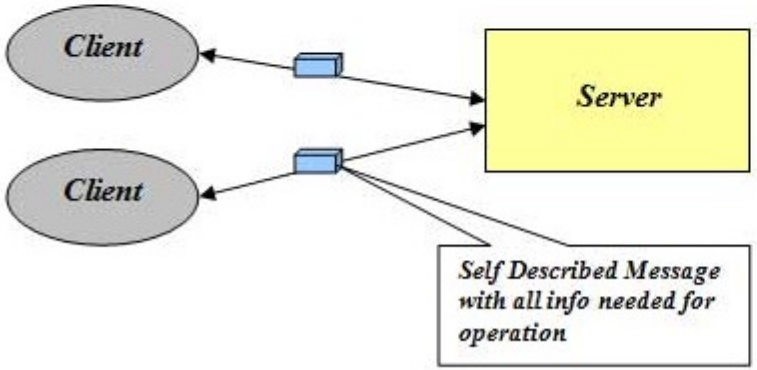
Topics in SW Engineering – Cloud Computing: Some Patterns and Concepts

Stateless, Sessions Example

Stateless

The notion of statelessness is defined from the perspective of the server. The constraint says that the server should not remember the state of the application. As a consequence, the client should send all information necessary for execution along with each request, because the server cannot reuse information from previous requests as it didn't memorize them. All info needed is in message.



Statelessness

2. Application State vs Resource State

It is important to understand the between the application state and the resource state. Both are completely different things.

Application state is server-side data that servers store to identify incoming client requests, their previous interaction details, and current context information.

Resource state is the current state of a resource on a server at any point in time – and it has nothing to do with the interaction between client and server. It is what we get as a response from the server as the API response. We refer to it as resource representation.

> REST statelessness means being free from the application state.

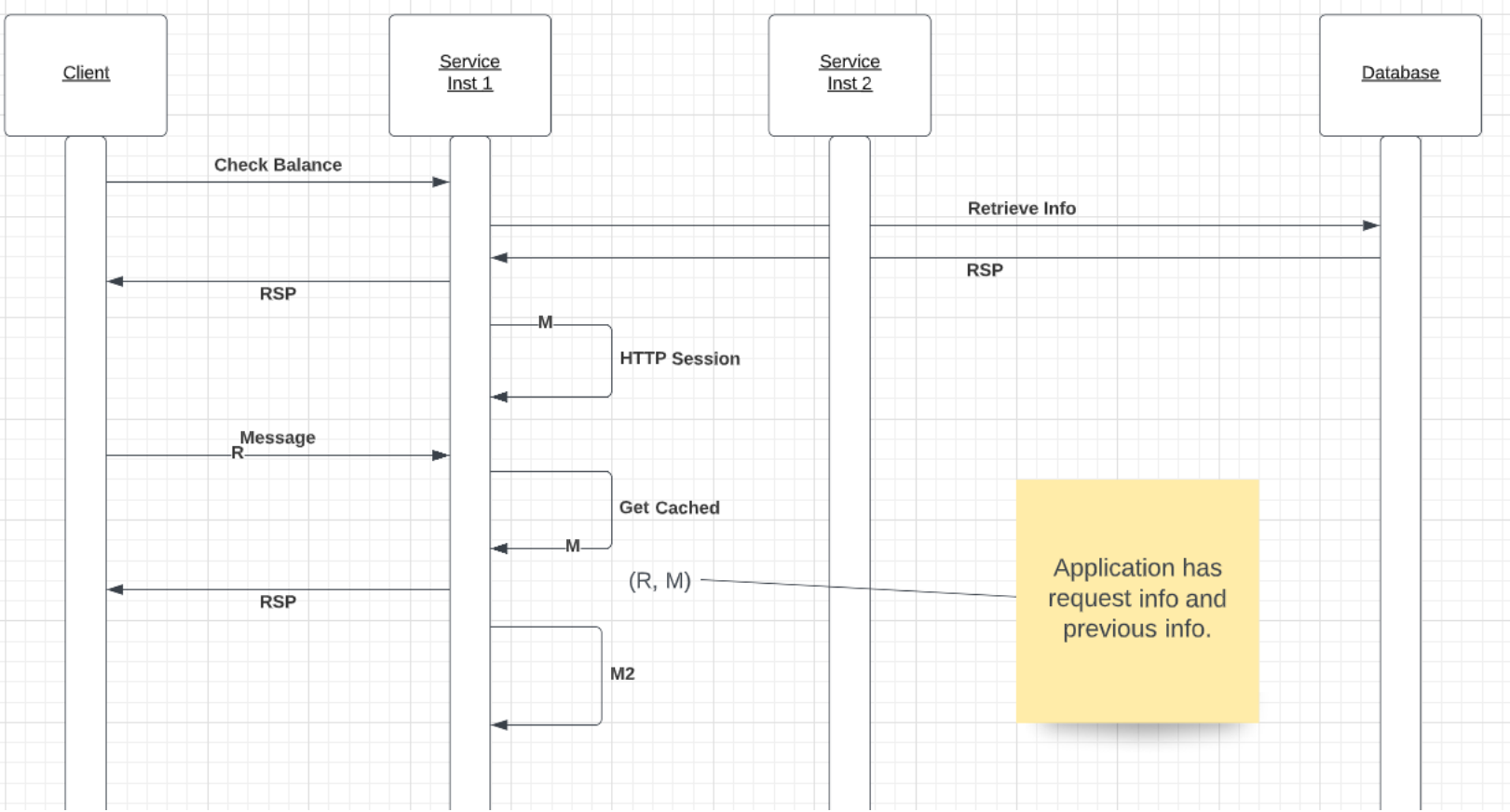
3. Advantages of Stateless APIs

There are some very noticeable advantages of having **REST APIs stateless**.

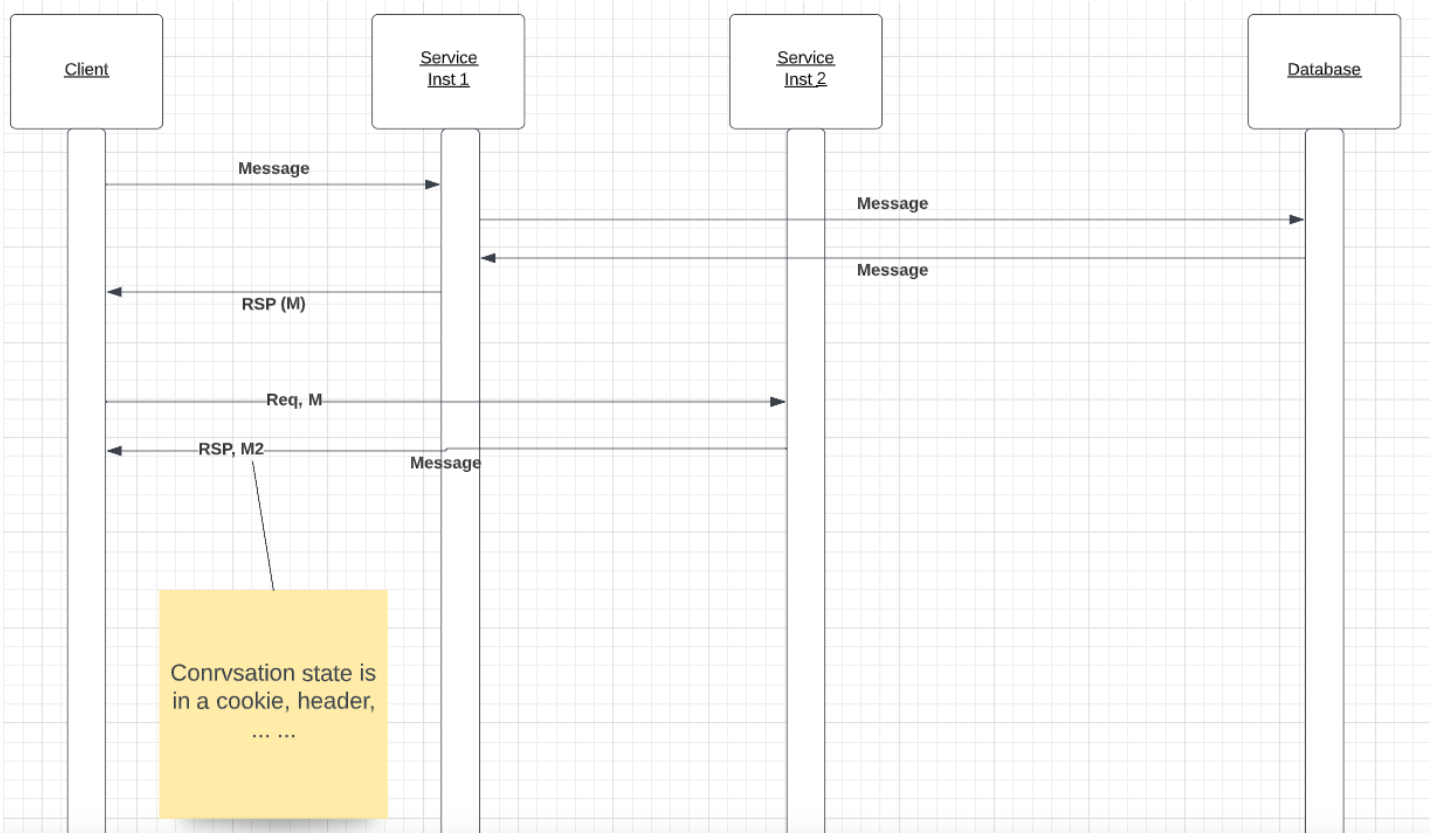
1. Statelessness helps in **scaling the APIs to millions of concurrent users** by deploying it to multiple servers. Any server can handle any request because there is no session related dependency.
2. Being stateless makes REST APIs **less complex** – by removing all server-side state synchronization logic.
3. A stateless API is also **easy to cache** as well. Specific softwares can decide whether or not to cache the result of an HTTP request just by looking at that one request. There's no nagging uncertainty that state from a previous request might affect the cacheability of this one. It **improves the performance** of applications.
4. The server never loses track of "where" each client is in the application because the client sends all necessary information with each request.

- This concept can be confusing with experience with other approach to session state.
- Some application frameworks did/do keep session state on the server.
- Application servers and frameworks have various support for "session state," e.g. https://flask-session.readthedocs.io/en/latest/

Stateful Conversation – Simple Example

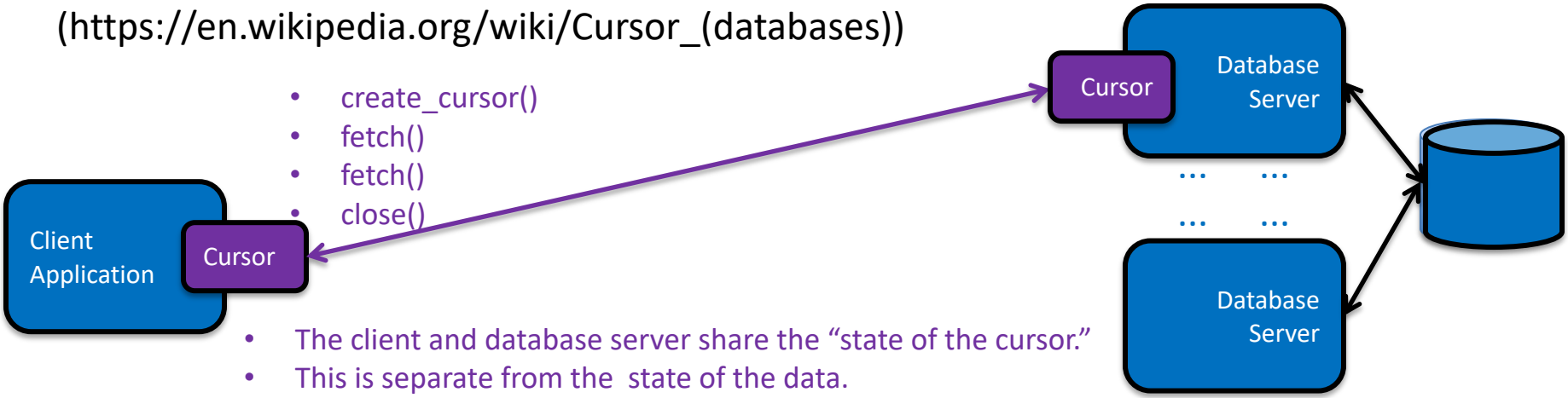


Stateless – Simple Example



Simple Example from Databases

- "In computer science, a database cursor is a mechanism that enables traversal over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records. The database cursor characteristic of traversal makes cursors akin to the programming language concept of iterator."
(https://en.wikipedia.org/wiki/Cursor_(databases))



- The client and database server share the "state of the cursor."
- This is separate from the state of the data.
- This hinders scalability and fault tolerance if there are multiple database servers in a cluster.

Some Code

```python
def t_cursor():

    print("\nUsing cursors ...")
    sql = "select * from users";

    for i in range(0,5):
        if i == 0:
            res = with_cursor(done=False, sql=sql)
        elif i == 4:
            res = with_cursor(done=True)
        else:
            res = with_cursor()

        print("Res = ", res)
```

```python
def t_cursor():

    print("\nUsing cursors ...")
    sql = "select * from users";

    for i in range(0,5):
        if i == 0:
            res = with_cursor(done=False, sql=sql)
        elif i == 4:
            res = with_cursor(done=True)
        else:
            res = with_cursor()

        print("Res = ", res)
```

```python
def t_without_cursor():

    print("\nNot using cursors.")

    sql = "select * from users";

    for i in range(0,5):
        res = without_cursor(sql=sql, offset=i)
        print("Res = ", res)
```

```python
def without_cursor(sql, offset):

    cursor = conn.cursor()
    tmp_sql = sql + " limit 1" + " offset " + str(offset)
    res = cursor.execute(tmp_sql)
    res = cursor.fetchone()
    cursor.close()
    return res
```

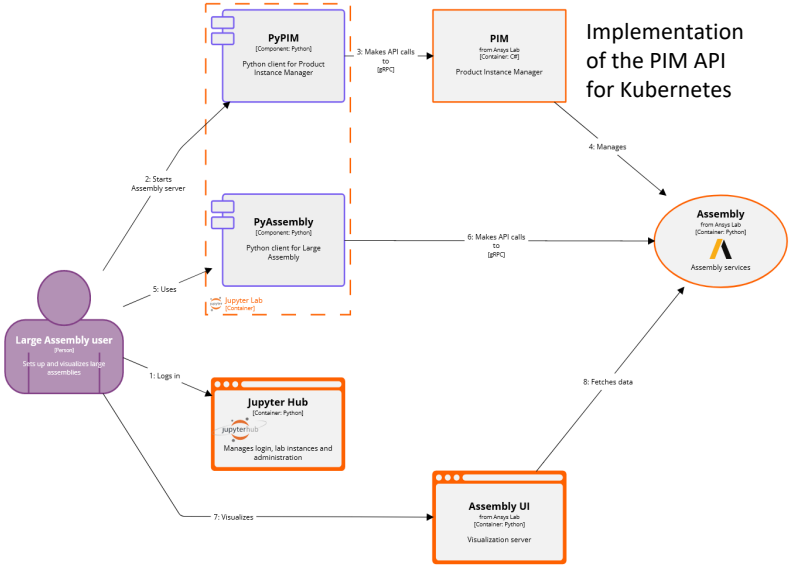
Summary

- This session explained the concept of “stateless” in REST.
- Most of the cloud deployment of our existing products will be stateful and have conversation state in memory.
  - That is fine. There are design patterns for handling the situation.
  - Our infrastructure will support “affinity.”
    - User affinity.
    - Session affinity.
  - Clients get “pinned” to a microservice instances based on affinity.
- Ansys Lab and the Product Instance Manager support this model.

Product Instance Management (PIM API & PyPIM)

Simple config-based creation, deletion, tracking of product aas

```
assembly = pim.create_instance(product_name="assembly", product_version="latest")
```



Implementation of the PIM API for Kubernetes

"Assembly" PIM config = Kubernetes yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  generateName: instances-assembly-latest-
  labels:
    instancemanagement.ansys.com/definition-name: assembly-latest
    instancemanagement.ansys.com/product-name: assembly
    instancemanagement.ansys.com/product-version: latest
  annotations:
    instancemanagement.ansys.com/services: |
      grpc:
        address: dns:{ip}:{port}
        portName: grpc
        headers: {}
      fileservice:
        address: dns:{ip}:{port}
        portName: fileservice
        headers: {}
spec:
  containers:
    - name: assembly
      image: azwepsifujiaksacr.azurecr.io/ansys/ansys-ch/assembly-management/large-model:latest
      imagePullPolicy: Always
      env: []
      args: []
      ports:
        - containerPort: 50053
          name: grpc
          protocol: TCP
        - containerPort: 2048
          name: fileservice
          protocol: TCP
      resources:
        limits:
          cpu: "2"
          memory: 64Gi
        requests:
          cpu: "2"
          memory: 2Gi
  runtimeClassName: linux-simulation
```

Specs driven by the dev team

Specs driven by Fuji's infra

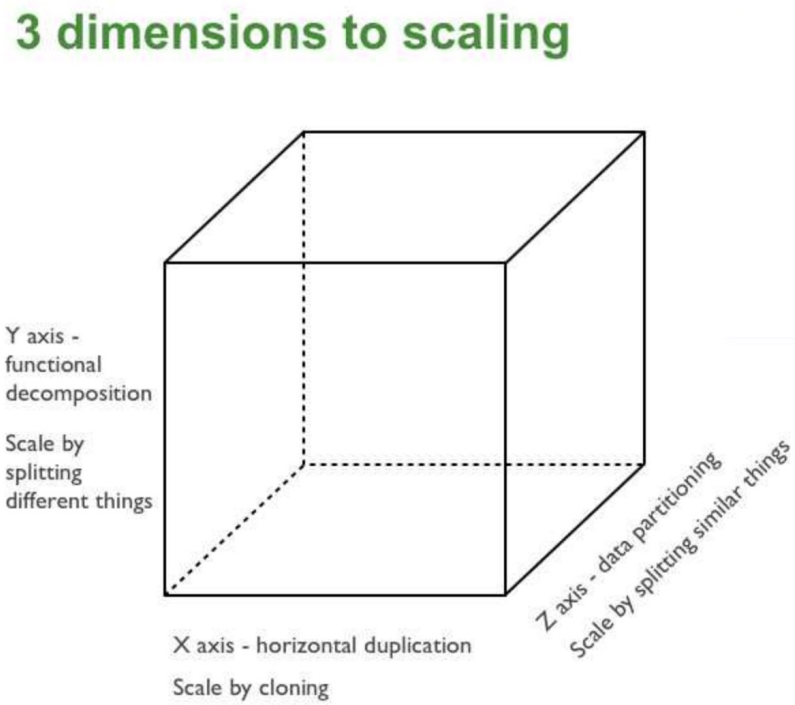
XYZ Scaling

Microservices Scalability Cube

The Scale Cube

https://microservices.io/articles/scalecube.html

The book, The Art of Scalability, describes a really useful, three dimension scalability model: the scale cube.



- X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application.
- Y-axis axis scaling splits the application into multiple, different services. Each service is responsible for one or more closely related functions.
- When using Z-axis scaling each server runs an identical copy of the code. In this respect, it's similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server

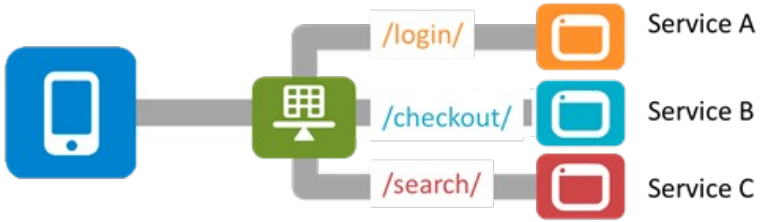
XYZ – Scaling

X-AXIS SCALING

Network name: Horizontal scaling, scale out

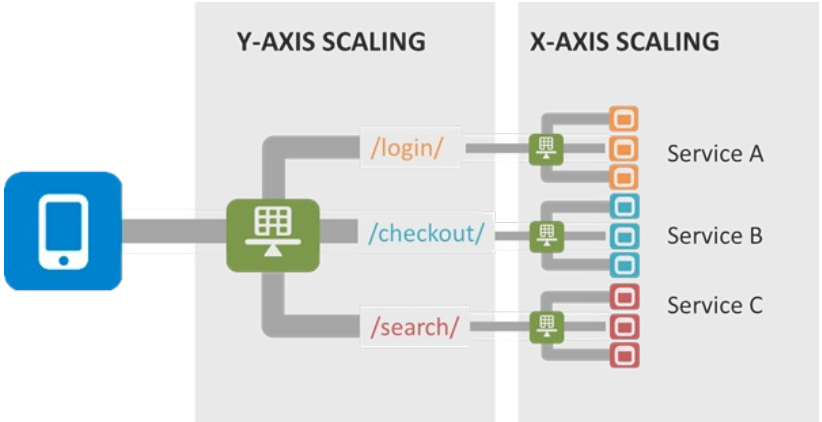
Y-AXIS SCALING

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



Z-AXIS SCALING

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

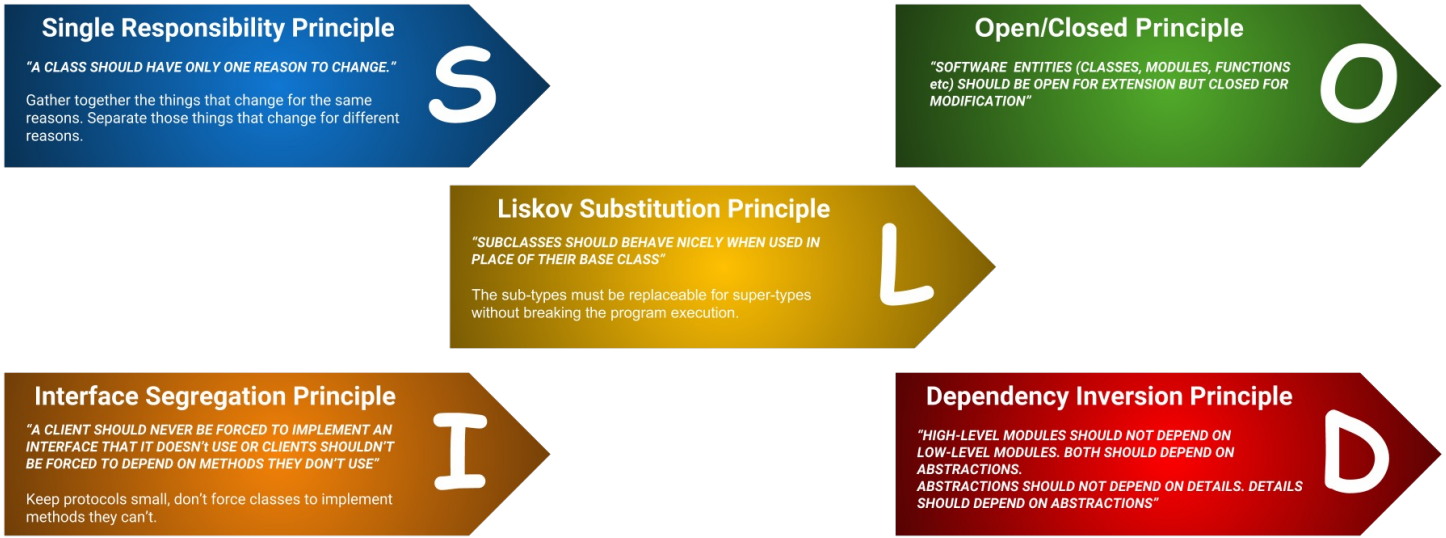


- There are three dimensions.
- A complete solution/environment typical is a mix and composition of patterns.
- Application design and data access determines opitions.

SOLID

SOLID Principles (Enumerate, Will Cover Later)

- "In software engineering, SOLID is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible, and maintainable. " (https://en.wikipedia.org/wiki/SOLID)
- The concept started with OO but applies to microservices.



12 Factor Applications

12 Factor Applications

https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser

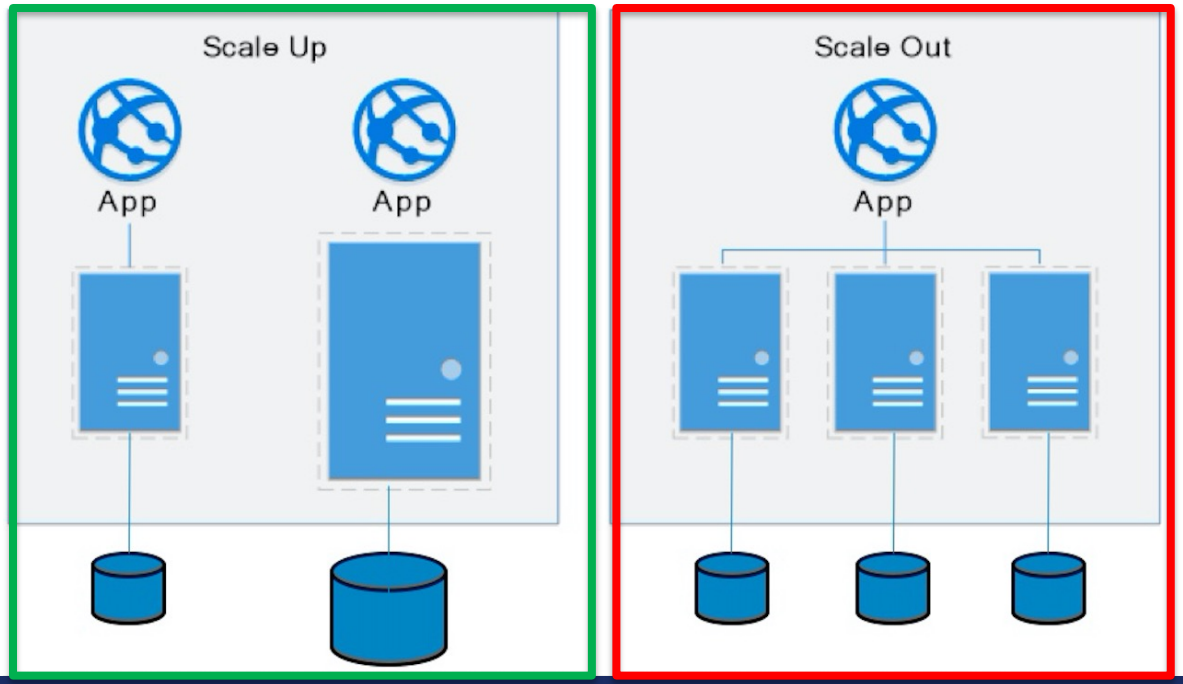
Scaling

Approaches to Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system.

Replace system with a bigger machine,
e.g. more memory, CPU, … …

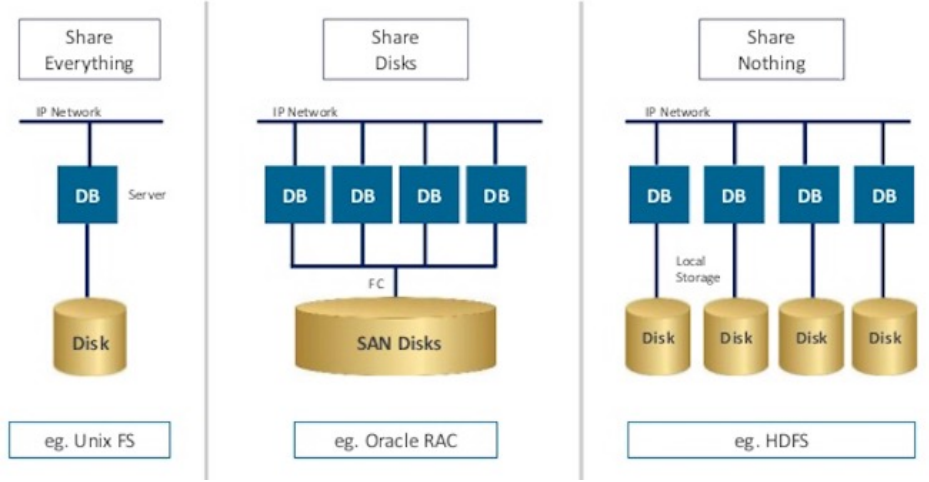
Add another system.



- Scale-up:
  - Less incremental.
  - More disruptive.
  - More expensive for extremely large systems.
  - Does not improve availability
- Scale-out:
  - Incremental cost.
  - Data replication enables availability.
  - Does not work well for functions like JOIN, referential integrity, … …

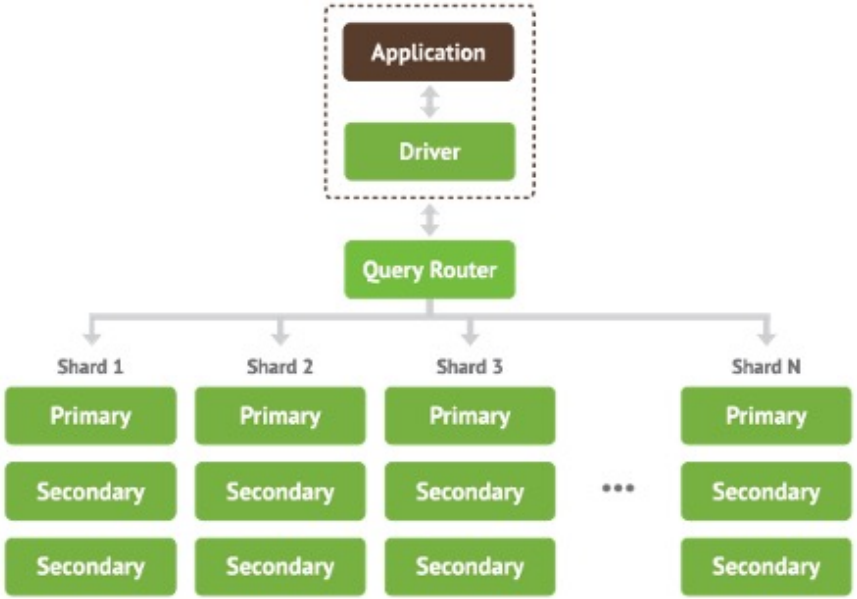
Disk Architecture for Scale-Out

- Share disks:
  - Is basically scale-up for data/disks. You can use NAS, SAN and RAID.
  - Isolation/Integrity requires distributed locking to control access from multiple database servers.
- Share nothing:
  - Is basically scale-out for disks.
  - Data is partitioned into *shards* based on a function f() applied to a key.
  - Can improve availability, at the code consistency, with data replication.
  - There is a router that sends requests to the proper shard based on the function.

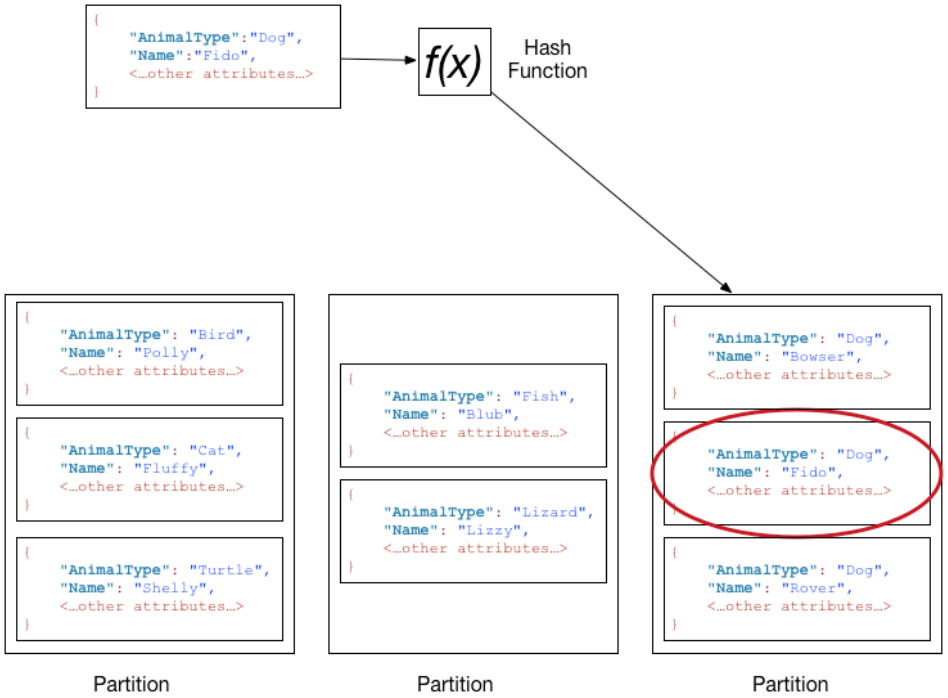


Shared Nothing, Scale-Out

MongoDB Sharding

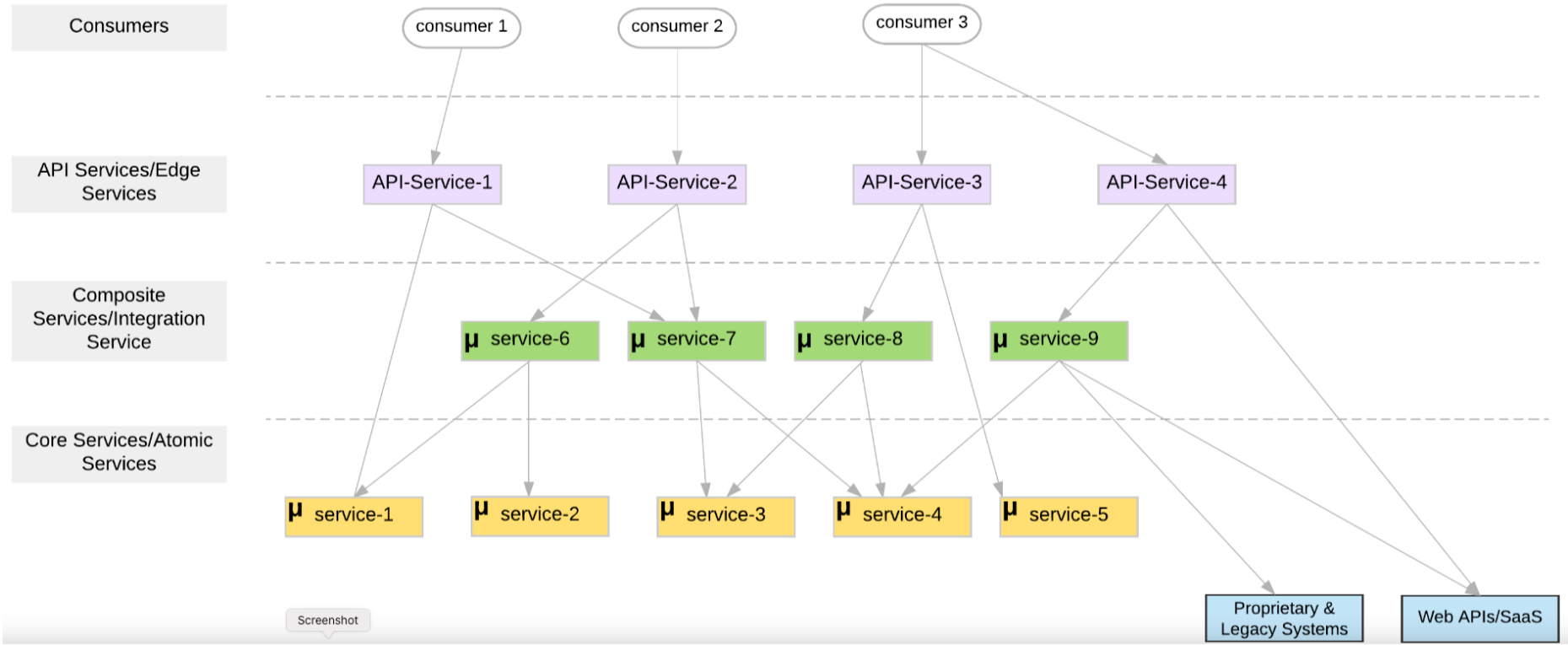


DynamoDB Partitioning

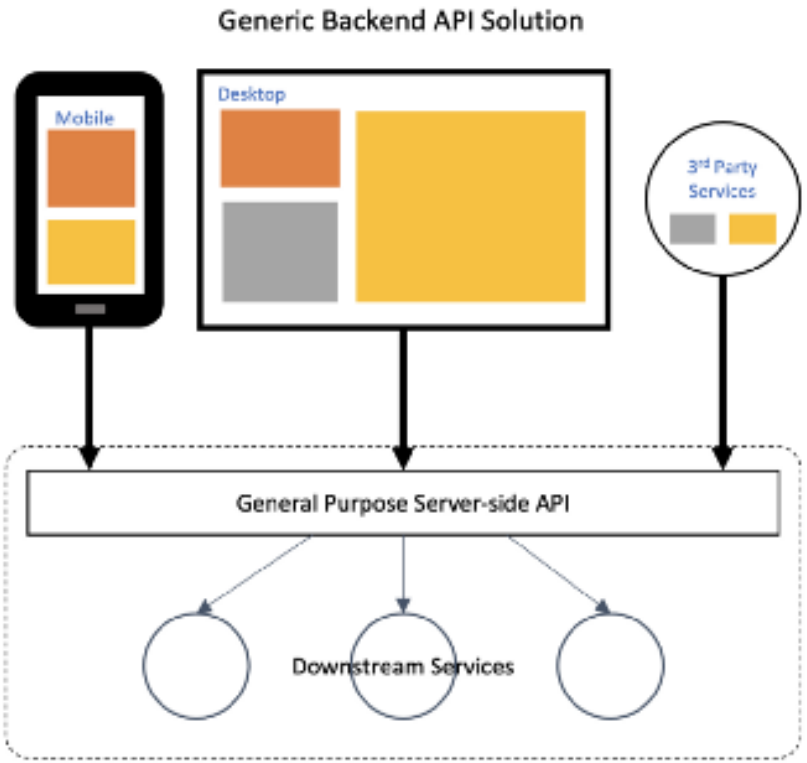


Microservice Layers

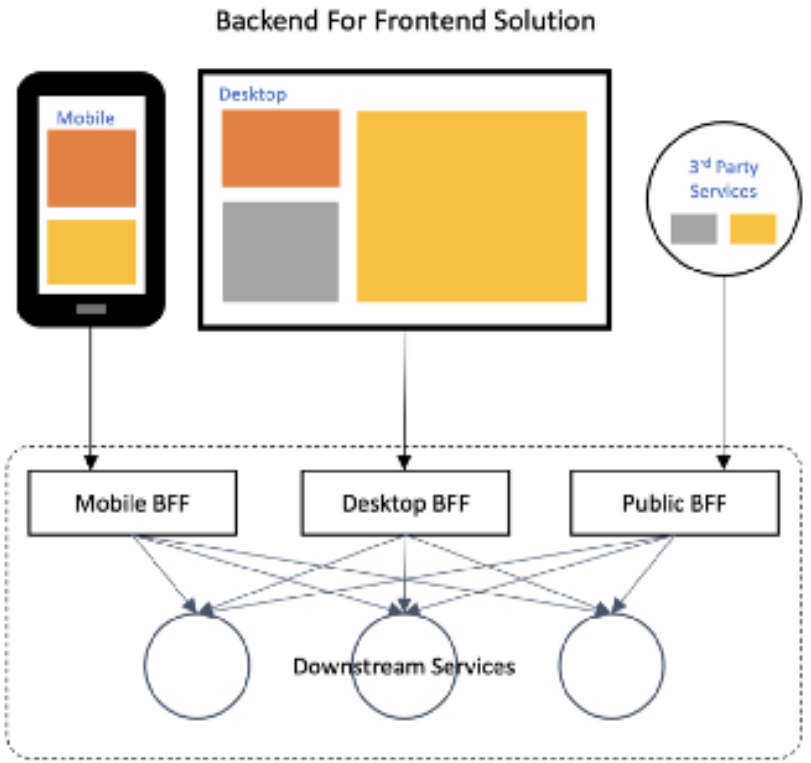
Microservice Layers



Backend for Frontend



General-purpose API backend takes multiple responsibilities.
Usually complex and not easy to maintain solution.



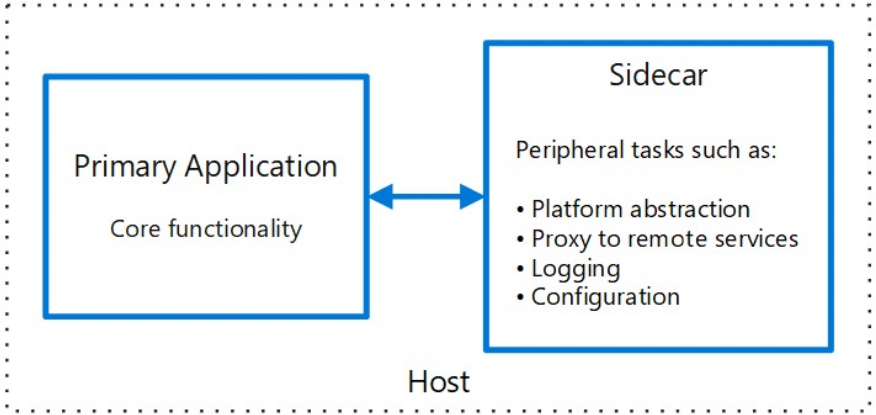
Layer between the user experience and the resources it calls on.
Provides dedicated and optimised solution per each frontend client.

Architecture Patterns

Sidecar Pattern

Solution

Co-locate a cohesive set of tasks with the primary application, but place them inside their own process or container, providing a homogeneous interface for platform services across languages.

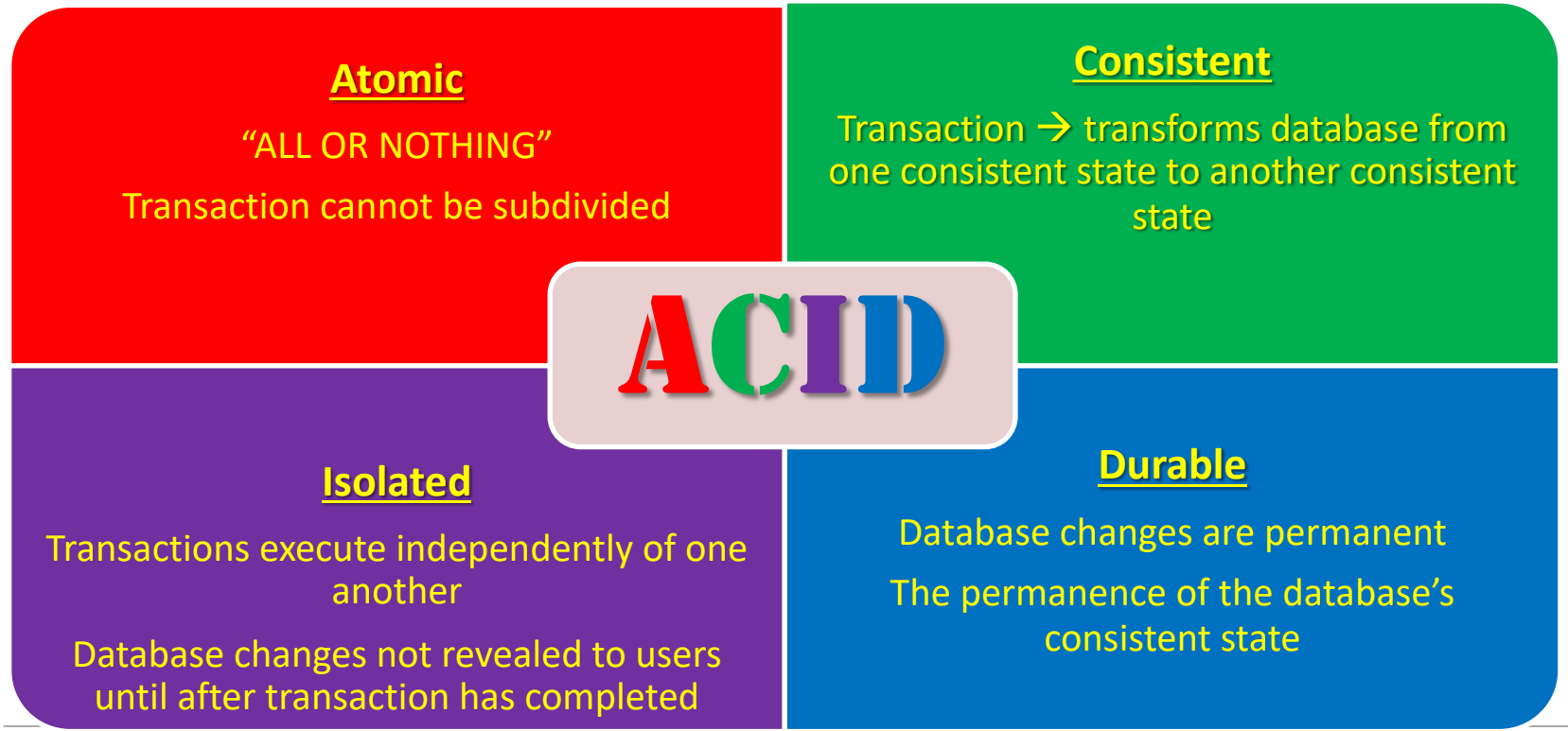


- A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language.
- The sidecar can access the same resources as the primary application. For example, a sidecar can monitor system resources used by both the sidecar and the primary application.
- Because of its proximity to the primary application, there's no significant latency when communicating between them.
- Even for applications that don't provide an extensibility mechanism, you can use a sidecar to extend functionality by attaching it as its own process in the same host or sub-container as the primary application.

NoSQL
DynamoDB

Core Transaction Concept is ACID Properties

http://slideplayer.com/slide/9307681



ACID Properties (httpwww.tutorialspoint.com/dbms/dbms_transaction.htm)

A *transaction* is a very small unit of a program and it may contain several low-level tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability – commonly known as ACID properties – in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** – This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** – The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** – The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** – In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Serializability

"In concurrency control of databases, transaction processing (transaction management), and various transactional applications (e.g., transactional memory[3] and software transactional memory), both centralized and distributed, a transaction schedule is **serializable** if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems." (https://en.wikipedia.org/wiki/Serializability)

CAP Theorem

- Consistency

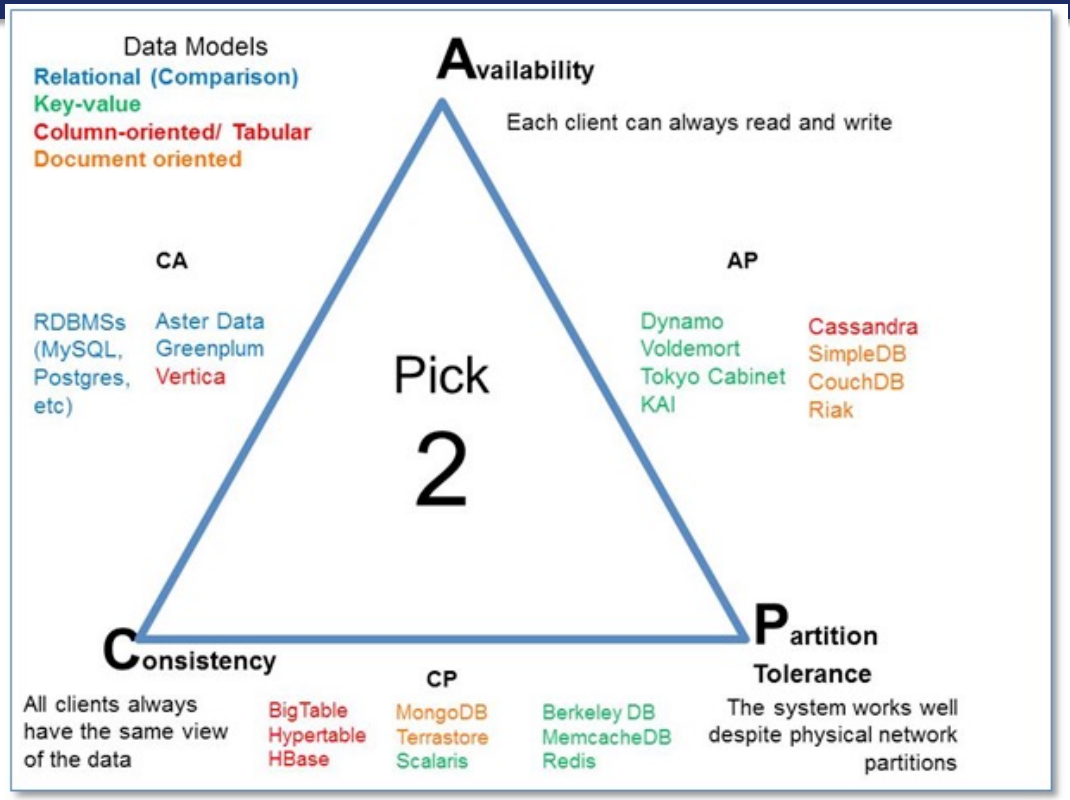
  Every read receives the most recent write or an error.

- 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



Consistency Models

- **STRONG CONSISTENCY:** Strong consistency is a consistency model where all subsequent accesses to a distributed system will always return the updated value after the update.
- **WEAK CONSISTENCY:** It is a consistency model used in distributed computing where subsequent accesses might not always be returning the updated value. There might be inconsistent responses.
- **EVENTUAL CONSISTENCY:** Eventual consistency is a special type of weak consistency method which informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

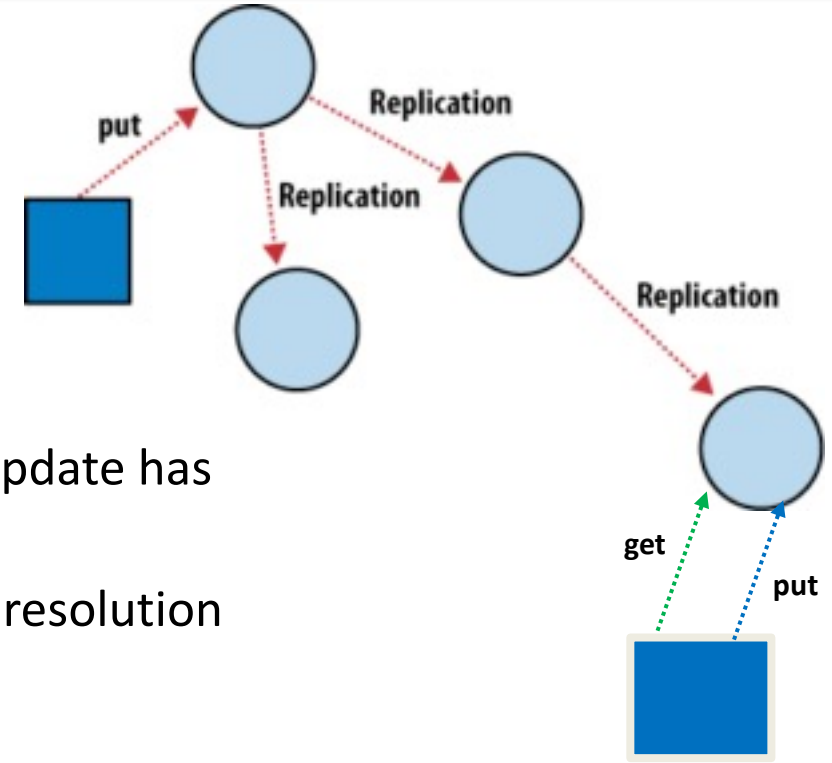
BASE

Eventually-consistent services are often classified as providing BASE (Basically Available, Soft state, Eventual consistency) semantics, in contrast to traditional ACID (Atomicity, Consistency, Isolation, Durability) guarantees. Rough definitions of each term in BASE:

- **Basically Available**: basic reading and writing operations are available as much as possible (using all nodes of a database cluster), but without any kind of consistency guarantees (the write may not persist after conflicts are reconciled, the read may not get the latest write)
- **Soft state**: without consistency guarantees, after some amount of time, we only have some probability of knowing the state, since it may not yet have converged
- **Eventually consistent**: If the system is functioning and we wait long enough after any given set of inputs, we will eventually be able to know what the state of the database is, and so any further reads will be consistent with our expectations

Eventual Consistency

- Availability and scalability via
  - Multiple, replicated data stores.
  - Read goes to “any” replica.
  - PUT/POST/DELETE
    - Goes to any replica
    - Change propagate asynchronously
- GET may not see the latest value if the update has not propagated to the replica.
- There are several algorithms for conflict resolution
  - Detect and handle in application.
  - Clock/change vectors/version numbers
  - … …



ACID – BASE (Simplistic Comparison)

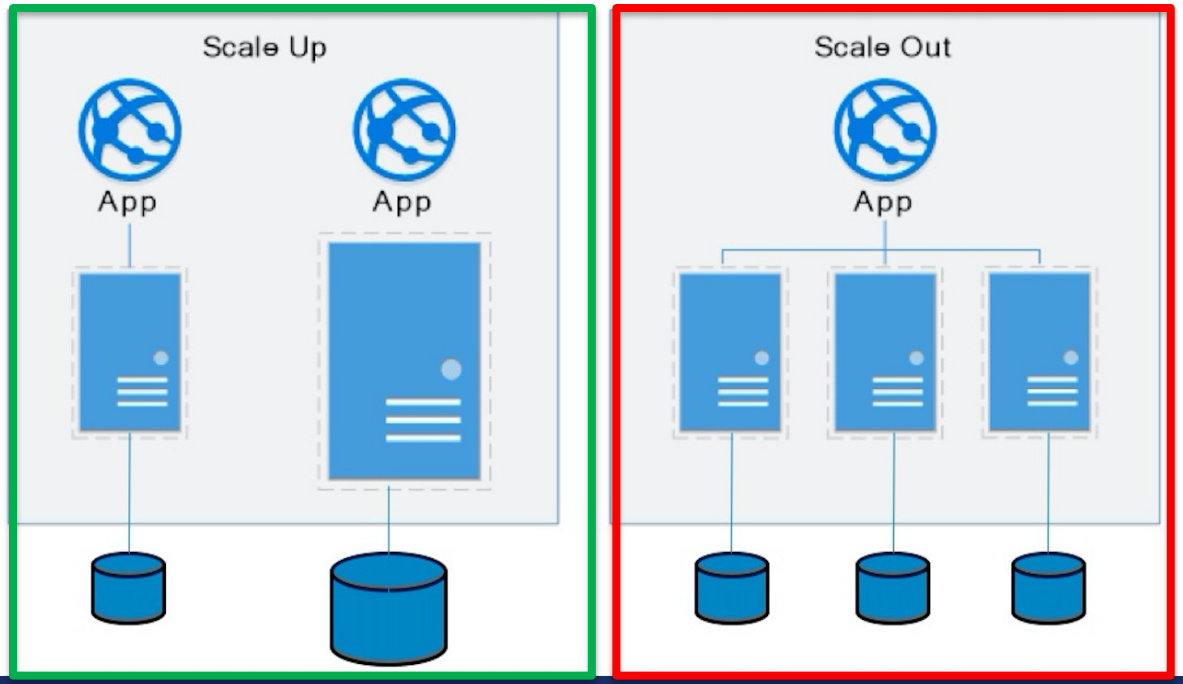
ACID (relational)	BASE (NoSQL)
Strong consistency	Weak consistency
Isolation	Last write wins (Or other strategy)
Transaction	Program managed
Robust database	Simple database
Simple code (SQL)	Complex code
Available and consistent	Available and partition-tolerant
Scale-up (limited)	Scale-out (unlimited)
Shared (disk, mem, proc etc.)	Nothing shared (parallellizable)

Approaches to Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system.

Replace system with a bigger machine,
e.g. more memory, CPU, … …

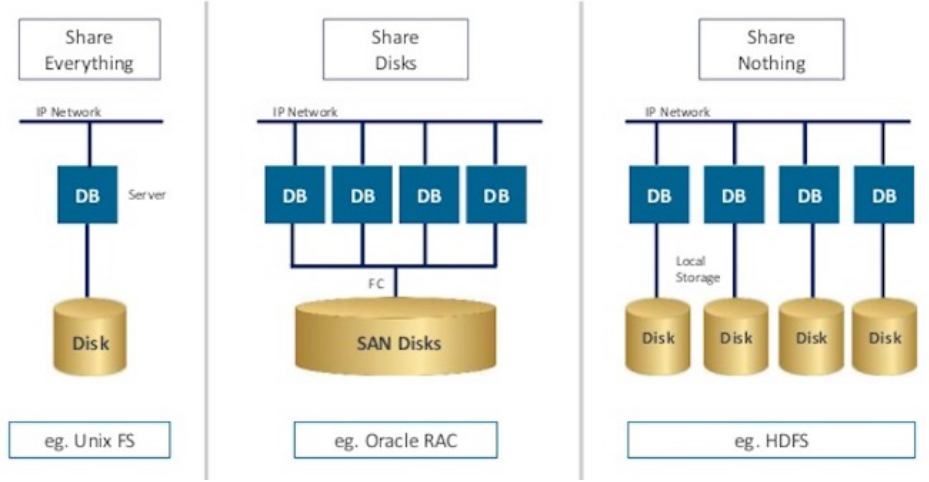
Add another system.



- Scale-up:
  - Less incremental.
  - More disruptive.
  - More expensive for extremely large systems.
  - Does not improve availability
- Scale-out:
  - Incremental cost.
  - Data replication enables availability.
  - Does not work well for functions like JOIN, referential integrity, … …

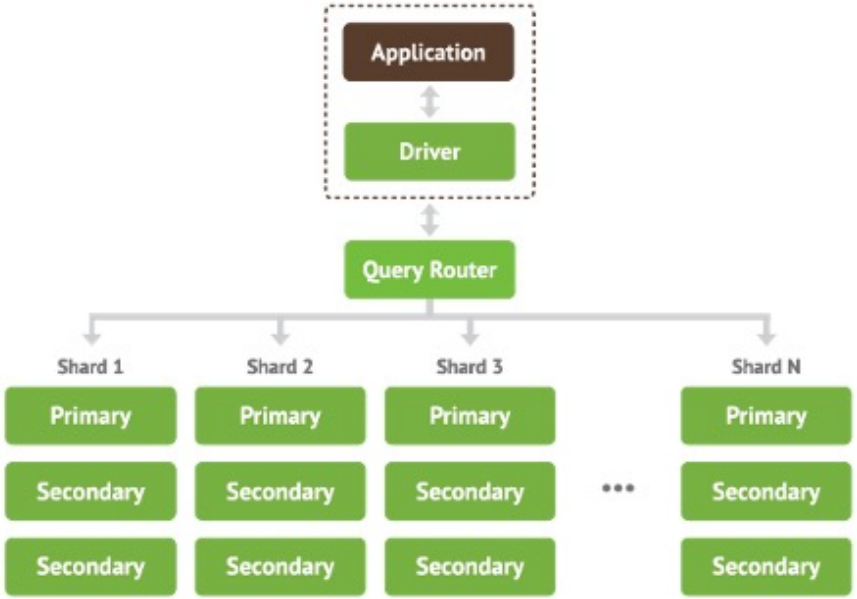
Disk Architecture for Scale-Out

- Share disks:
  - Is basically scale-up for data/disks. You can use NAS, SAN and RAID.
  - Isolation/Integrity requires distributed locking to control access from multiple database servers.
- Share nothing:
  - Is basically scale-out for disks.
  - Data is partitioned into *shards* based on a function f() applied to a key.
  - Can improve availability, at the code consistency, with data replication.
  - There is a router that sends requests to the proper shard based on the function.

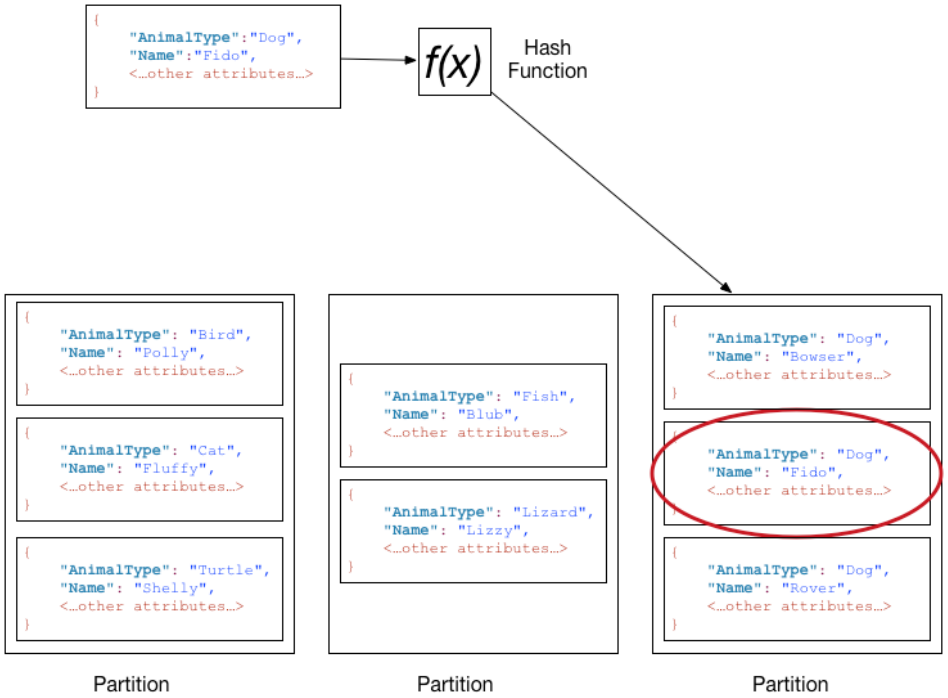


Shared Nothing, Scale-Out

MongoDB Sharding



DynamoDB Partitioning



Database-as-a-Service

Cloud Concepts – One Perspective

Categorizing and Comparing the Cloud Landscape

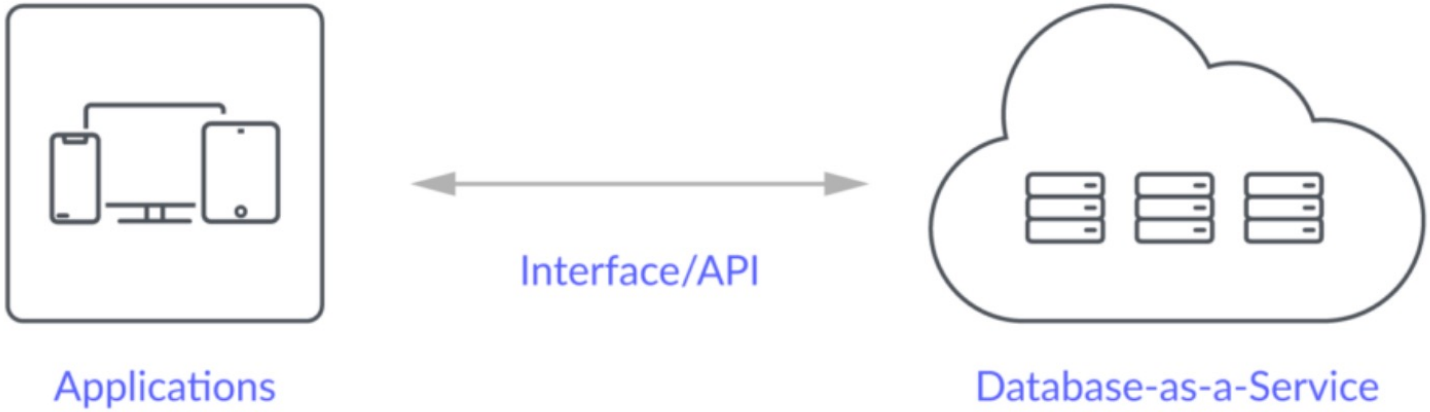
http://www.theenterprisearchitect.eu/blog/2013/10/12/the-cloud-landscape-described-categorized-and-compared/

6	**SaaS**	Applications			*End-users*
5	**App Services**	App Services	Communication and Social Services	Data-as-a-Service	*Citizen Developers*
4	**Model-Driven PaaS**	Model-Driven aPaaS, bpmPaaS	Model-Driven iPaaS	Data Analytics, baPaaS	*Rapid Developers*
3	**PaaS**	aPaaS	iPaaS	dbPaaS	*Developers / Coders*
2	**Foundational PaaS**	Application Containers	Routing, Messaging, Orchestration	Object Storage	*DevOps*
1	**Software-Defined Datacenter**	Virtual Machines	Software-Defined Networking (SDN), NFV	Software-Defined Storage (SDS), Block Storage	*Infrastructure Engineers*
0	**Hardware**	Servers	Switches, Routers	Storage	
		Compute	Communicate	Store	

Database-as-a-Service

“A cloud database is a database that typically runs on a cloud computing platform and access to the database is provided as-a-service. There are two common deployment models: users can run databases on the cloud independently, using a virtual machine image, or they can purchase access to a database service, maintained by a cloud database provider. Of the databases available on the cloud, some are SQL-based and some use a NoSQL data model.

Database services take care of scalability and high availability of the database. Database services make the underlying software-stack transparent to the user.” (Wikipedia)

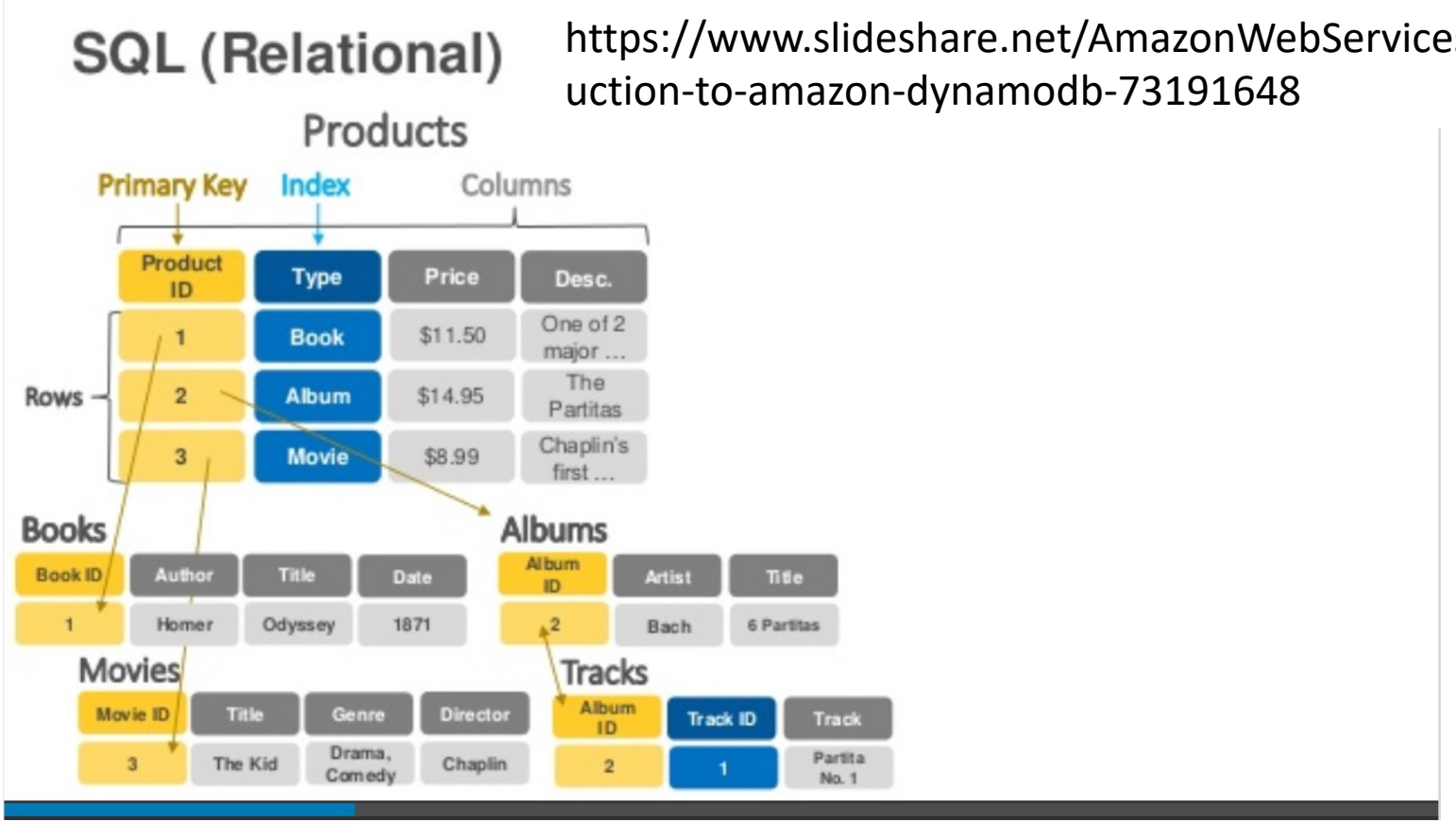


Let’s take a look at 3: Relational Data Service, Dynamo DB, MongoDB (Compass)

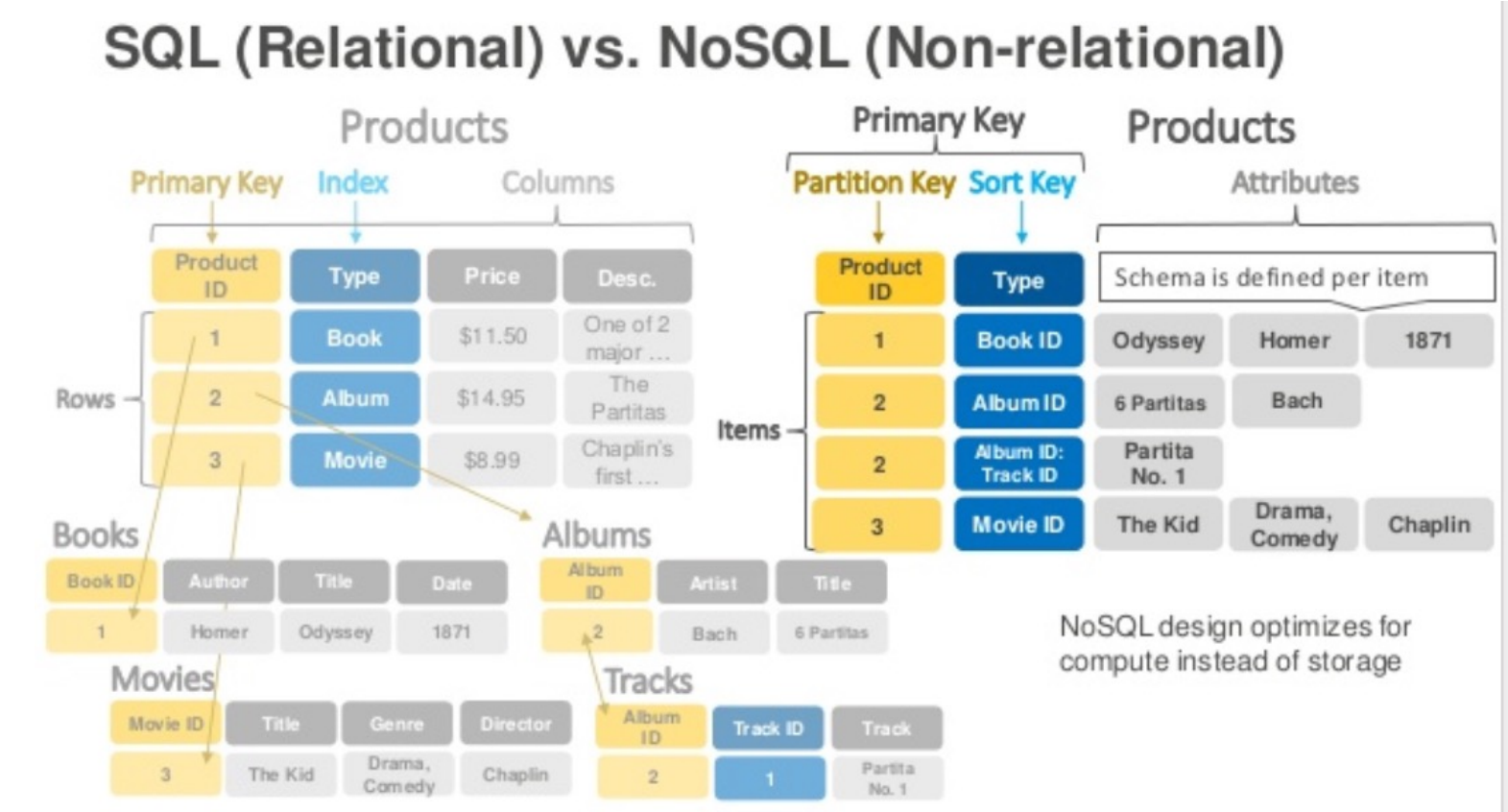
DynamoDB

DynamoDB – Relational

https://www.slideshare.net/AmazonWebServices/introduction-to-amazon-dynamodb-73191648



Dynamo DB – Relational



DynamoDB Benefits

DynamoDB Benefits


Fully managed


Fast, consistent performance

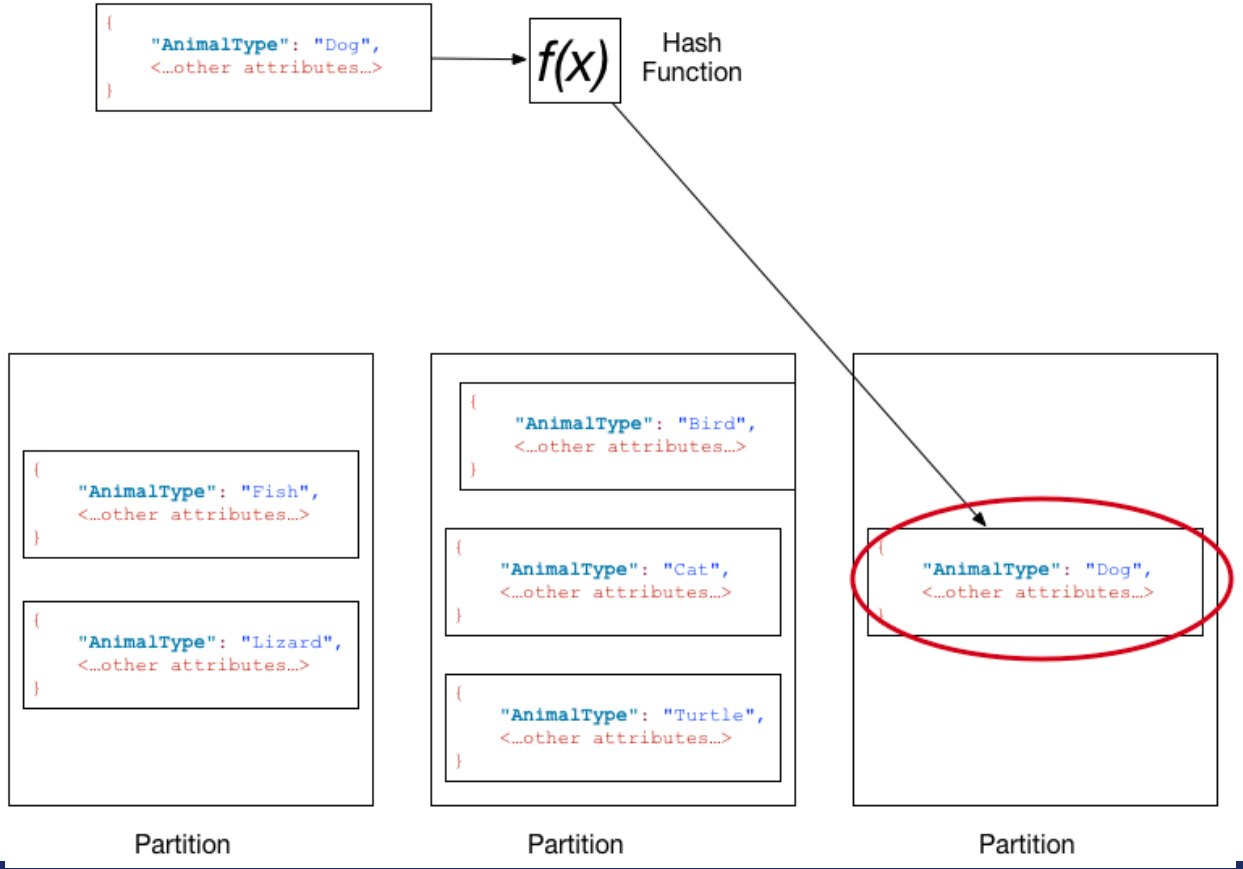

Highly scalable


Flexible

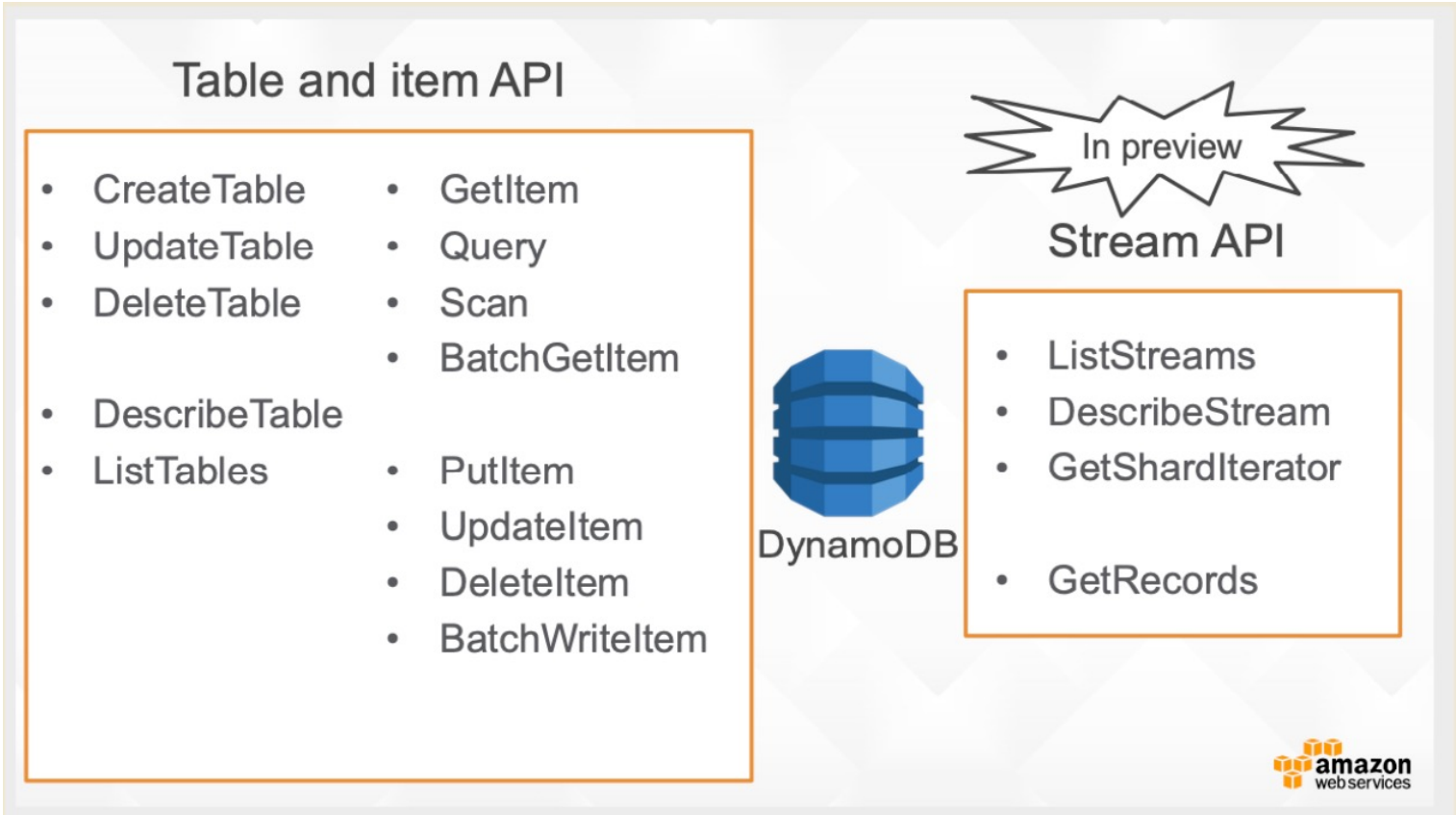

Event-driven programming


Fine-grained access control

DynamoDB – Hash Key



DynamoDB Model



DynamoDB Model

Data types

- String (S)
- Number (N)
- Binary (B)

- String Set (SS)
- Number Set (NS)
- Binary Set (BS)

- Boolean (BOOL)
- Null (NULL)
- List (L)
- Map (M)

Used for storing nested JSON documents

amazon web services

DynamoDB Model

Documents (JSON)

Data types (M, L, BOOL, NULL) introduced to support JSON

Document SDKs

- Simple programming model
- Conversion to/from JSON
- Java, JavaScript, Ruby, .NET

Cannot create an Index on elements of a JSON object stored in Map

- They need to be modeled as top-level table attributes to be used in LSIs and GSIs

Set, Map, and List have no element limit but depth is 32 levels

Javascript	DynamoDB
string	S
number	N
boolean	BOOL
null	NULL
array	L
object	M

```
var image = { // JSON Object for an image
    imageid: 12345,
    url: 'http://example.com/awesome_image.jpg'
};
var params = {
    TableName: 'images',
    Item: image, // JSON Object to store
};
dynamodb.putItem(params, function(err, data){
    // response handler
};
```

Rich expressions

Projection expression

- Query/Get/Scan: ProductReviews.FiveStar[0]

Filter expression

- Query/Scan: #V > :num (#V is a place holder for keyword VIEWS)

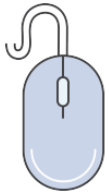
Conditional expression

- Put/Update/DeleteItem: attribute_not_exists (#pr.FiveStar)

Update expression

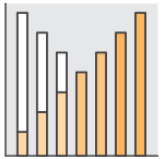
- UpdateItem: set Replies = Replies + :num

Amazon DynamoDB



Fully Managed NoSQL

Document or Key-Value



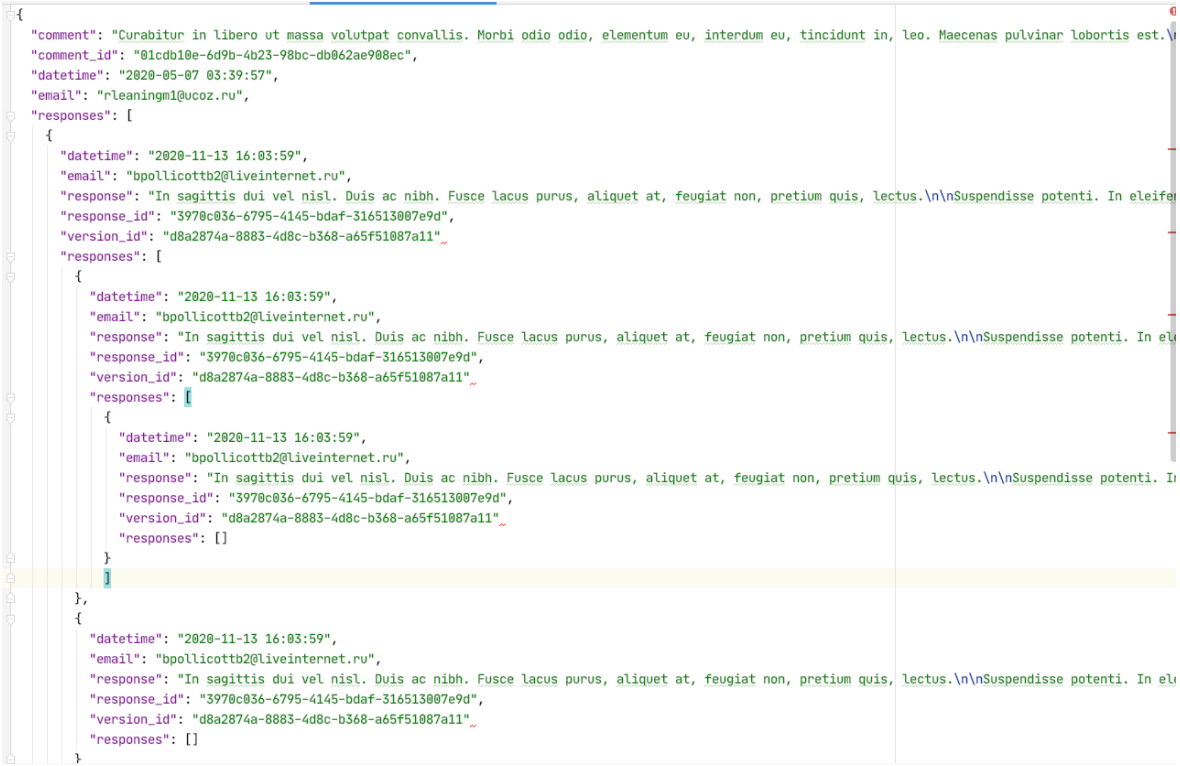
Scales to Any Workload

Fast and Consistent

Access Control

Event Driven Programming

Document Databases – Semi-Structured Schema



The relation model has difficulties with some types of data:

- Columns that are lists or maps
- Nesting – things inside things
- Discover you need a modified schema when you get a piece of data.
- etc.

All of these are common for documented like data:

- Threaded discussions.
- Documents.
- … …

- Document DBs emerged to handle these scenarios.

DynamoDB Summary

- Achieves much greater scalability and performance than RDBMs
- Does not support some RDBMs capabilities:
  - Referential integrity.
  - JOIN
  - Queries limited to key fields.
  - Non-key field queries are always scans.
- Data model better fit for *semi-structured* data models and good fit for JSON
  - Maps
  - Lists

batch_get_item()	get_waiter()
batch_write_item()	list_backups()
can_paginate()	list_global_tables()
create_backup()	list_tables()
create_global_table()	list_tags_of_resource()
create_table()	put_item()
delete_backup()	query()
delete_item()	restore_table_from_backup()
delete_table()	restore_table_to_point_in_time()
describe_backup()	scan()
describe_continuous_backups()	tag_resource()
describe_endpoints()	untag_resource()
describe_global_table()	update_continuous_backups()
describe_global_table_settings()	update_global_table()
describe_limits()	update_global_table_settings()
describe_table()	update_item()
describe_time_to_live()	update_table()
generate_presigned_url()	update_time_to_live()
get_item()	get_paginator()

DynamoDB Python API