(String keyspace, String columnFamily,
                                          String key, String sliceStart, String sliceEnd,
                                          SuperSliceResultMapper<T> resultMapper) {

    SuperSliceQuery<String, String, String, String> q =
      HFactory.createSuperSliceQuery(HFactory.createKeyspace(keyspace, cluster),
            StringSerializer.get(), StringSerializer.get(), StringSerializer.get(), StringSerializer.get());
    q.setColumnFamily(columnFamily);
    q.setKey(key);
    q.setRange(sliceStart, sliceEnd, false, 10000);

    QueryResult<SuperSlice<String, String, String>> qr = q.execute();

    SuperColumnRowProcessor<T> rowProcessor = new SuperColumnRowProcessor<T>(resultMapper);

    for (HSuperColumn<String, String, String> superColumn : qr.get().getSuperColumns()) {
      List<HColumn<String, String>> columns = superColumn.getColumns();
      rowProcessor.processRow(key, superColumn.getName(), columns);
    }
    return rowProcessor.getResult();
  }
}
```

MongoDB = easy to store

```
{
   "_id": "1234"
   "name": "Ajanta",
   "serviceArea": ["94619", "99999"],
   "openingHours": [
       {
          "dayOfWeek": 1,
          "open": 1130,
          "close": 1430
       },
       {
          "dayOfWeek": 2,
          "open": 1130,
          "close": 1430
       },
       …
    ]
}
```

MongoDB = easy to query

```
{
   "serviceArea": "94619",
   "openingHours": {
      "$elemMatch": {
         "open": { "$lte": 1815},
         "dayOfWeek": 4,
         "close": {  $gte": 1815}
      }
   }
```

```
db.availableRestaurants.ensureIndex({serviceArea: 1})
```

MongoTemplate-based code

```
@Repository
public class AvailableRestaurantRepositoryMongoDbImpl
                                    implements AvailableRestaurantRepository {

@Autowired private final MongoTemplate mongoTemplate;

@Autowired @Override
public List<AvailableRestaurant> findAvailableRestaurants(Address deliveryAddress,
                                                          Date deliveryTime) {
  int timeOfDay = DateTimeUtil.timeOfDay(deliveryTime);
  int dayOfWeek = DateTimeUtil.dayOfWeek(deliveryTime);

Query query = new Query(where("serviceArea").is(deliveryAddress.getZip())
          .and("openingHours").elemMatch(where("dayOfWeek").is(dayOfWeek)
                    .and("openingTime").lte(timeOfDay)
                    .and("closingTime").gte(timeOfDay)));

  return mongoTemplate.find(AVAILABLE_RESTAURANTS_COLLECTION, query,
                                    AvailableRestaurant.class);
}
```

```
mongoTemplate.ensureIndex("availableRestaurants",
      new Index().on("serviceArea", Order.ASCENDING));
```

MongoDB – Spring Configuration

```
@Configuration
public class MongoConfig extends AbstractDatabaseConfig {
  private @Value("#{mongoDbProperties.databaseName}")
  String mongoDbDatabase;

  public @Bean MongoFactoryBean mongo() {
    MongoFactoryBean factory = new MongoFactoryBean();
    factory.setHost(databaseHostName);
    MongoOptions options = new MongoOptions();
    options.connectionsPerHost = 500;
    factory.setMongoOptions(options);
    return factory;
  }

  public @Bean
  MongoTemplate mongoTemplate(Mongo mongo) throws Exception {
    MongoTemplate mongoTemplate = new MongoTemplate(mongo, mongoDbDatabase);
    mongoTemplate.setWriteConcern(WriteConcern.SAFE);
    mongoTemplate.setWriteResultChecking(WriteResultChecking.EXCEPTION);
    return mongoTemplate;
  }
}
```

Is NoSQL webscale?

Benchmarking is **still work in progress** but so far

http://www.youtube.com/watch?v=b2F-DItXtZs

	Redis	Mongo	Cassandra
Insert for PK	Awesome	Awesome	Fast*
Find by PK	Awesome	Awesome	Fast
Insert for find available	Fast	Awesome	Ok*
Find available restaurants	Awesome	Ok	Ok

* Cassandra can be clustered for improved write performance

In other words: it depends

Summary

Relational databases are great but

- Object/relational impedance mismatch
- Relational schema is rigid
- Extremely difficult/impossible to scale writes
- Performance can be suboptimal

Each NoSQL databases can solve some combination of those problems BUT

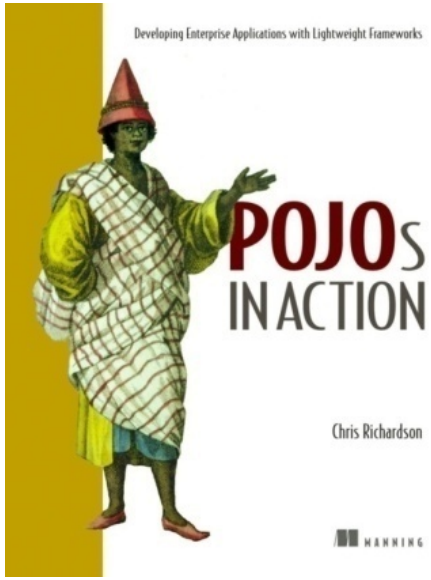
- Limited transactions
- Query-driven, denormalized database design
- …

➔

Carefully pick the NoSQL DB for your application

Consider a polyglot persistence architecture

Thank you!



My contact info

chris.richardson@springsource.com

@crichardson