COMS E6998: **Microservices and Cloud Applications**

Lecture 9: CAP Theorem, Database Models, XYZ-Scaling, 12 Factor Apps, Graph DB, Redis

Dr. Donald F. Ferguson
dff9@columbia.edu

Comments
Questions

Databases

Consistency
CAP
Theorem

Database Models

ACID Databases

(https://www.linkedin.com/pulse/what-acid-properties-database-aseem-jain/)

CAP Theorem

- Consistency

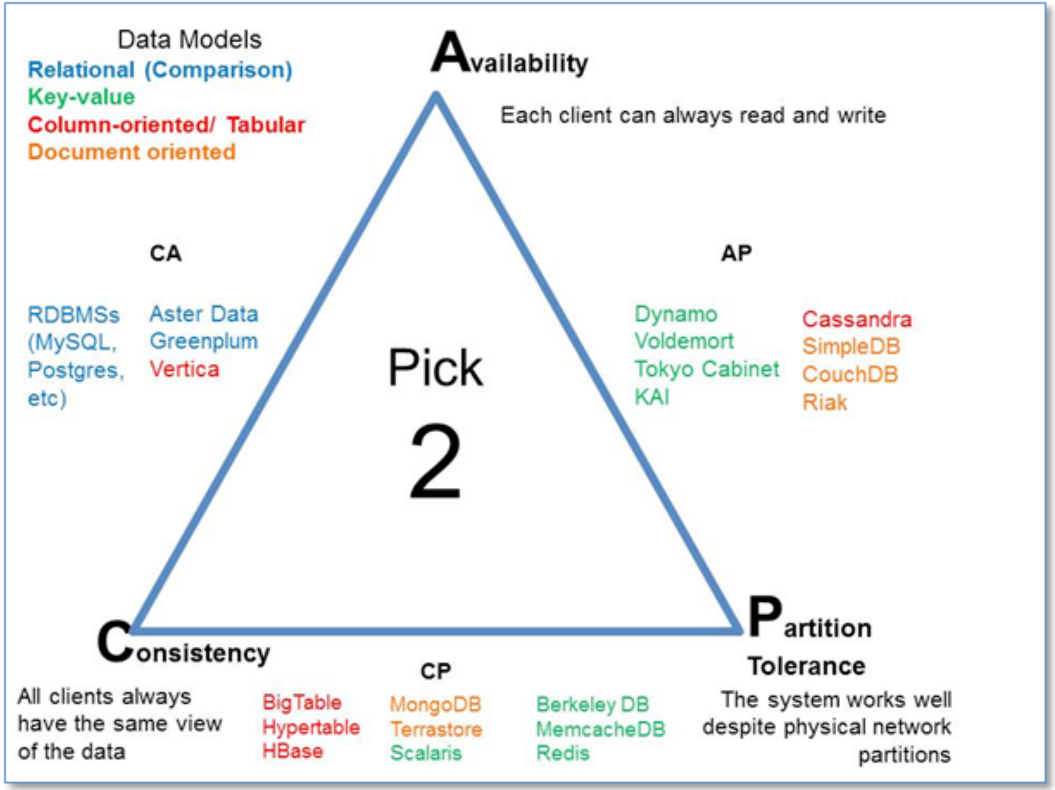
  Every read receives the most recent write or an error.

- Availability

  Every request receives a (non-error) response – without guarantee that it contains the most recent write.

- Partition Tolerance

  The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes



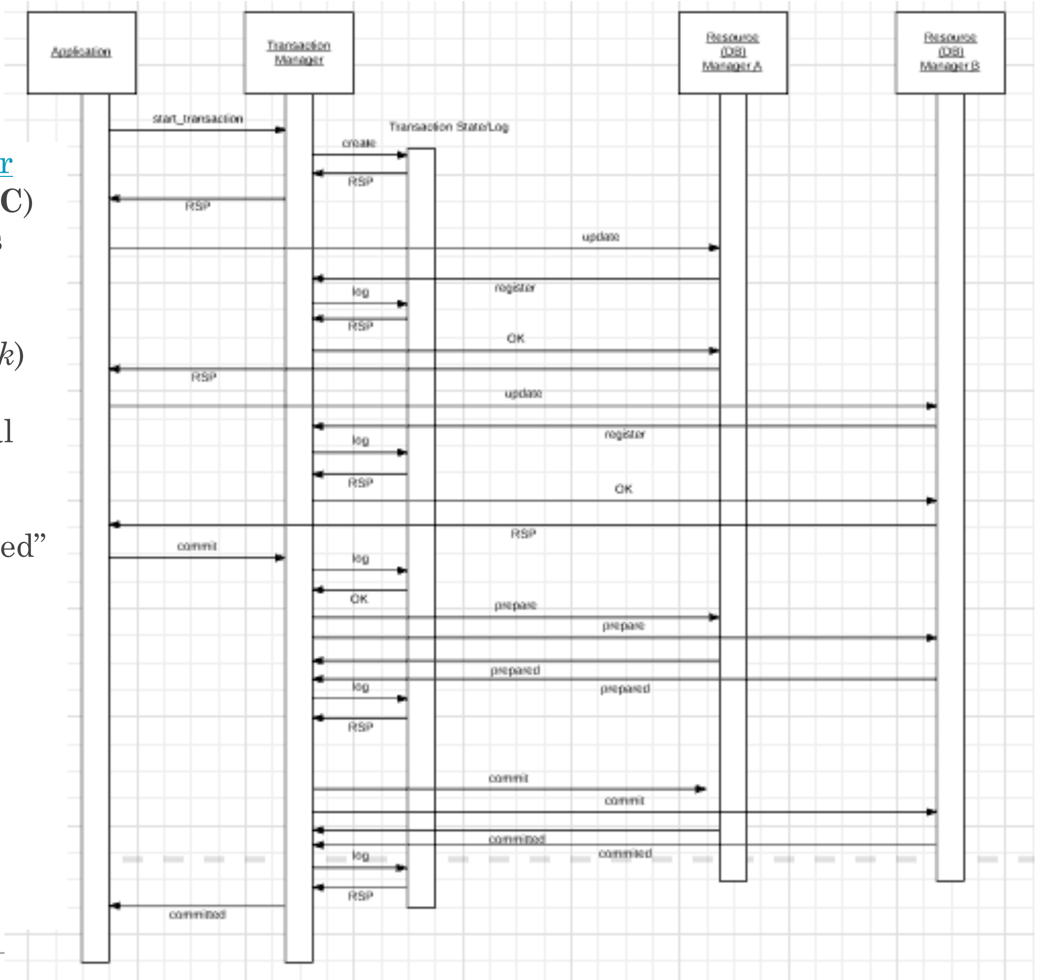
Two Phase Commit

"In transaction processing, databases, and computer networking, the **two-phase commit protocol** (2PC) is a type of atomic commitment protocol (ACP). It is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction on whether to *commit* or *abort* (*roll back*) the transaction (it is a specialized type of consensus protocol). The protocol achieves its goal even in many cases of temporary system failure (involving either process, network node, communication, etc. failures), and is thus widely used" (www.wikipedia.org)

Note: This sequence diagram omits some steps.

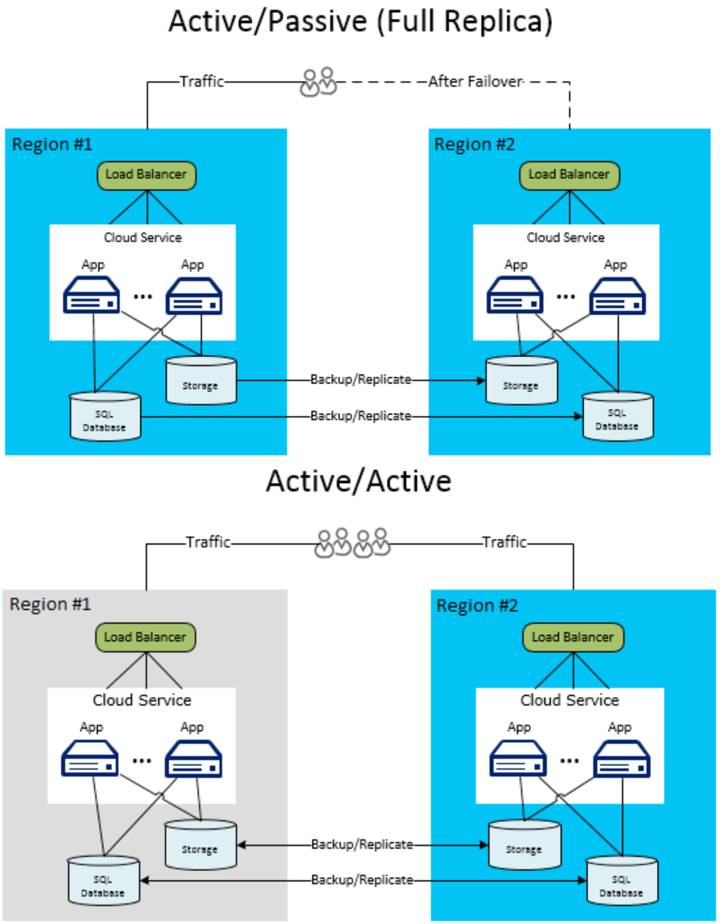
Uses:

- ACID, especially consistency, for data in different databases/services.
- Modified approach is required to strictly achieve consistency and availability for replicated data,



Availability and Replication

- There are two basic patterns
  - Active/Passive
    - All requests go to *master* during normal processing.
    - Updates are transactionally queued for processing at passive backup.
    - Failure of *master*
      - Routes subsequent requests to *backup*.
      - Backup must process and commit updates before accepting requests.
  - Active/Active
    - Both environments process requests.
    - Some form of distributed transaction commit required to synchronize updates on both copies.
- Multi-system communication to guarantee consistency is the foundation for tradeoffs in CAP.
  - The system can be CAP if and only iff
  - There are never any partitions or system failures
  - Which is unrealistic in cloud/Internet systems.



Core Data Models

(http://blog.nahurst.com/visual-guide-to-nosql-systems)

In addition to CAP configurations, another significant way data management systems vary is by the data model they use: relational, key-value, column-oriented, or document-oriented (there are others, but these are the main ones).

- **Relational** systems are the databases we've been using for a while now. RDBMSs and systems that support ACIDity and joins are considered relational.
- **Key-value** systems basically support get, put, and delete operations based on a primary key.
- **Column-oriented** systems still use tables but have no joins (joins must be handled within your application). Obviously, they store data by column as opposed to traditional row-oriented databases. This makes aggregations much easier.
- **Document-oriented** systems store structured "documents" such as JSON or XML but have no joins (joins must be handled within your application). It's very easy to map data from object-oriented software to these systems.

Core Data Models

(http://blog.nahurst.com/visual-guide-to-nosql-systems)

In addition to CAP configurations, another significant way data management systems vary i
orien

There are also graph databases, which we will cover.

– **Document-oriented** systems store structured "documents" such as JSON or XML but have no joins (joins must be handled within your application). It's very easy to map data from object-oriented software to these systems.

Database Models

Available, Partition-Tolerant (AP) Systems achieve "eventual consistency" through replication and verification. Examples of AP systems include:

- DynamoDB (key-value)
- Voldemort (key-value)
- Tokyo Cabinet (key-value)
- KAI (key-value)
- Cassandra (column-oriented/tabular)
- CouchDB (document-oriented)
- SimpleDB (document-oriented)
- Riak (document-oriented)

Consistent, Available (CA) Systems have trouble with partitions and typically deal with it with replication. Examples of CA systems include:

- Traditional RDBMSs like Postgres, MySQL, (relational)
- Vertica (column-oriented)
- Aster Data (relational)
- Greenplum (relational)

Consistent, Partition-Tolerant (CP) Systems have trouble with availability while keeping data consistent across partitioned nodes. Examples of CP systems include:

- BigTable (column-oriented/tabular)
- Hypertable (column-oriented/tabular)
- HBase (column-oriented/tabular)
- MongoDB (document-oriented)
- Terrastore (document-oriented)
- Redis (key-value)
- Scalaris (key-value)
- MemcacheDB (key-value)
- Berkeley DB (key-value)

http://blog.nahurst.com/visual-guide-to-nosql-systems

Consistent, Available (CA) Systems have trouble with

Data

Availa ... al)

"eventu

verifica

- Dyn
- Vol
- Tok
- KA
- Cas
- Cou
- Sim
- Rial

le

- Key-Value versus Document can be a little confusing.
- Many, if not most, Key-Value stores hold documents.
- The difference is
  - Not in what the data store holds
  - But how you can query and find it.
- Key-Value store finds data by key values explicitly associated with a document.
- Document stores allow queries on the fields/values in the document, similar to a relational WHERE on columns.

http://blog.nahurst.com/visual-guide-to-nosql-systems

Some Queries

- MongoDB Query
  - db.inventory.find({   status: "A", $or: [{ qty: { $lt: 30 } }, { item: /^p/ }]})
  - Finds all documents in the collection wherthe status equals "A" **and** *either* qty is less than ($lt) 30 *or* item starts with the character ”p”

- Redis
  - SADD product:category:1 5 10 // Put product IDs 5 and 10 in category 1.
  - SMEMBERS product:category:2 // Get the IDs of all products in category 2.

- DynamoDB
  - **var** params =
    { TableName : "Movies",
    KeyConditionExpression: "#yr = :yyyy",
    ExpressionAttributeNames:{ "#yr": "year" },
    ExpressionAttributeValues: { ":yyyy":1985 }
    };

Dynamo DB

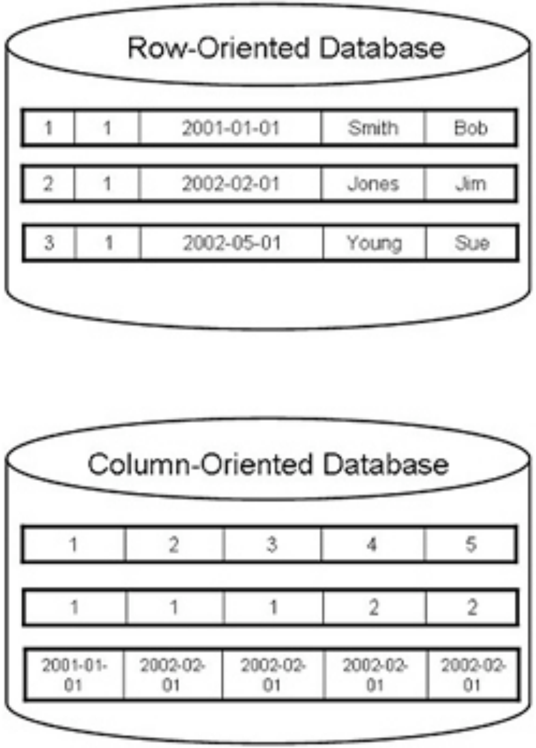
- Queries on keys
- Scans on values

Columnar (Relational) Database

(https://www.dbbest.com/blog/column-oriented-database-technologies/)

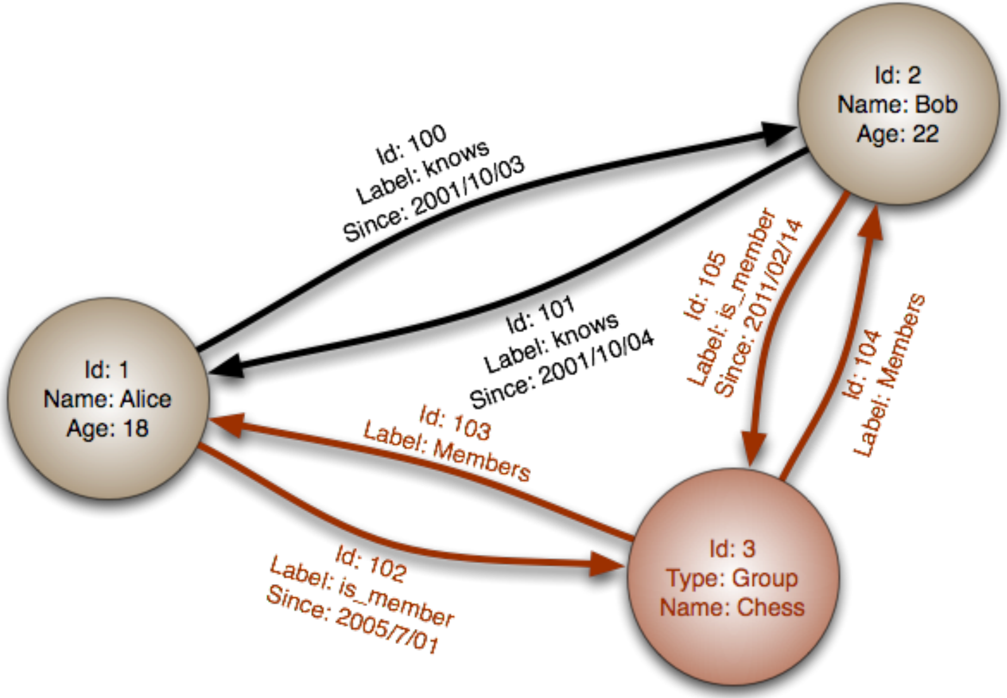
- Columnar and Row are both
  - Relational
  - Support SQL operations
- But differ in data storage
  - Row keeps row data together in blocks.
  - Columnar keys column data together in blocks.
- This determines performance for different types of query, e.g.
  - Columnar is extremely powerful for BI scenarios
    - Aggregation ops, e.g. SUM, AVG
    - PROJECT (do not load all of the row) to get a few columns
  - Row is powerful for OLTP. Transaction typically create and retrieve
    - One row at a time
    - All the columns of a single row.

Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura

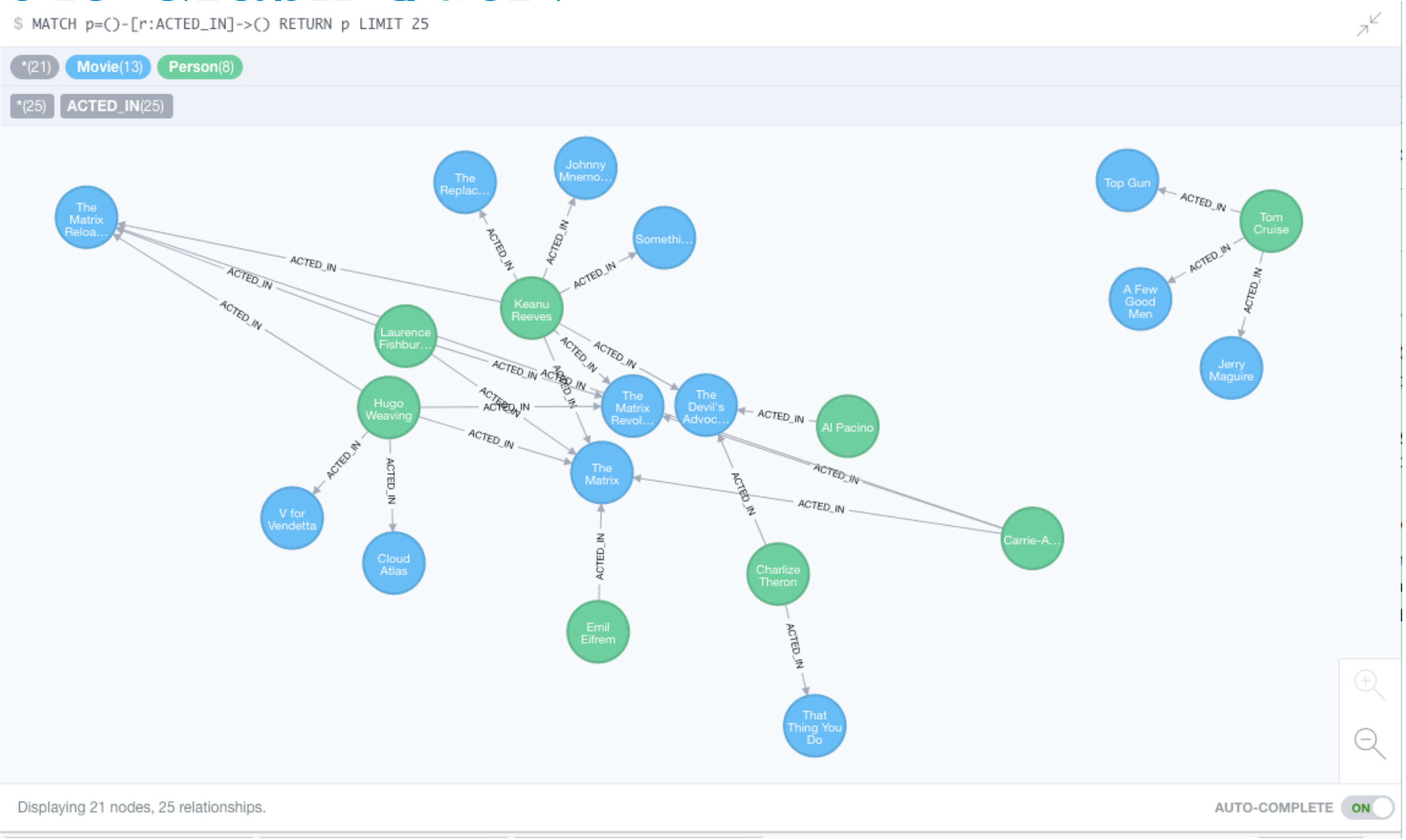


Graph Database

- Exactly what it sounds like
- Two core types
  - Node
  - Edge (link)
- Nodes and Edges have
  - Label(s) = "Kind"
  - Properties (free form)
- Query is of the form
  - p1(n)-p2(e)-p3(m)
  - n, m are nodes; e is an edge
  - p1, p2, p3 are predicates on labels



Neo4J Graph Query



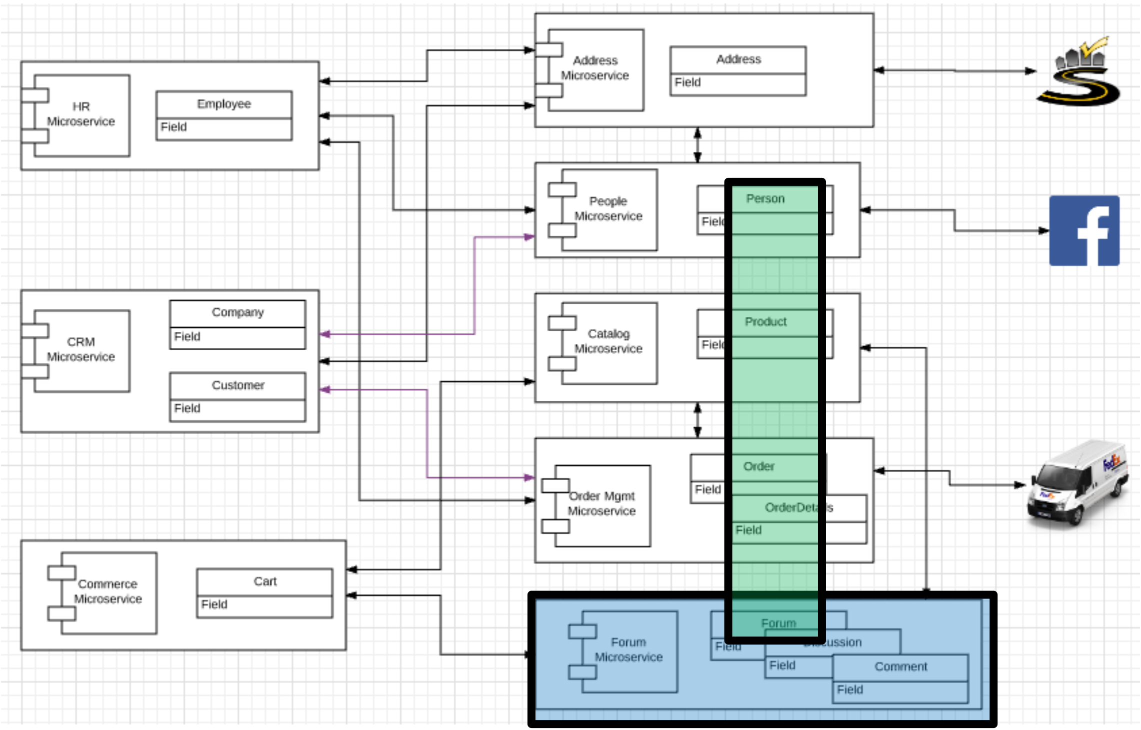
Thanks, but Why?

The initial idea was

- DynamoDB for the forum
- Neo4J to track/query
  - Who bought what?
  - Who has bought things similar to whom?
  - Who commented on what?
  - etc.
- Use Redis to optimize
  - Idempotency
  - Etag

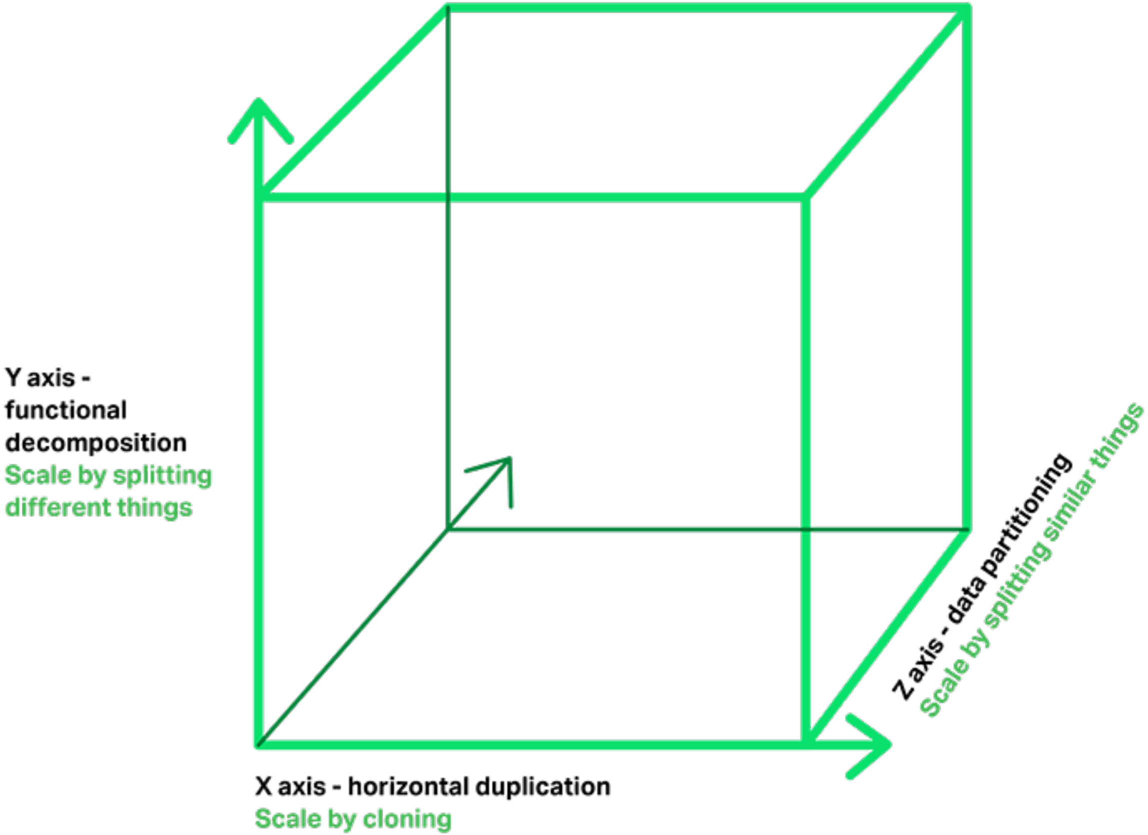
But

- we do not have enough time to do in context of solution.
- Will have to do smaller scenarios and use cases.



Microservice XYZ Scaling

Scaling Microservices



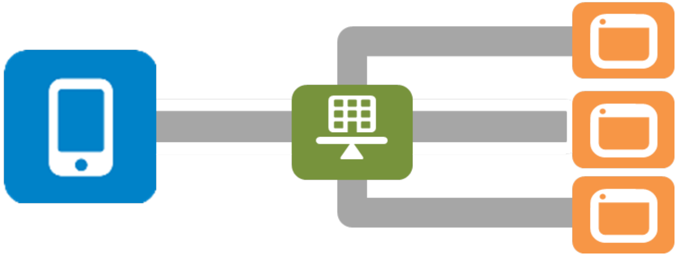
Microservice Scaling

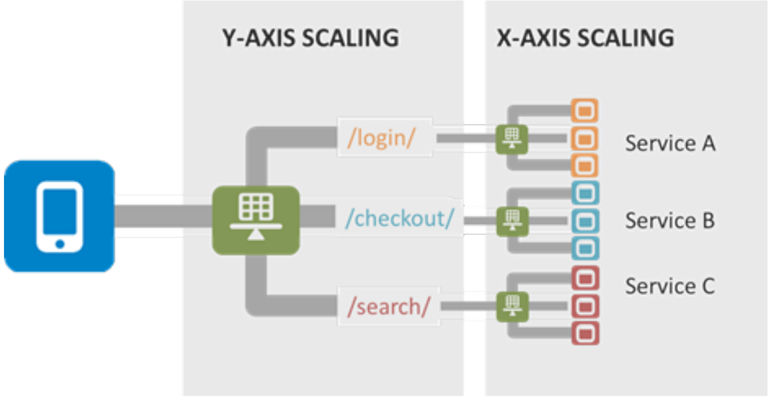
Y-AXIS SCALING

Network name: Layer 7 Load Balancing, Content switching, HTTP Message Steering

X-AXIS SCALING

Network name: Horizontal scaling, scale out





Z-AXIS SCALING

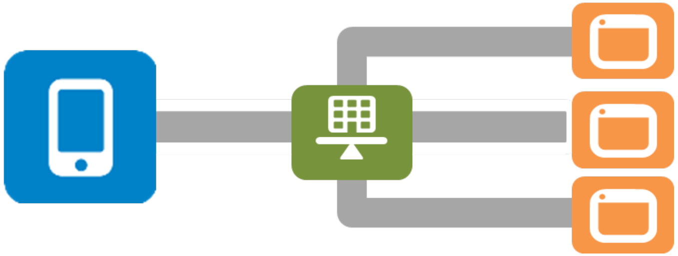
Network name: Layer 7 Load Balancing, Content switching, HTTP Message Steering

https://devcentral.f5.com/articles/the-art-of-scale-microservices-the-scale-cube-and-load-balancing

Microservice Scaling

X-AXIS SCALING

Network name: Horizontal scaling, scale out



X-Axis (Horizontal Scaling) assumes

- Each instance has equal access to data ➔
- Highly scalable database in some scenarios.

And is the assumed model for

- Lambda functions
- Elastic BeanStalk

This is hard to achieve for scenarios like

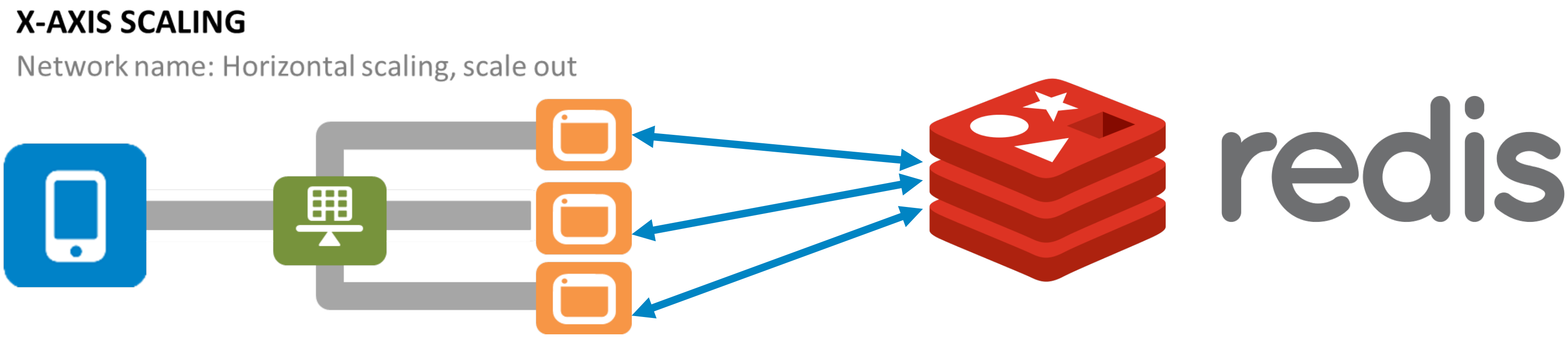
- ETag
- Idempotentcy tokens

CALING

name: Layer 7 Load Balancing, Content switching, HTTP Message Steering

https://devcentral.f5.com/articles/the-art-of-scale-microservices-the-scale-cube-and-load-balancing

Redis (Memcache), ETag, Idempotency



- ETag
  - Store {URL, ETag} on GET cache miss.
  - Invalidate tag on any PUT, POST
  - Fail safe: If any doubt, e.g. partition, error, …, treat as a conflict.
- Idempotency
  - Check for duplicate, and create if not exists.
  - Fail safe: do not process a request if any doubt.

Similar to express-redis-cache
(https://www.npmjs.com/package/express-redis-cache)

Just use it as a middleware in the stack of the route you want to cache.

```
var app = express();
var cache = require('express-redis-cache')();

// replace
app.get('/',
  function (req, res)  { ... });

// by
app.get('/',
  cache.route(),
  function (req, res)  { ... });
```

This will check if there is a cache entry for this route. If not. it will cache it and serve the cache next time route is called.

12 Factor Applications

https://12factor.net/

https://www.slideshare.net/bifer/twelve-factor-apps

The 12 Factors

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing Services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. LogsTreat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

- See PDF