

# SQL, NoSQL, NewSQL? What's a developer to do?

---

**Chris Richardson**

Author of POJOs in Action

Founder of the original CloudFoundry.com

chris.richardson@springsource.com

 @crichardson

Blog: <http://plainoldobjects.com>

# Overall presentation goal

---

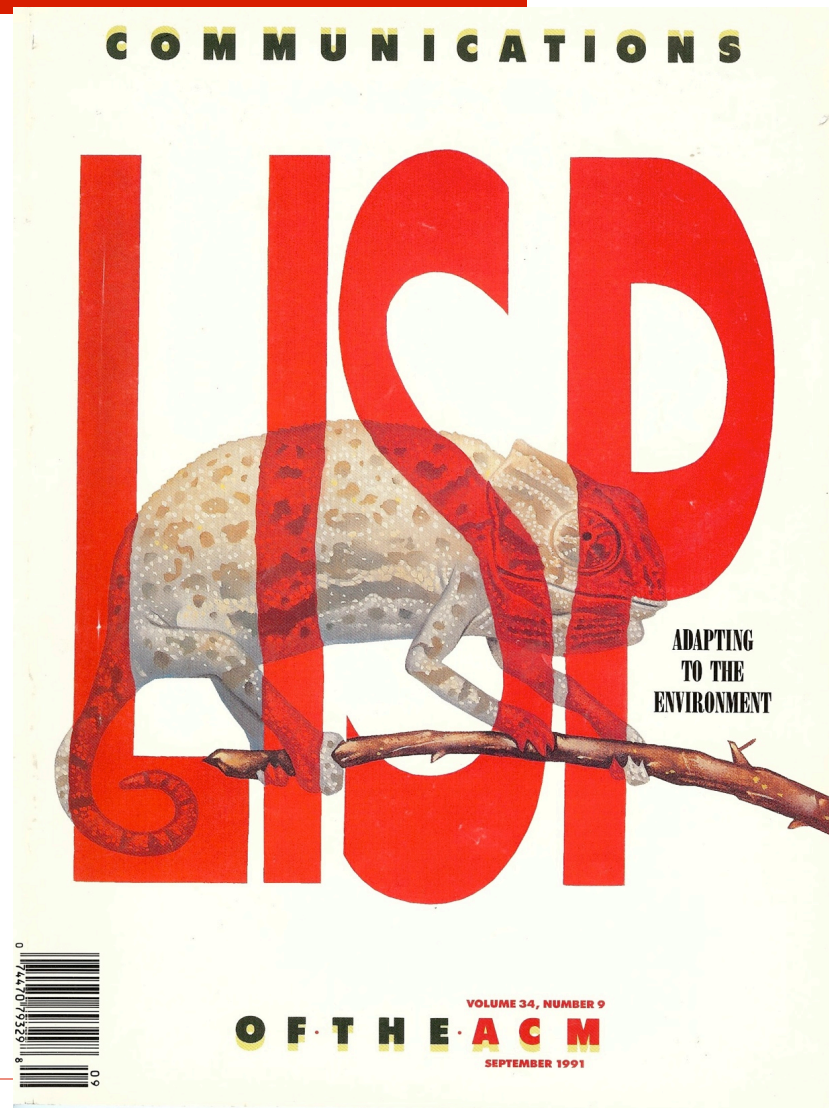
The joy and pain of  
building Java  
applications that use  
NoSQL and NewSQL

# About Chris

---



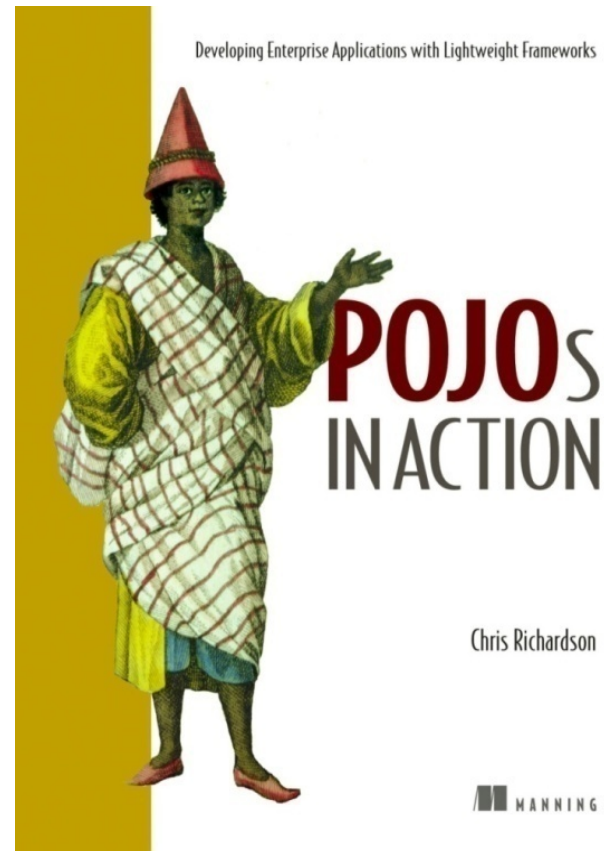
# (About Chris)





# About Chris()

---



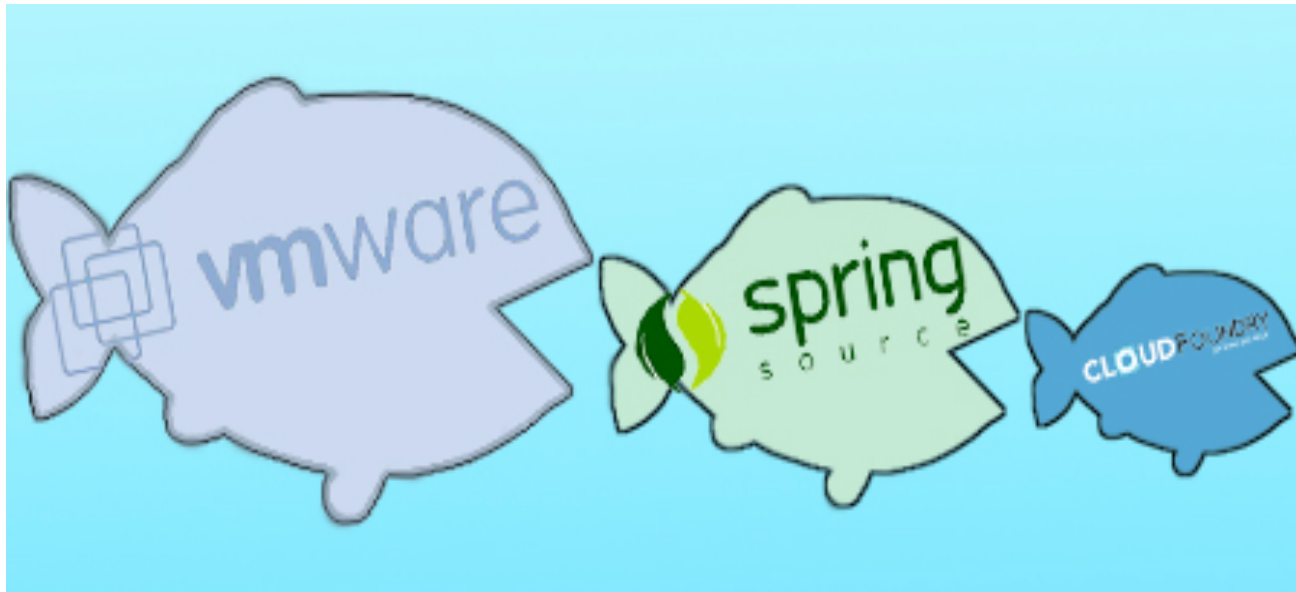
# About Chris

---

The screenshot shows the Cloud Foundry website homepage. At the top left is the Cloud Foundry logo with 'SPRINGSOURCE' underneath. To the right is a sign-in form with fields for 'Email' and a password, and a 'SIGN IN' button. Below the sign-in form are links for 'Sign Up' and 'Forgot password?'. A navigation bar contains dropdown menus for 'HOW WE HELP', 'FEATURES', and 'INFORMATION', along with 'BLOG' and 'CONTACT US' buttons. A 'SIGN UP BETA' button is on the right. A dark blue banner with a red exclamation mark icon contains the text: 'SYSTEM ALERT. PLEASE READ: Cloud Foundry will be moving to a new URL. [More](#)'. The main content area has a green background and features the heading 'The Enterprise Java Cloud'. Below this are three bullet points: 'Real Java Applications Deployed in Minutes', 'Built for Spring and Grails Web Applications', and 'Most Widely Used Technologies Delivered as a Platform'. At the bottom of this section are 'SIGN UP BETA' and 'LEARN MORE' buttons. On the right is a video player with the Cloud Foundry logo, a play button, and the text 'APPLICATION DEMO: Deploying Web Applications To Amazon EC2 with Cloud Foundry'.

# About Chris

---



[http://www.theregister.co.uk/2009/08/19/springsource\\_cloud\\_foundry/](http://www.theregister.co.uk/2009/08/19/springsource_cloud_foundry/)

# About Chris

---

Developer Advocate for



# Agenda

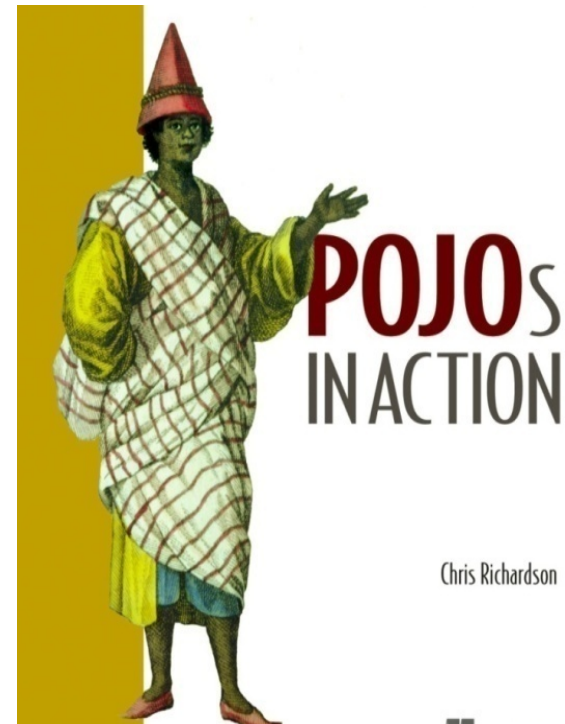
---

- **Why NoSQL? NewSQL?**
- Persisting entities
- Implementing queries

# Food to Go

---

- ❑ Take-out food delivery service
- ❑ “Launched” in 2006
- ❑ Used a relational database (naturally)



# Success → Growth challenges

---

- ❑ Increasing traffic
  - ❑ Increasing data volume
  - ❑ Distribute across a few data centers
  - ❑ Increasing domain model complexity
-



# Limitations of relational databases

---

- ❑ Scaling
- ❑ Distribution
- ❑ Updating schema
- ❑ O/R impedance mismatch
- ❑ Handling semi-structured data

# Solution: Spend Money

---



[http://upload.wikimedia.org/wikipedia/commons/e/e5/Rising\\_Sun\\_Yacht.JPG](http://upload.wikimedia.org/wikipedia/commons/e/e5/Rising_Sun_Yacht.JPG)

- Hire more DevOps
- Use application-level sharding
- Build your own middleware
- ...

- Buy SSD and RAM
- Buy Oracle
- Buy high-end servers
- ...

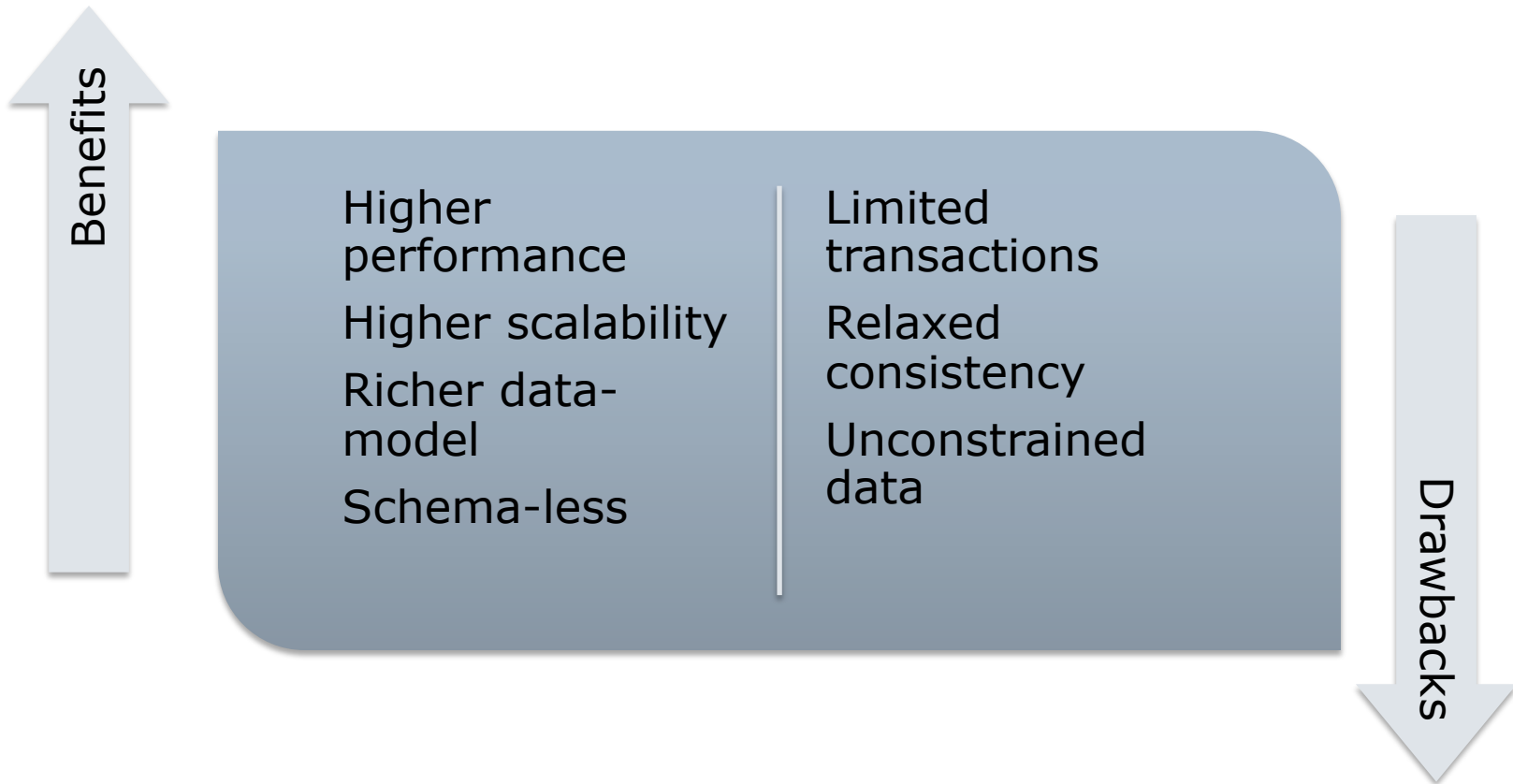
**OR**



[http://www.trekbikes.com/us/en/bikes/road/race\\_performance/madone\\_5\\_series/madone\\_5\\_2/#](http://www.trekbikes.com/us/en/bikes/road/race_performance/madone_5_series/madone_5_2/#)

# Solution: Use NoSQL

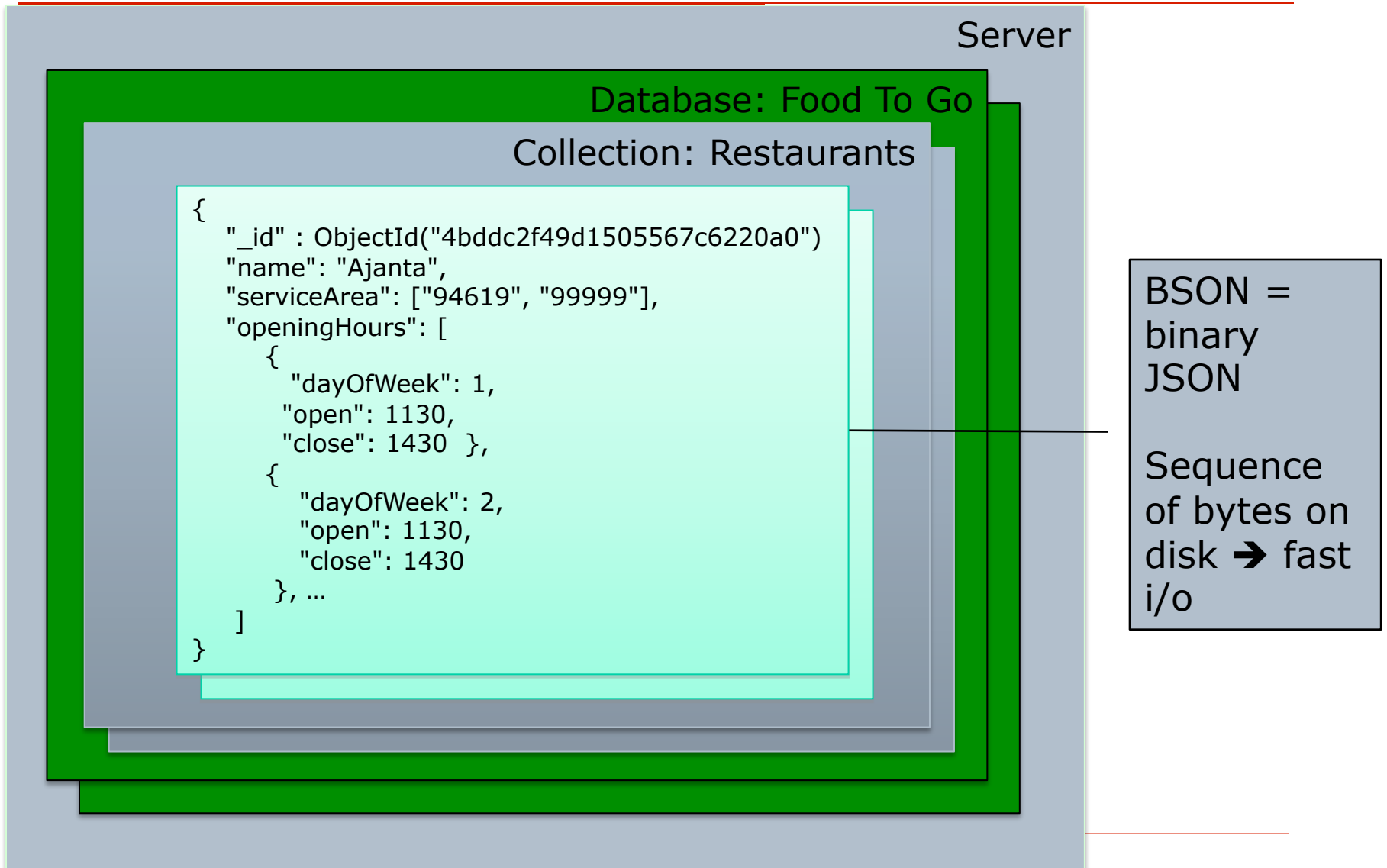
---



MongoDB

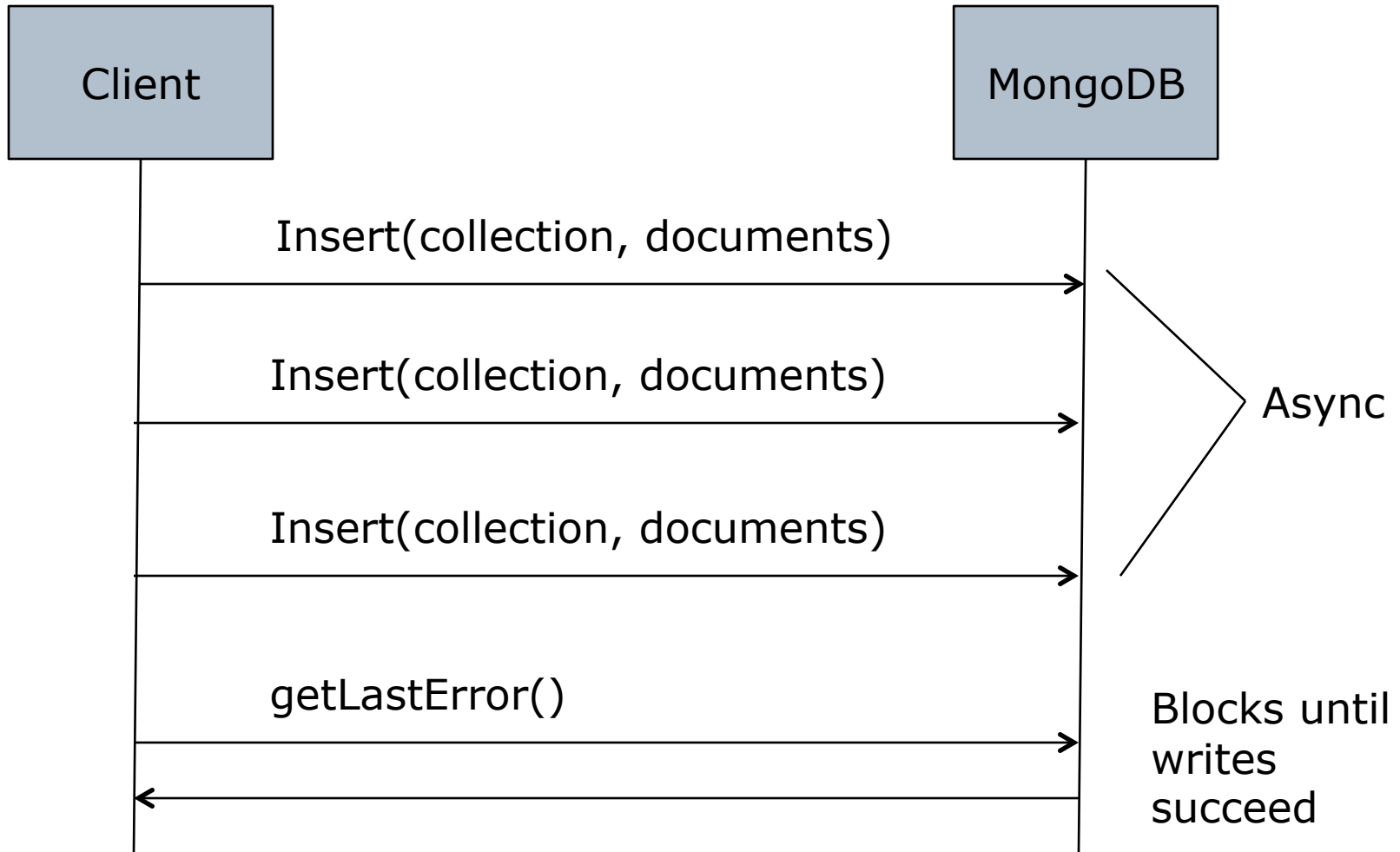


# MongoDB is a document oriented DB

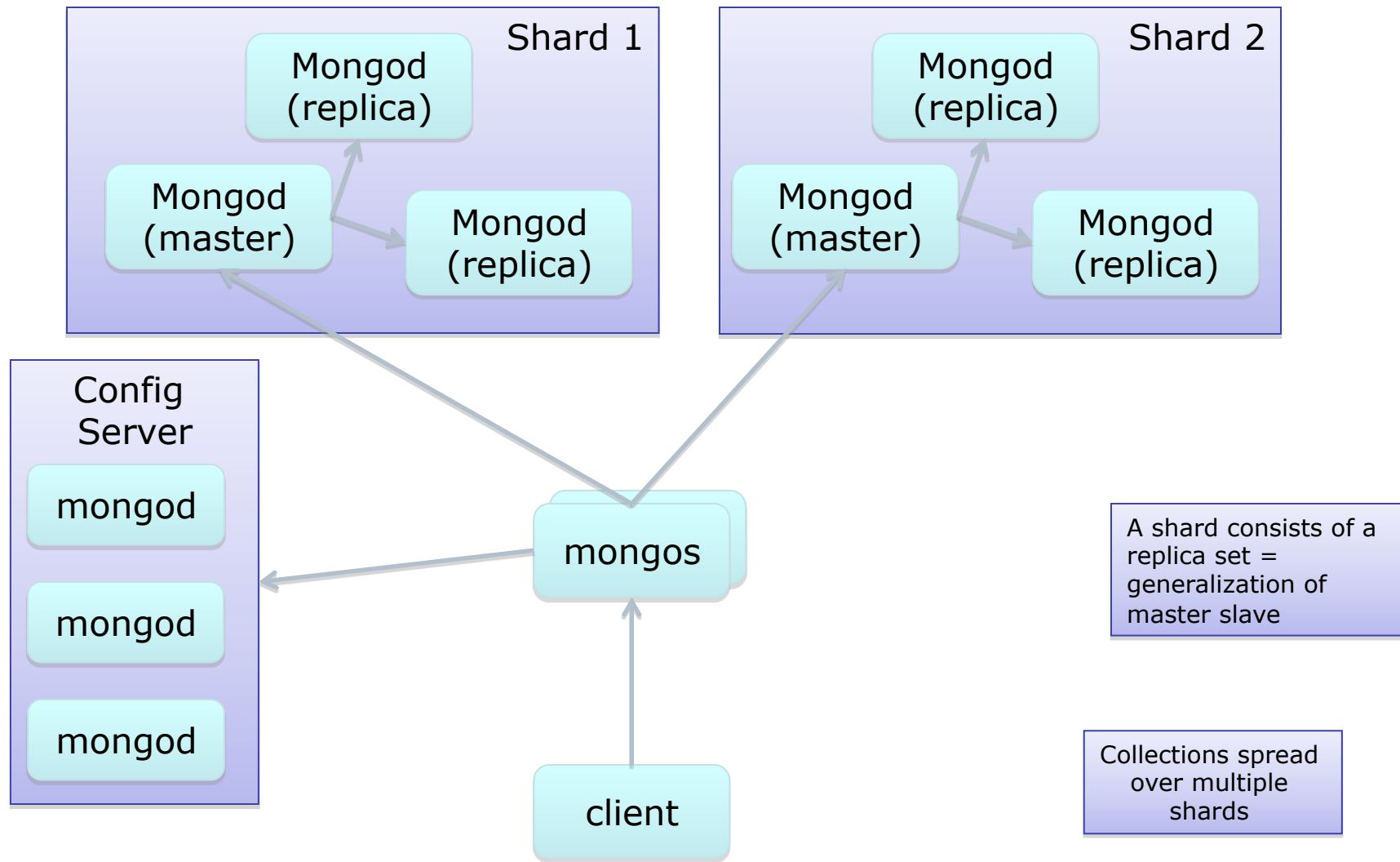


# MongoDB – writes are fast

---



# MongoDB is scalable





# MongoDB use cases

---

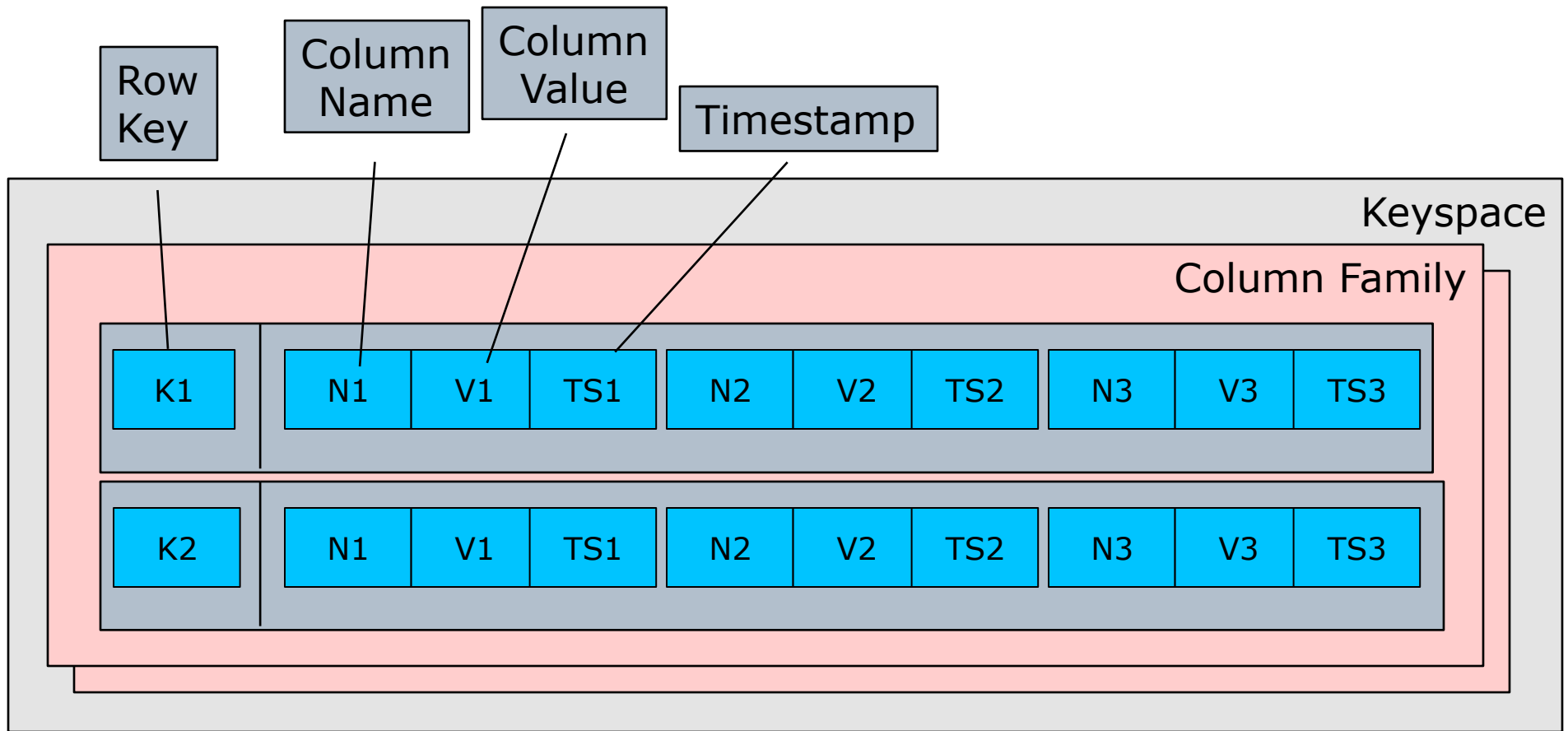
- Use cases
  - High volume writes
  - Complex data
  - Semi-structured data
- Used by: Shutterfly, Foursquare, Bit.ly, ...

# Apache Cassandra



# Cassandra is a column-oriented DB

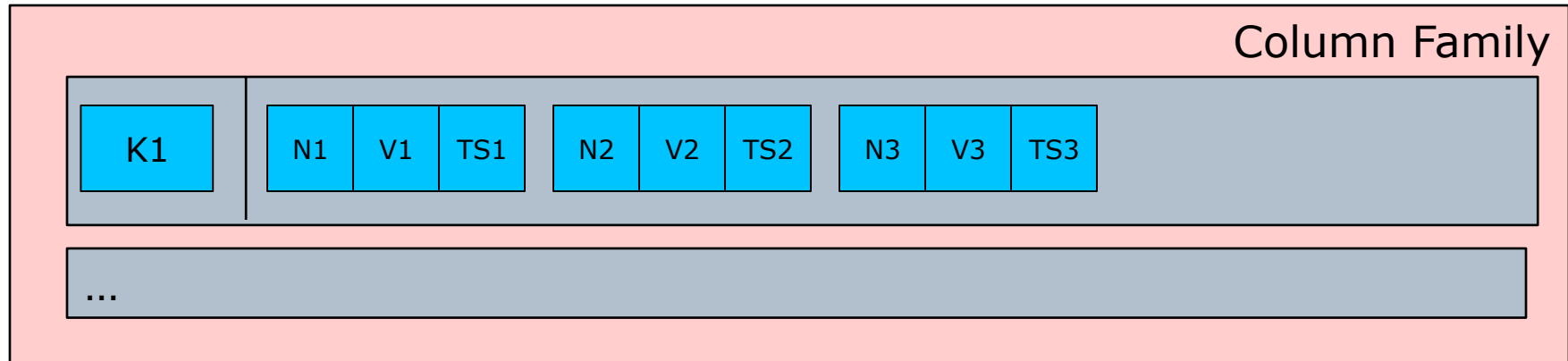
---



Column name/value: number, string, Boolean, timestamp, and **composite**

---

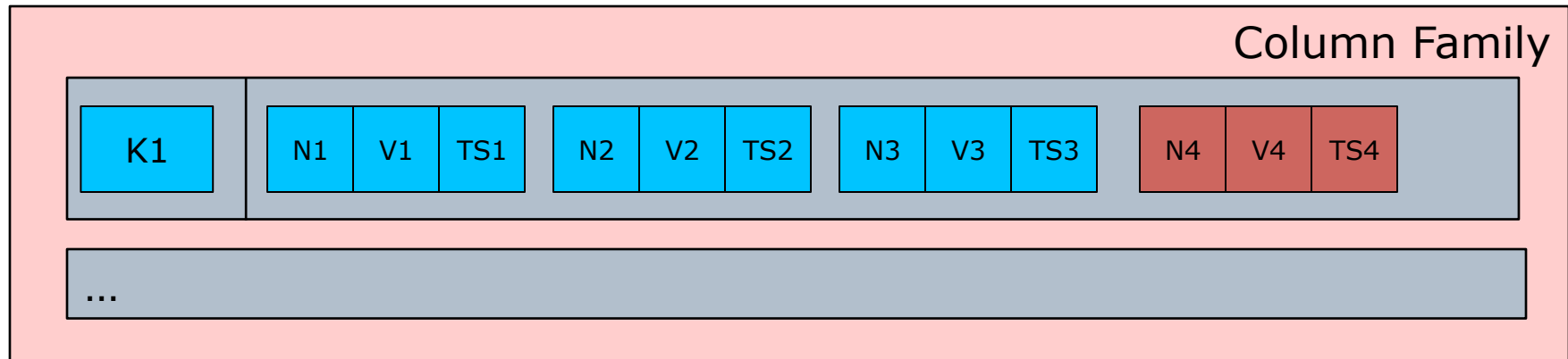
# Cassandra– inserting/updating data



Idempotent= transaction

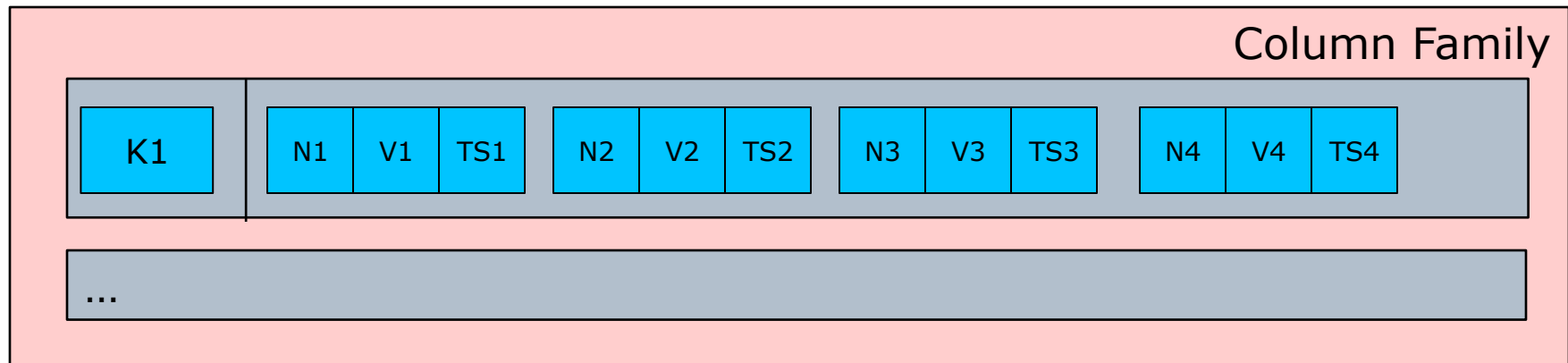


CF.insert(key=K1, (N4, V4, TS4), ...)

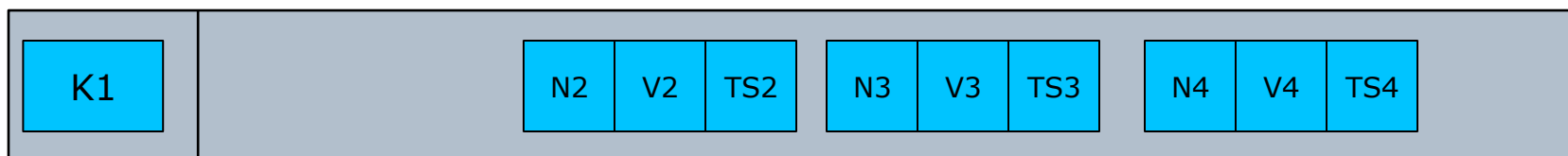


# Cassandra– retrieving data

---



`CF.slice(key=K1, startColumn=N2, endColumn=N4)`

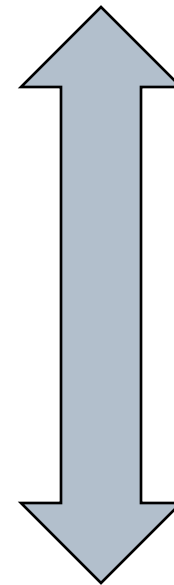


# Tunable reads and writes

---

- Any node (for writes)
- One replica
- Quorum of replicas
- Local quorum
- Each quorum
- All replicas

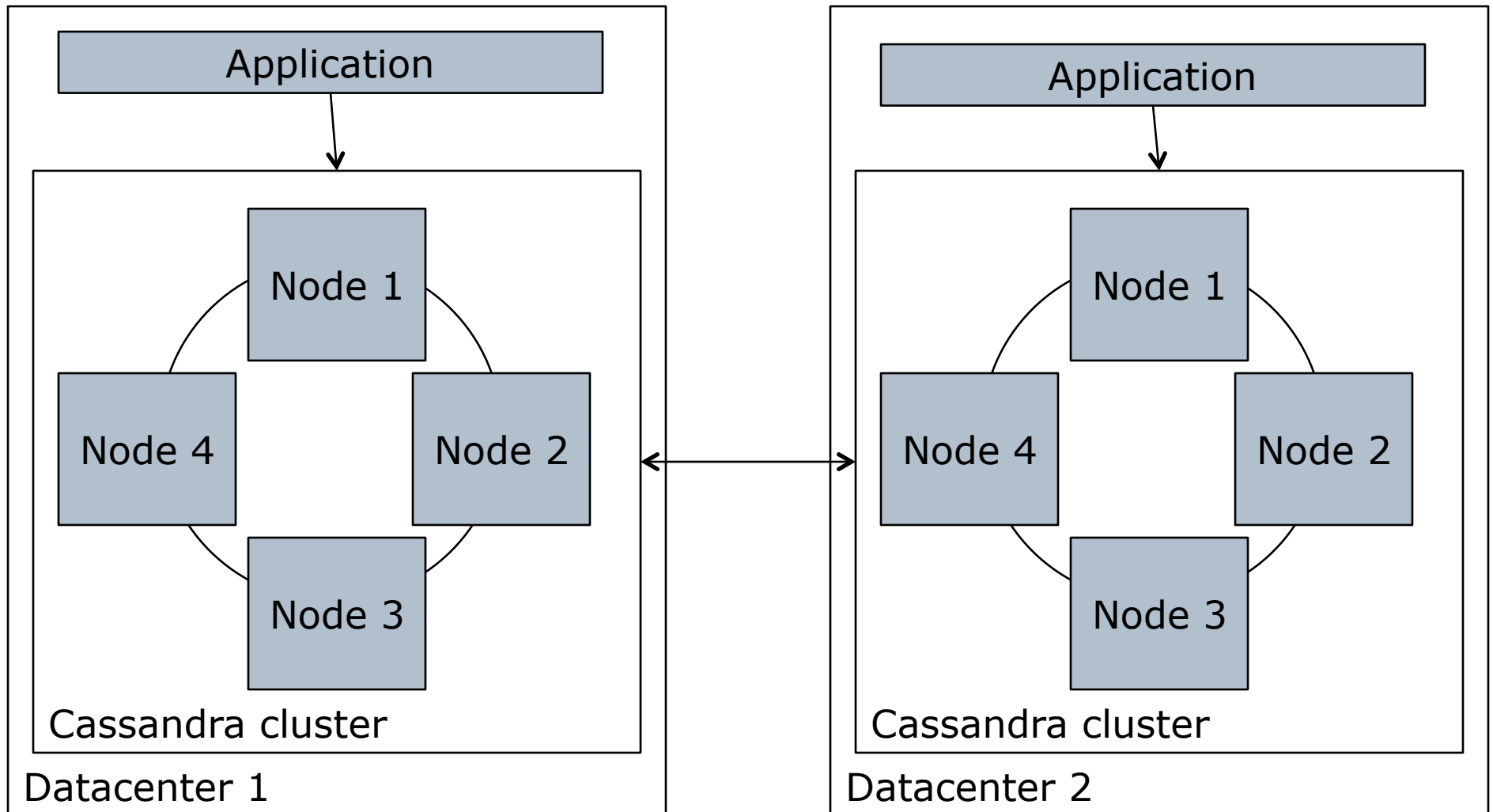
Higher availability  
Lower response time  
Less consistency



Lower availability  
Higher response time  
More consistency

# Cassandra is scalable

---





# Cassandra use cases

---

- Use cases
  - Big data
  - Multiple Data Center distributed database
  - (Write intensive) Logging
  - High-availability (writes)
- Used by: Netflix, Facebook, Digg, etc.

# Other NoSQL databases

---

Type	Examples
Extensible columns/Column-oriented	Hbase SimpleDB DynamoDB
Graph	Neo4j
Key-value	Redis Membase
Document	CouchDb

*sorry if I left out your favorite*

<http://nosql-database.org/> lists 122+ NoSQL databases

# Solution: Use **NewSQL**

---

- ❑ Relational databases with SQL and ACID transactions

**AND**

- ❑ New and improved architecture
- ❑ Radically better scalability and performance
  
- ❑ NewSQL vendors: ScaleDB, NimbusDB, ..., VoltDB

# Stonebraker's motivations

---



“...Current databases are designed for 1970s hardware ...”

Stonebraker: <http://www.slideshare.net/VoltDB/sql-myths-webinar>

Significant overhead in “...logging, latching, locking, B-tree, and buffer management operations...”

SIGMOD 08: Though the looking glass: <http://dl.acm.org/citation.cfm?id=1376713>

# About VoltDB

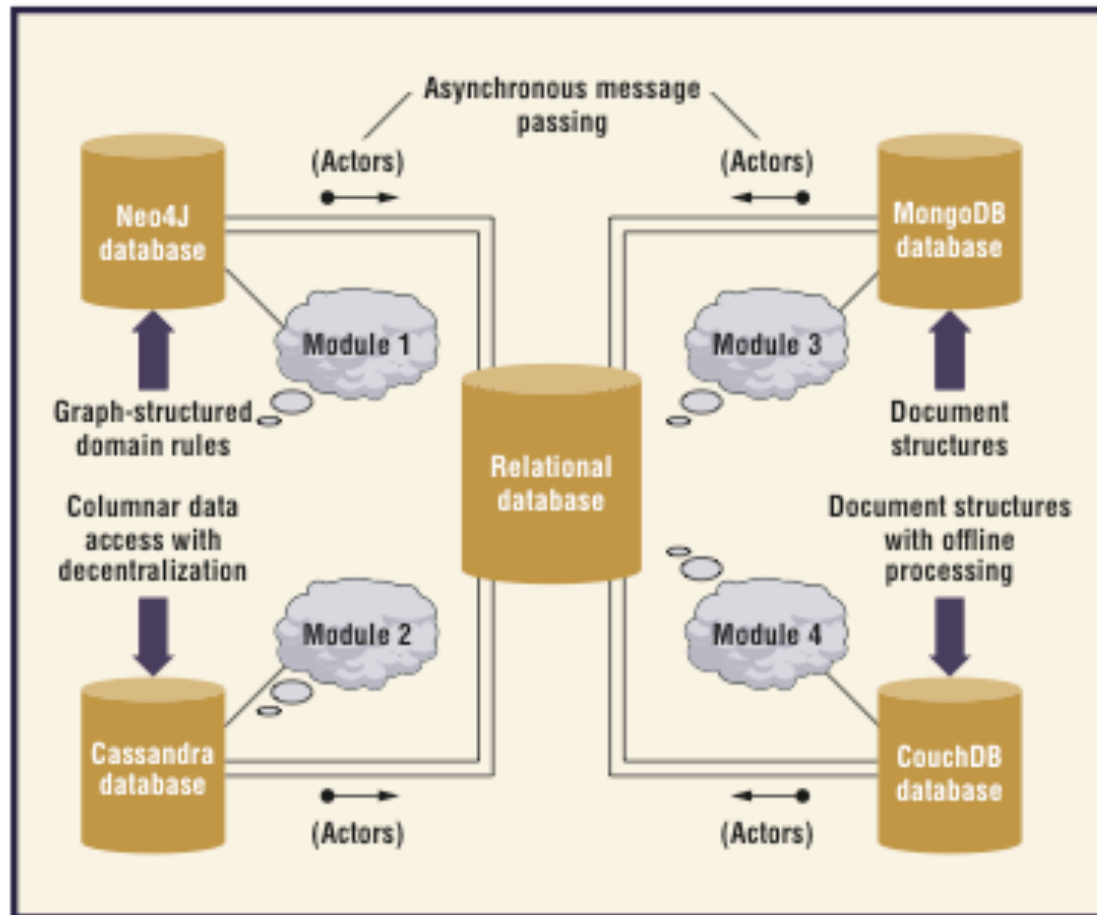
---

- ❑ Open-source
- ❑ In-memory relational database
- ❑ Durability thru replication; snapshots and logging
- ❑ Transparent partitioning
- ❑ Fast and scalable

...VoltDB is very scalable; it should scale to 120 partitions, 39 servers, and 1.6 million complex transactions per second at over 300 CPU cores...

<http://www.mysqlperformanceblog.com/2011/02/28/is-voltdb-really-as-scalable-as-they-claim/>

# The future is polyglot persistence



- e.g. Netflix
- RDBMS
  - SimpleDB
  - Cassandra
  - Hadoop/Hbase

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

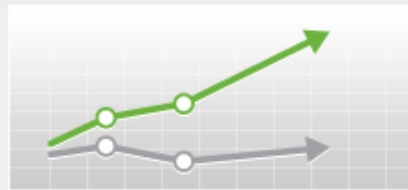
# Spring Data is here to help

---

## SPRING KEY BENEFITS



Modularity



Productivity



Portability



Testability

**For**

**NoSQL databases**

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



# Agenda

---

- Why NoSQL? NewSQL?
- **Persisting entities**
- Implementing queries

# Food to Go – Place Order use case

---

1. Customer enters delivery address and delivery time
2. System displays available restaurants
3. Customer picks restaurant
4. System displays menu
5. Customer selects menu items
6. Customer places order

# Food to Go – Domain model (partial)

---

```
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

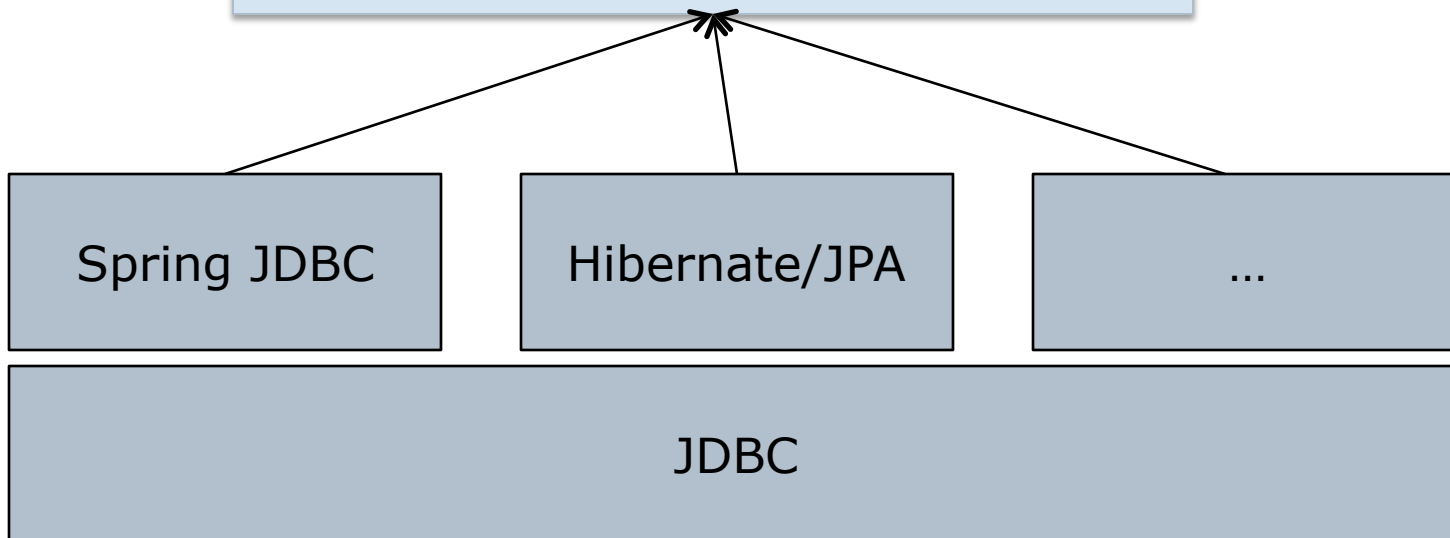
RESTAURANT\_TIME\_RANGE table

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

# Pick your JDBC-based framework

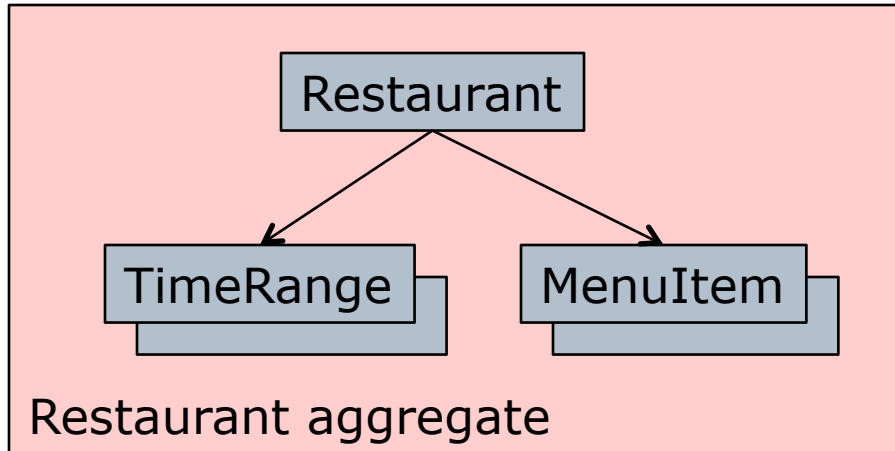
---

```
interface AvailableRestaurantRepository {  
    void add(Restaurant restaurant);  
    Restaurant findDetailsById(int id);  
    ...  
}
```

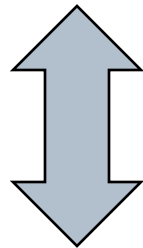


# But with NoSQL?

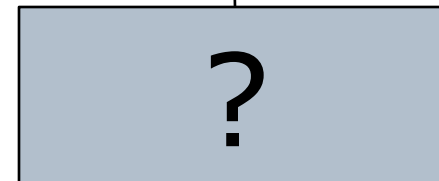
---



```
interface AvailableRestaurantRepository {  
    void add(Restaurant restaurant);  
    Restaurant findDetailsById(int id);  
    ...  
}
```



?



MongoDB



# MongoDB: persisting restaurants is easy

---

```
{
  "_id" : ObjectId("4bddc2f49d1505567c6220a0")
  "name": "Ajanta",
  "serviceArea": ["94619", "99999"],
  "openingHours": [
    {
      "dayOfWeek": 1,
      "open": 1130,
      "close": 1430 },
    {
      "dayOfWeek": 2,
      "open": 1130,
      "close": 1430
    }, ...
  ]
}
```



# Spring Data for Mongo code

---

```
@Repository
public class AvailableRestaurantRepositoryMongoDbImpl
    implements AvailableRestaurantRepository {

    public static String AVAILABLE_RESTAURANTS_COLLECTION = "availableRestaurants";

    @Autowired
    private MongoTemplate mongoTemplate;

    @Override
    public void add(Restaurant restaurant) {
        mongoTemplate.insert(restaurant, AVAILABLE_RESTAURANTS_COLLECTION);
    }

    @Override
    public Restaurant findDetailsById(int id) {
        return mongoTemplate.findOne(new Query(where("_id").is(id)),
            Restaurant.class,
            AVAILABLE_RESTAURANTS_COLLECTION);
    }
}
```

# Spring Configuration

---

```
@Configuration
public class MongoConfig extends AbstractDatabaseConfig {

    @Value("#{mongoDbProperties.databaseName}")
    private String mongoDbDatabase;

    @Bean
    public Mongo mongo() throws UnknownHostException, MongoException {
        return new Mongo(databaseHostName);
    }

    @Bean
    public MongoTemplate mongoTemplate(Mongo mongo) throws Exception {
        MongoTemplate mongoTemplate = new MongoTemplate(mongo, mongoDbDatabase);
        ...
        return mongoTemplate;
    }
}
```

---

# Apache Cassandra



# Option #1: Use a column per attribute

Column Name = path/expression to access property value

Column Family: RestaurantDetails

	Column Family: RestaurantDetails					
1	name	Ajanta	serviceArea[0]	94619	openingHours[0].dayOfWeek	Monday
	type	indian	serviceArea[1]	94707	openingHours[0].open	1130
					openingHours[0].close	1430
2	name	Egg shop	serviceArea[0]	94611	openingHours[0].dayOfWeek	Monday
	type	Break Fast	serviceArea[1]	94619	openingHours[0].open	0830
					openingHours[0].close	1430

# Option #2: Use a single column

---

Column value = serialized object graph, e.g. JSON

Column Family: RestaurantDetails		
1	attributes	{ name: "Ajanta", ... }
2	attributes	{ name: "Eggshop", ... }

Can't use secondary indexes but they aren't helpful for these use cases



# Cassandra code: wrapper around Hector

```
public class AvailableRestaurantRepositoryCassandraKeyImpl
    implements AvailableRestaurantRepository {

    @Autowired
    private final CassandraTemplate cassandraTemplate;

    public void add(Restaurant restaurant) {
        cassandraTemplate.insertEntity(keyspace,
            RESTAURANT_DETAILS_CF,
            restaurant);
    }

    public Restaurant findDetailsById(int id) {
        String key = Integer.toString(id);
        return cassandraTemplate.findEntity(Restaurant.class,
            keyspace, key, RESTAURANT_DETAILS_CF);
    }
    ...
}
```

Home grown wrapper class



Hector brought back to Troy. From a Roman sarcophagus of ca. 180–200 AD.

<http://en.wikipedia.org/wiki/Hector>



# Using VoltDB

---

- ❑ Use the original schema
- ❑ Standard SQL statements

BUT YOU MUST

- ❑ Write stored procedures and invoke them using proprietary interface
- ❑ Partition your data



# About VoltDB stored procedures

---

- ❑ Key part of VoltDB
- ❑ Replication = executing stored procedure on replica
- ❑ Logging = log stored procedure invocation
- ❑ Stored procedure invocation = transaction

# About partitioning

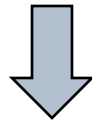
---

Partition column



RESTAURANT table

ID	Name	...
1	Ajanta	
2	Eggshop	
...		



Partition 1

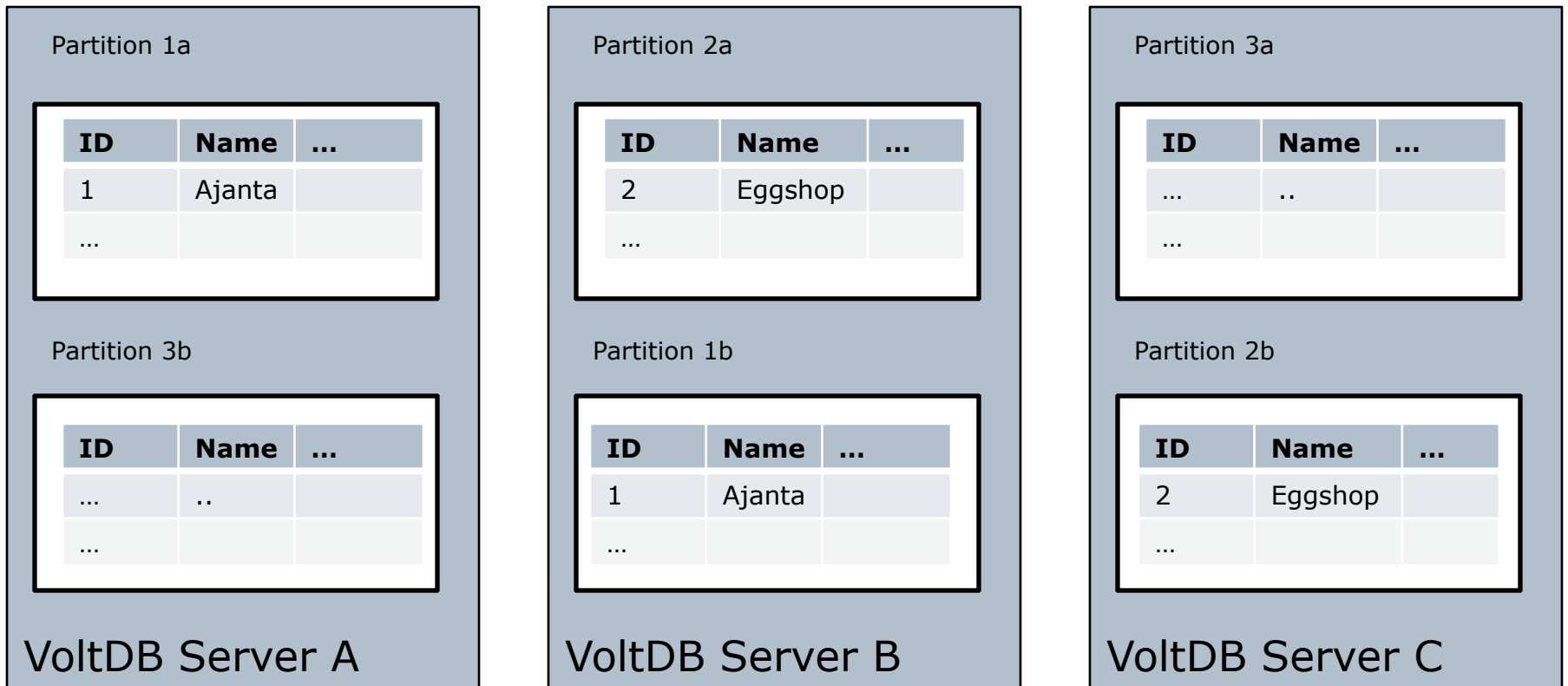
ID	Name	...
1	Ajanta	
...		

Partition 2

ID	Name	...
2	Eggshop	
...		

# Example VoltDB cluster

$$\frac{(3 \text{ servers}) \times (2 \text{ partitions per server})}{(2 \text{ replicas per partition})} \longrightarrow 3 \text{ partitions}$$



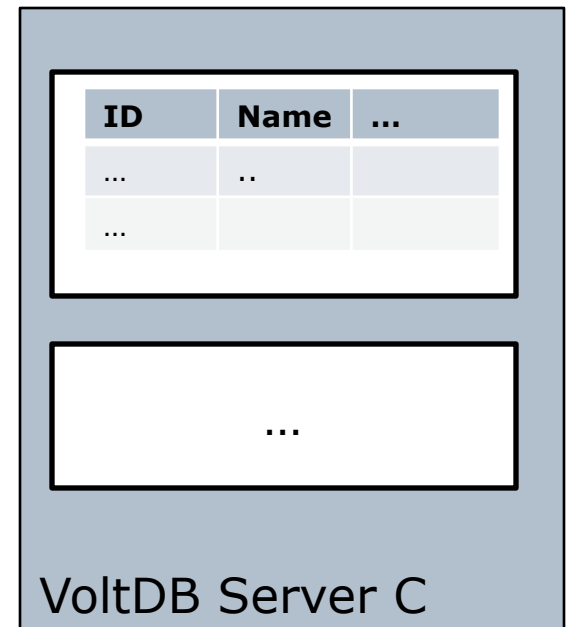
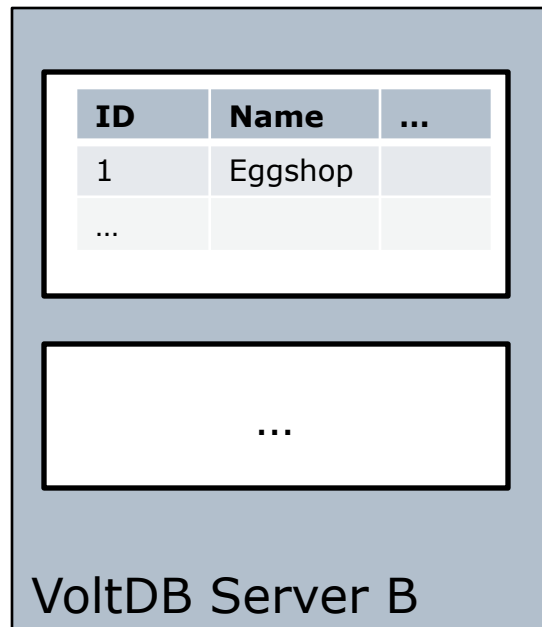
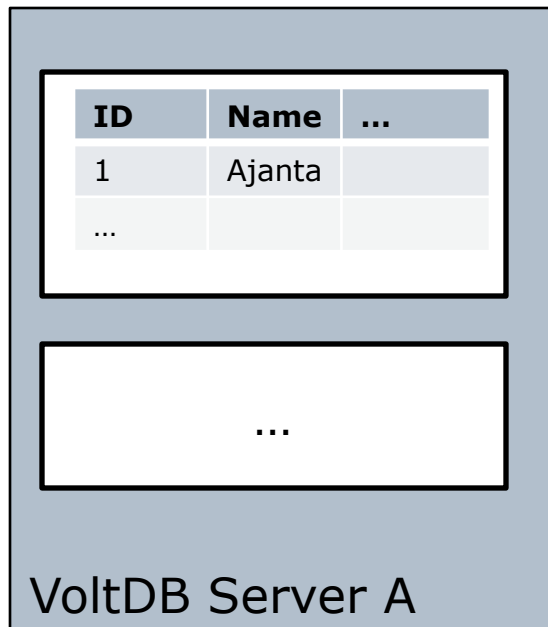
# Single partition procedure: FAST

```
SELECT * FROM RESTAURANT WHERE ID = 1
```

High-performance lock free code

Partition column

Stored procedure parameter

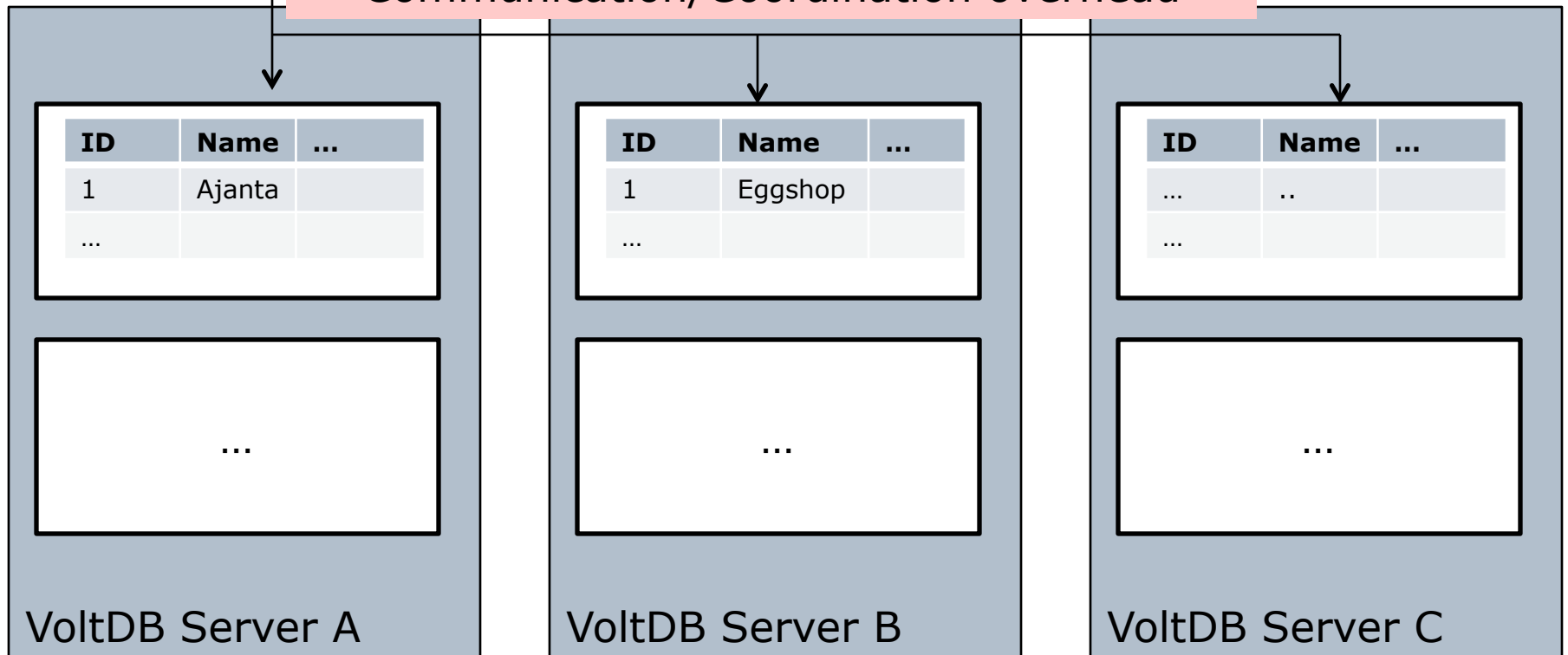


# Multi-partition procedure: SLOWER

---

SELECT \* FROM RESTAURANT WHERE NAME = 'Ajanta'

Communication/Coordination overhead



# Chosen partitioning scheme

---

```
<partitions>  
  <partition table="restaurant" column="id"/>  
  <partition table="service_area" column="restaurant_id"/>  
  <partition table="menu_item" column="restaurant_id"/>  
  <partition table="time_range" column="restaurant_id"/>  
  <partition table="available_time_range" column="restaurant_id"/>  
</partitions>
```

Performance is excellent: much faster than MySQL

# Stored procedure – AddRestaurant

---

```
@ProcInfo( singlePartition = true, partitionInfo = "Restaurant.id: 0")
public class AddRestaurant extends VoltProcedure {

    public final SQLStmt insertRestaurant =
        new SQLStmt("INSERT INTO Restaurant VALUES (?,?);");
    public final SQLStmt insertServiceArea =
        new SQLStmt("INSERT INTO service_area VALUES (?,?);");
    public final SQLStmt insertOpeningTimes =
        new SQLStmt("INSERT INTO time_range VALUES (?,?,?,?);");
    public final SQLStmt insertMenuItem =
        new SQLStmt("INSERT INTO menu_item VALUES (?,?,?);");

    public long run(int id, String name, String[] serviceArea, long[] daysOfWeek, long[] openingTimes,
        long[] closingTimes, String[] names, double[] prices) {

        voltQueueSQL(insertRestaurant, id, name);

        for (String zipCode : serviceArea)
            voltQueueSQL(insertServiceArea, id, zipCode);

        for (int i = 0; i < daysOfWeek.length ; i++)
            voltQueueSQL(insertOpeningTimes, id, daysOfWeek[i], openingTimes[i], closingTimes[i]);

        for (int i = 0; i < names.length ; i++)
            voltQueueSQL(insertMenuItem, id, names[i], prices[i]);

        voltExecuteSQL(true);

        return 0;
    }
}
```

# VoltDb repository – add()

---


```
@Repository
public class AvailableRestaurantRepositoryVoltdbImpl
    implements AvailableRestaurantRepository {

    @Autowired
    private VoltDbTemplate voltDbTemplate;

    @Override
    public void add(Restaurant restaurant) {
        invokeRestaurantProcedure("AddRestaurant", restaurant);
    }

    private void invokeRestaurantProcedure(String procedureName, Restaurant restaurant) {
        Object[] serviceArea = restaurant.getServiceArea().toArray();
        long[][] openingHours = toArray(restaurant.getOpeningHours());
        Object[][] menuItems = toArray(restaurant.getMenuItems());

        voltDbTemplate.update(procedureName, restaurant.getId(), restaurant.getName(),
            serviceArea, openingHours[0], openingHours[1],
            openingHours[2], menuItems[0], menuItems[1]);
    }
}
```




Flatten  
Restaurant



# VoltDbTemplate wrapper class

---

```
public class VoltDbTemplate {  
  
    private Client client;   
  
    public VoltDbTemplate(Client client) {  
        this.client = client;  
    }  
  
    public void update(String procedureName, Object... params) {  
        try {  
            ClientResponse x =  
                client.callProcedure(procedureName, params);  
  
            ...  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

# VoltDb server configuration

---

```
<?xml version="1.0"?>
<project>
  <info>
    <name>Food To Go</name>
  </info>
  <database>
    <schemas>
      <schema path='schema.sql' />
    </schemas>
    <partitions>
      <partition table="restaurant" column="id"/>
    </partitions>
    <procedures>
      <procedure class='net.chrisrichardson.foodToGo.newsqldb.procs.AddRestaurant' />
    </procedures>
  </database>
</project>
```

```
<deployment>
  <cluster hostcount="1"
    sitesperhost="5" kfactor="0" />
</deployment>
```

```
voltcompiler target/classes \
  src/main/resources/sql/voltdb-project.xml foodtogo.jar
```

```
bin/voltdb leader localhost catalog foodtogo.jar deployment deployment.xml
```

# Agenda

---

- Why NoSQL? NewSQL?
- Persisting entities
- **Implementing queries**

# Finding available restaurants

---

Available restaurants =

Serve the zip code of the delivery address

AND

Are open at the delivery time

```
public interface AvailableRestaurantRepository {  
  
    List<AvailableRestaurant>  
        findAvailableRestaurants(Address deliveryAddress,  
                                Date deliveryTime); ...  
}
```

## Finding available restaurants on Monday, 6.15pm for 94619 zip

```
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 <= 1815
and 1815 <= tr.closingtime
```

Straightforward  
three-way join

MongoDB



# MongoDB = easy to query

---

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

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

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

Find a restaurant that serves the 94619 zip code and is open at 6.15pm on a Monday

# MongoTemplate-based code

```
@Repository
public class AvailableRestaurantRepositoryMongoDbImpl
    implements AvailableRestaurantRepository {

    @Autowired private final MongoTemplate mongoTemplate;

    @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));
}
```



But how do this with  
Apache Cassandra??!



# Slice is equivalent to a simple SELECT

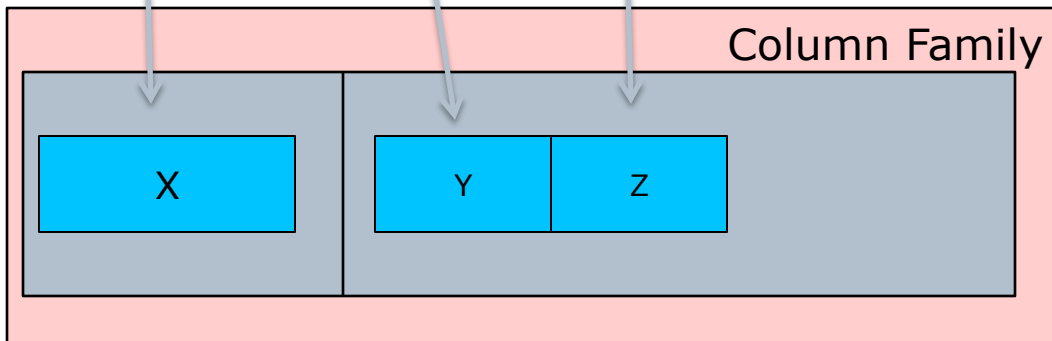
---

keyVal	colName	colValue
X	Y	Z
...		

```
select key, colName, colValue
from columnFamily
where key = keyVal
and colName >= startVal
and colName <= endVal
```



```
columnFamily.slice(key=keyVal,
startColumn=startVal,
endColumn=endVal)
```



**We need to implement an index  
that can be queried using a slice**

**Queries** instead of data  
model drives NoSQL  
database design

# Simplification #1: Denormalization

---

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

Simpler query:

- No joins
- Two = and two <

# Simplification #2: Application filtering

---

```
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
```

Even simpler query

- No joins
- Two = and one <

# Simplification #3: Eliminate multiple '='s with concatenation

---

Restaurant_id	Zip_dow	Open_time	Close_time
1	94707:Monday	1130	1430
1	94619:Monday	1130	1430
1	94707:Monday	1730	2130
1	94619:Monday	1730	2130
2	94619:Monday	0700	1430
...			

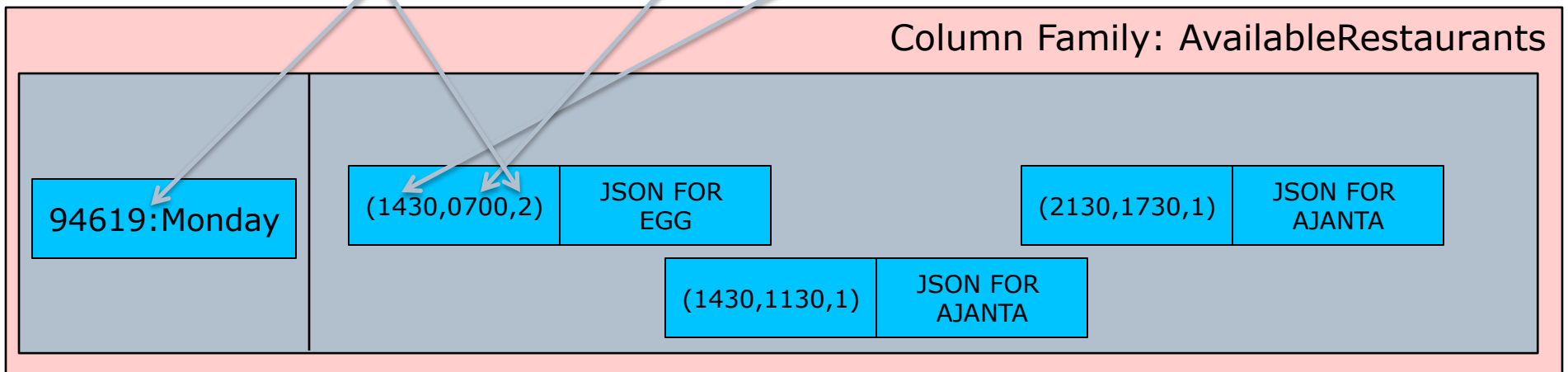
```
SELECT restaurant_id, open_time
FROM time_range_zip_code
WHERE zip_code_day_of_week = '94619:Monday'
AND 1815 < close_time
```

range

Row  
key

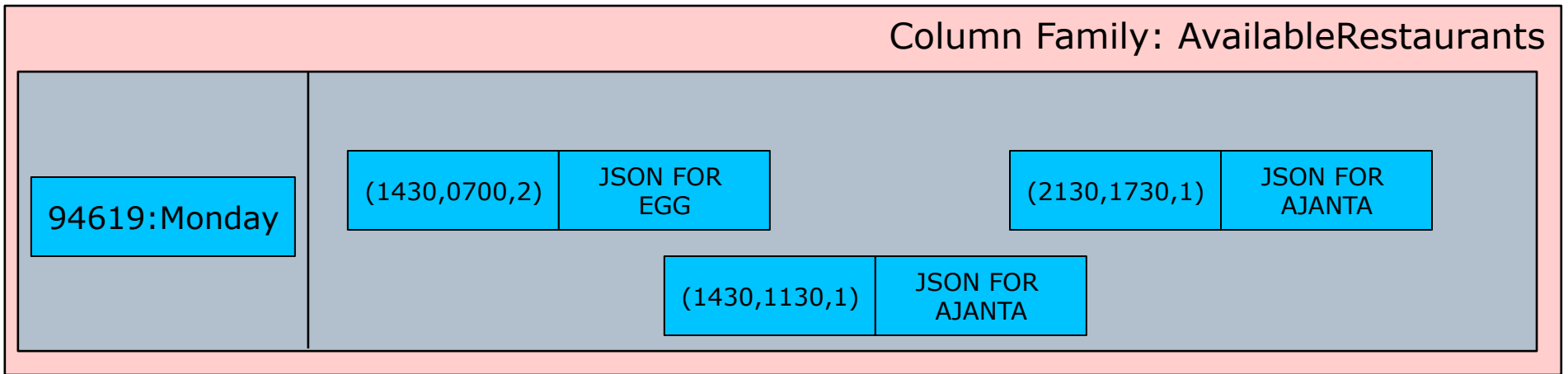
# Column family as an index

Restaurant_id	Zip_dow	Open_time	Close_time
1	94707:Monday	1130	1430
1	94619:Monday	1130	1430
1	94707:Monday	1730	2130
1	94619:Monday	1730	2130
2	94619:Monday	0700	1430
...			

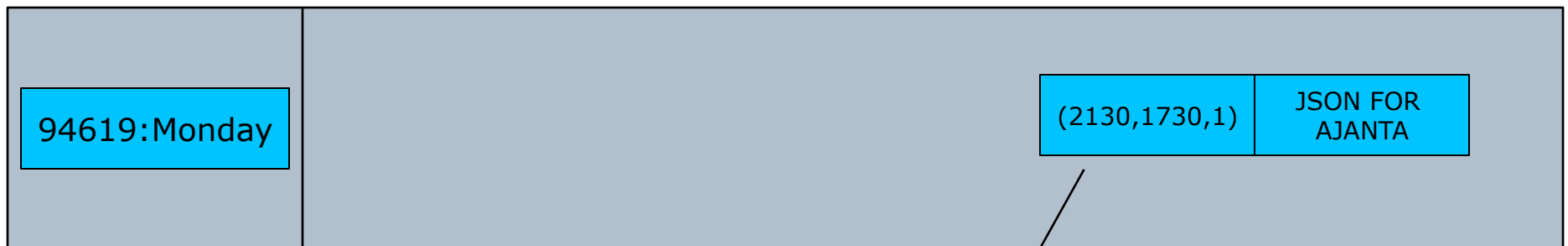
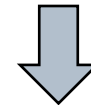


# Querying with a slice

Column Family: AvailableRestaurants



slice(key= 94619:Monday, sliceStart = (1815, \*, \*), sliceEnd = (2359, \*, \*))



18:15 is after 17:30 → {Ajanta}



# Needs a few pages of code

```
private void insertAvailability(Restaurant restaurant) {
```

```
    for (String zipCode : (Set<String>) restaurant.getServiceArea()) {
```

```
@Override
```

```
public List<AvailableRestaurant> findAvailableRestaurants(Address deliveryAddress, Date deliveryTime) {  
    int dayOfWeek = DateTimeUtil.dayOfWeek(deliveryTime);  
    int timeOfDay = DateTimeUtil.timeOfDay(deliveryTime);  
    String zipCode = deliveryAddress.getZip();  
    String key = formatKey(zipCode, format2(dayOfWeek));
```

```
    HSlicePredicate<Composite> predicate = new HSlicePredicate<Composite>(new CompositeSerializer());  
    Composite start = new Composite();  
    Composite finish = new Composite();  
    start.addComponent(0, format4(timeOfDay), ComponentEquality.GREATER_THAN_EQUAL);  
    finish.addComponent(0, format4(2359), ComponentEquality.GREATER_THAN_EQUAL);  
    predicate.setRange(start, finish, false, 100);
```

```
    final List<AvailableRestaurantIndexEntry> closingAfter = new ArrayList<AvailableRestaurantIndexEntry>();
```

```
    ColumnFamilyRowMapper<String, Composite, Object> mapper = new ColumnFamilyRowMapper<String, Composite, Object>() {
```

```
        @Override
```

```
        public Object mapRow(ColumnFamilyResult<String, Composite> results) {  
            for (Composite columnName : results.getColumnNames()) {  
                String openTime = columnName.get(1, new StringSerializer());  
                String restaurantId = columnName.get(2, new StringSerializer());  
                closingAfter.add(new AvailableRestaurantIndexEntry(openTime, restaurantId, results.getString(columnName)));  
            }  
            return null;  
        }  
    };
```

```
    compositeCloseTemplate.queryColumns(key, predicate, mapper);
```

```
    List<AvailableRestaurant> result = new LinkedList<AvailableRestaurant>();
```

```
    for (AvailableRestaurantIndexEntry trIdAndAvailableRestaurant : closingAfter) {  
        if (trIdAndAvailableRestaurant.isOpenBefore(timeOfDay))  
            result.add(trIdAndAvailableRestaurant.getAvailableRestaurant());  
    }
```

```
    return result;  
}
```

# What did I just do to query the data?

---

- ❑ Wrote code to maintain an index
- ❑ Reduced performance due to extra writes



# But what would you rather implement?

---

“Complex” query logic

OR

Multi-datacenter, multi-  
master database  
infrastructure



# VoltDB - attempt #0

---

```
@ProcInfo( singlePartition = false)
public class FindAvailableRestaurants extends VoltProcedure { ... }
```

```
SELECT r.*
FROM restaurant r,time_range tr, service_area sa
WHERE ? = sa.zip_code and sa.restaurant_id= tr.restaurant_id
AND r.restaurant_id = sa.restaurant_id and tr.day_of_week=?
AND tr.open_time <= ?
AND ? <= tr.close_time
```

Slow 😞

# VoltDB - attempt #1

```
@ProcInfo( singlePartition = false)
public class FindAvailableRestaurants extends VoltProcedure { ... }
```

```
create index idx_service_area_zip_code on service_area(zip_code);
..
```

```
ERROR 10:12:03,251 [main] COMPILER: Failed to plan for statement
type(findAvailableRestaurants_with_join) select r.* from restaurant
r,time_range tr, service_area sa Where ? = sa.zip_code and
sa.restaurant_id= tr.restaurant_id and r.restaurant_id =
sa.restaurant_id and tr.day_of_week=? and tr.open_time <= ? and ?
<= tr.close_time Error: "Unable to plan for statement. Possibly joining
partitioned tables in a multi-partition procedure using a column that is
not the partitioning attribute or a non-equality operator. This is
statement not supported at this time."
2012-03-19 08:19:31,743 ERROR [main] COMPILER: Catalog
compilation failed.
```

**#fail**

# VoltDB - attempt #2

---

```
@ProcInfo( singlePartition = true, partitionInfo = "Restaurant.id: 0")
public class AddRestaurant extends VoltProcedure {

    public final SQLStmt insertAvailable=
        new SQLStmt("INSERT INTO available_time_range VALUES (?,?,, ?, ?, ?);");

    public long run(...) {
        ...
        for (int i = 0; i < daysOfWeek.length ; i++) {
            voltQueueSQL(insertOpeningTimes, id, daysOfWeek[i], openingTimes[i], closingTimes[i]);
            for (String zipCode : serviceArea) {
                voltQueueSQL(insertAvailable, id, daysOfWeek[i], openingTimes[i],
                    closingTimes[i], zipCode, name);
            }
        }
        ...
        voltExecuteSQL(true);
        return 0;
    }
}
```

```
public final SQLStmt findAvailableRestaurants_denorm = new SQLStmt(
    "select restaurant_id, name from available_time_range tr " +
    "where ? = tr.zip_code " +
    "and tr.day_of_week=? " +
    "and tr.open_time <= ? " +
    " and ? <= tr.close_time ");
```

Works but queries are only slightly faster than MySQL! ☹

# VoltDB - attempt #3

---

```
<partitions>
  ...
  <partition table="available_time_range" column="zip_code"/>
</partitions>
```

```
@ProcInfo( singlePartition = false, ...)
public class AddRestaurant extends VoltProcedure { ... }
```

```
@ProcInfo( singlePartition = true,
           partitionInfo = "available_time_range.zip_code: 0")
public class FindAvailableRestaurants extends VoltProcedure { ... }
```

Queries are really fast 😊  
But inserts are not ☹️



# VoltDB – key lesson

---

A given partitioning scheme  
can be good for some use  
cases but bad for others

# Summary

---

Different databases  
=  
Different tradeoffs

# ... Summary...

---

	<b>Benefits</b>	<b>Drawbacks</b>
RDBMS	<ul style="list-style-type: none"><li>•SQL</li><li>•ACID</li><li>•Familiar</li></ul>	<ul style="list-style-type: none"><li>•Lack of performance and scalability</li><li>•Difficult to distribute</li><li>•Schema updates</li><li>•Handling semi-structured data</li></ul>
NoSQL	<ul style="list-style-type: none"><li>•Scalability</li><li>•Performance</li><li>•Schema-less</li></ul>	<ul style="list-style-type: none"><li>•Lack of ACID</li><li>•Limited querying (some)</li></ul>
NewSQL	<ul style="list-style-type: none"><li>•ACID</li><li>•SQL</li><li>•Familiar</li><li>•Scalability</li><li>•Performance</li></ul>	<ul style="list-style-type: none"><li>•Proprietary API</li><li>•Schema updates</li><li>•Handling semi-structured data</li><li>•Not all use cases are performant</li></ul>

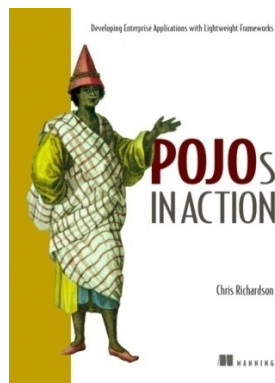
## ... Summary

---

- ❑ Very carefully pick the NewSQL/NoSQL DB for your application
- ❑ Consider a polyglot persistence architecture
- ❑ Encapsulate your data access code so you can switch
- ❑ Startups = avoid NewSQL/NoSQL for shorter time to market?

# Thank you!

---



**My contact info:**

**[chris.richardson@springsource.com](mailto:chris.richardson@springsource.com)**

** @crichardson**

**Blog: <http://plainoldobjects.com>**

---