E6156 – Topics in SW Engineering (F23)
Cloud Computing
Lecture 10: DynamoDB, Design Patterns, Composition, Asynch, Step Functions

Service Composition Continued

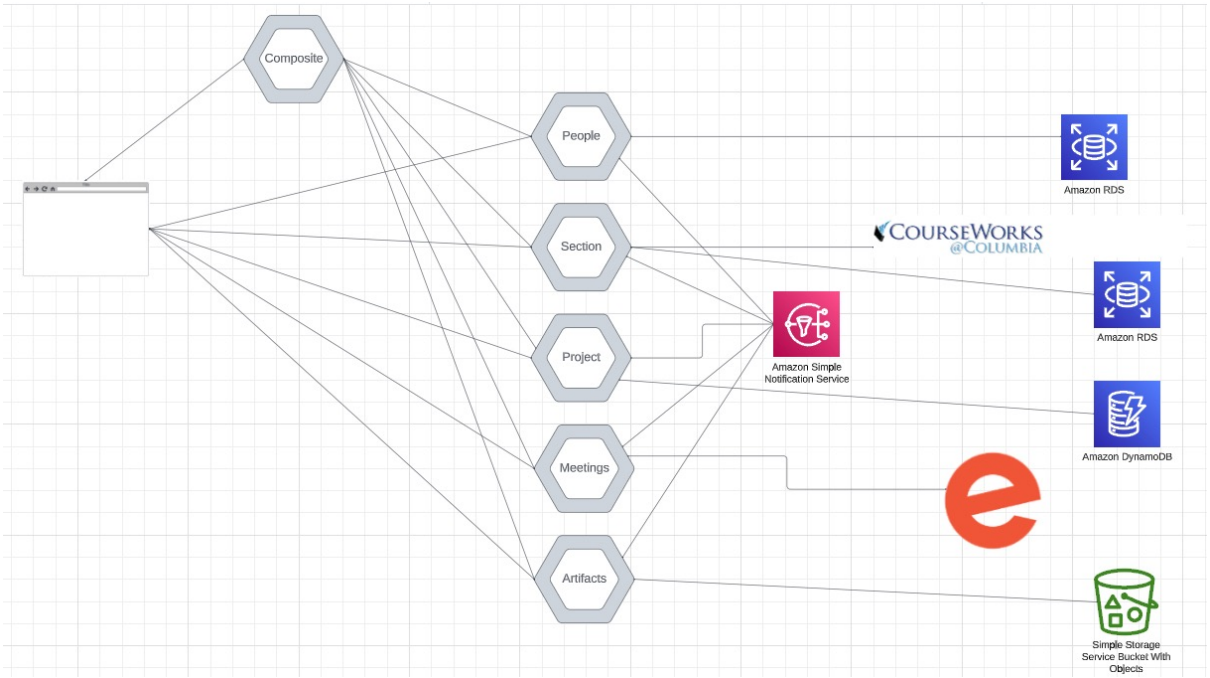
Overview/Review

Concepts

- There two core concepts in “composition:”
  - Structure:
    - “What is connected to what?”
    - “How does component A find the implementation of interface B that is uses?”
  - Behavior:
    - The internal logic of a component is encapsulated.
    - “What is the behavior of the overall system built on individual components behavior.”
- There are logical or conceptual ways to think about and implement the concepts:
  - Structure: C4 Models, UML Class and Component Diagrams, Dependency Injection, Service Factory/Locator, … …
  - Behavior: State Machines, Sequence Diagrams, BPMN, Step Functions, Workflow Engines, … …

Structural Composition

Structural Composition



- There are a lot of lines connecting “components” and “services.”
- How does this work?
  - Ultimately, the composition relies on URLs.
  - But, how does a service know the URLs?
  - Hardcoding is a bad idea.
  - Also, how do we encapsulate implementation changes to prevent cascading code changes?
- There is not generally agreed approach, but there are various patterns.

Do Not do This … …

```
7 usages  Donald F. Ferguson (Ansys) *
class StudentResource:
    #
    # These endpoints are on Prof. Ferguson's SwaggerHub mock APIs
    #
    resources = [
        {
            "resource": "student",
            "url": 'https://virtserver.swaggerhub.com/donald-f-ferguson/E6156Student/1.0.0/students/bb2101'
        },
        {
            "resource": "sections",
            "url": 'https://virtserver.swaggerhub.com/Columbia-Classes/Sections/1.0.0/sections?uni=bb2101'
        },
        {
            "resource": "projects",
            "url": 'https://virtserver.swaggerhub.com/Columbia-Classes/ProjectInfo/1.0.0/projects?uni=bb2021'
        }
    ]
```

- Clearly, hard coding URLs, etc. is not the right approach.
- But, other people must have solved this problem.

```
1 usage  Donald F. Ferguson (Ansys) *
async def get_student_async(self):
    full_result = None
    start_time = time.time()
    async with aiohttp.ClientSession() as session:
        tasks = [asyncio.ensure_future(
            StudentResource.fetch(session, res)) for res in StudentResource.resources]
        responses = await asyncio.gather(*tasks)
        full_result = {}
        for response in responses:
            full_result[response["resource"]] = response["data"]
        end_time = time.time()
        full_result["elapsed_time"] = end_time - start_time

        return full_result

        # print("\nFull Result = ", json.dumps(full_result, indent=2))

1 usage  Donald F. Ferguson (Ansys) *
async def get_student_sync(self):
    full_result = None
    start_time = time.time()

    full_result = {}

    for r in StudentResource.resources:
        response = requests.get(r["url"])
        full_result[r["resource"]] = response.json()
    end_time = time.time()
    full_result["elapsed_time"] = end_time - start_time

    return full_result
```

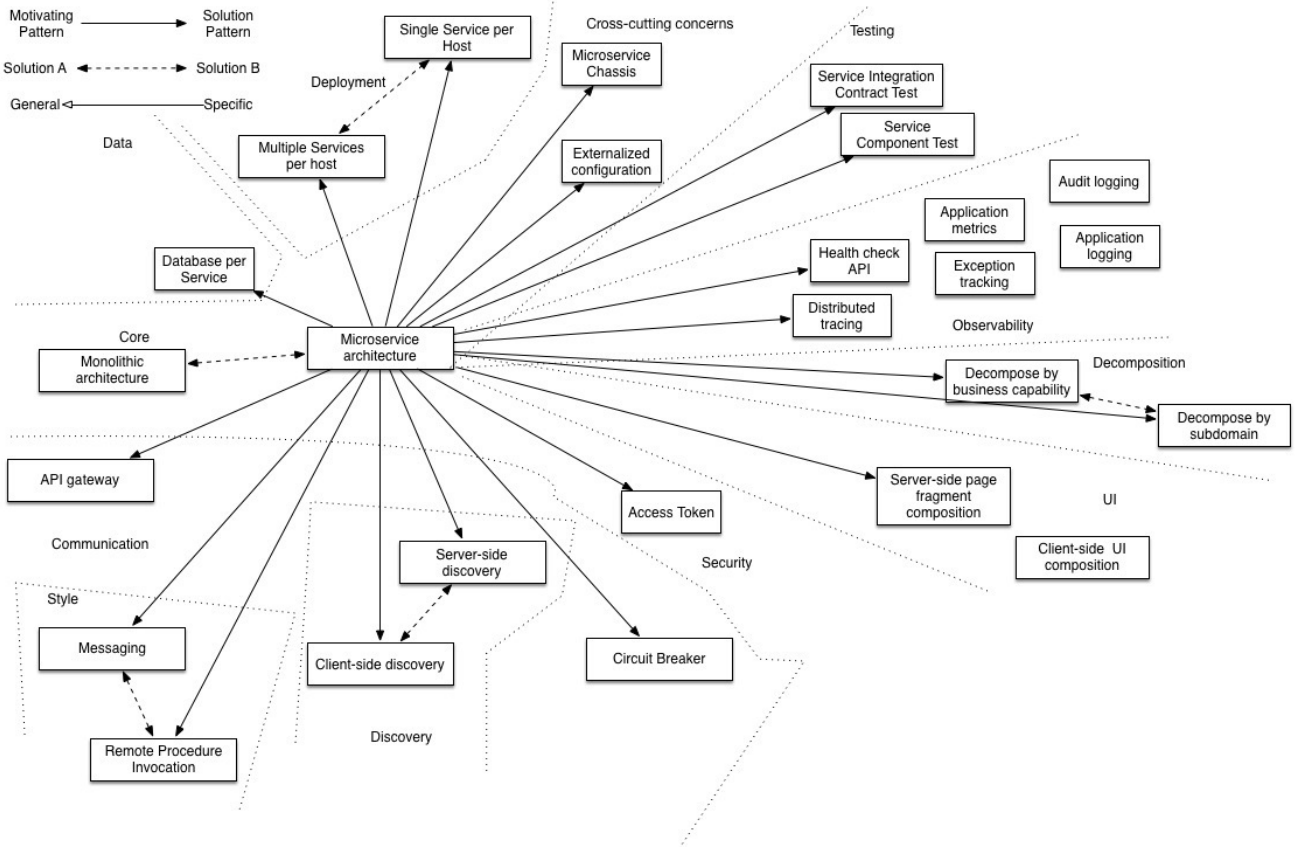
Design Patterns:
A Digression

Design Patterns

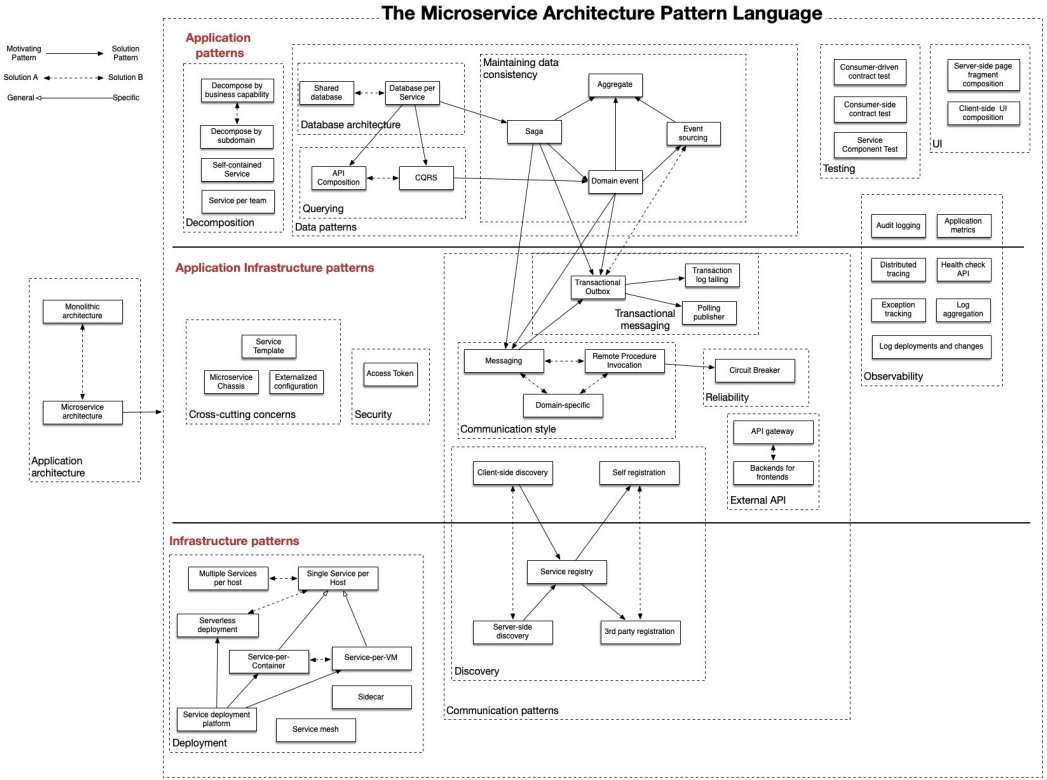
- "In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system." (https://en.wikipedia.org/wiki/Software_design_pattern)

- Design patterns occur at many levels and contexts:
  - Data Modeling
  - Implementing OO applications, FastAPI applications, … …
  - Microservices
  - … …

Microservice Patterns (One Perspective)



Microservice Pattern Language (One Perspective)

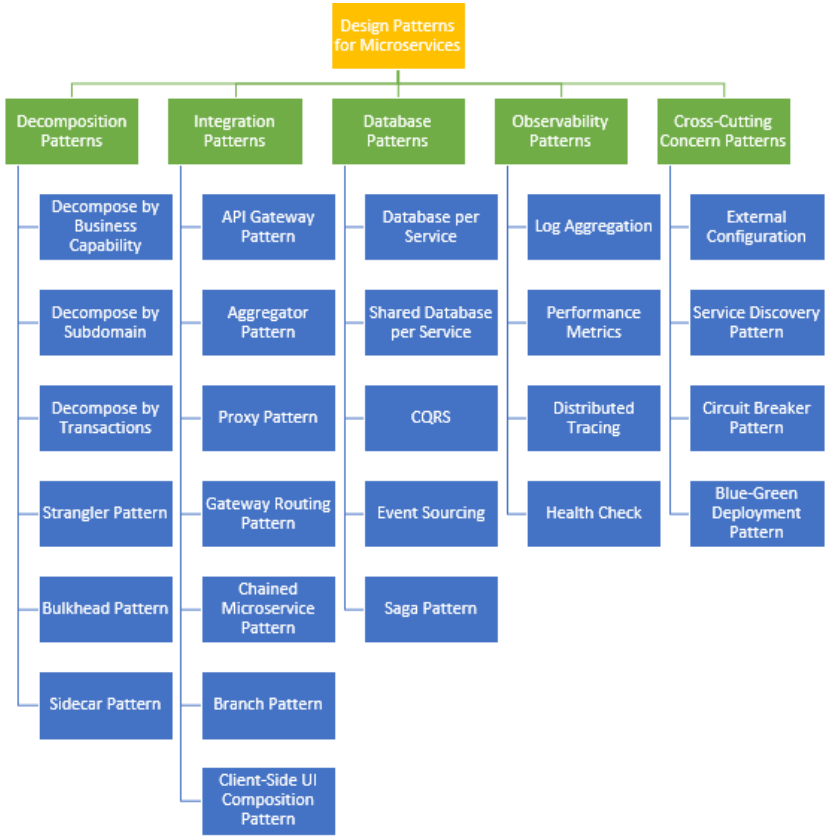


https://microservices.io/patterns/index.html

Microservice Pattern Language (One Perspective)

Some patterns that I regularly

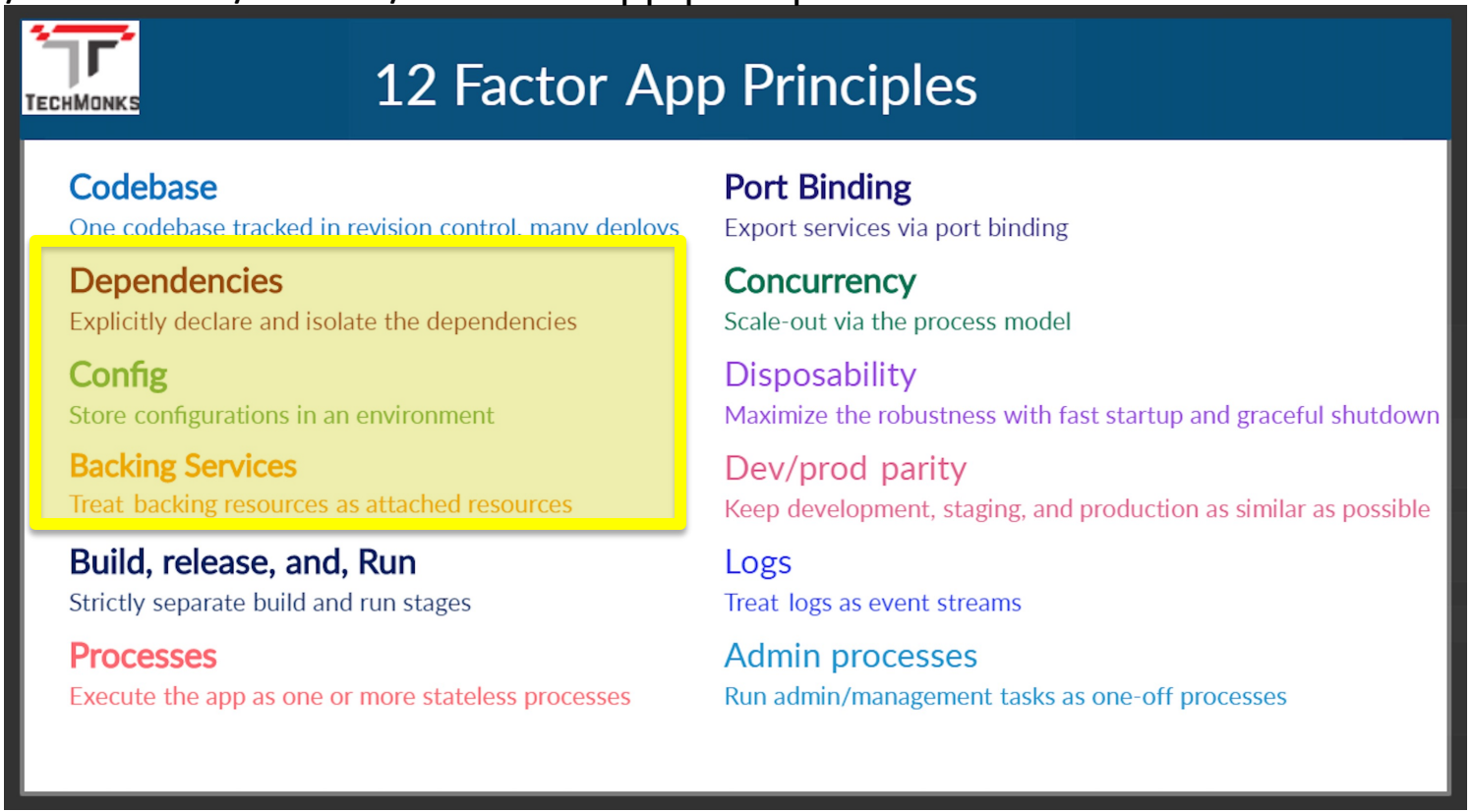
- Strangler
- Sidecar
- Aggregator
- Database per Service
- CQRS
- Saga
- External Configuration
- … …



Composition Continued (Structure)

12 Factor Applications

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



Dependencies

II. Dependencies

Explicitly declare and isolate dependencies

Most programming languages offer a packaging system for distributing support libraries, such as CPAN for Perl or Rubygems for Ruby. Libraries installed through a packaging system can be installed system-wide (known as "site packages") or scoped into the directory containing the app (known as "vendoring" or "bundling").

A twelve-factor app never relies on implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a *dependency declaration* manifest. Furthermore, it uses a *dependency isolation* tool during execution to ensure that no implicit dependencies "leak in" from the surrounding system. The full and explicit dependency specification is applied uniformly to both production and development.

For example, Bundler for Ruby offers the `Gemfile` manifest format for dependency declaration and `bundle exec` for dependency isolation. In Python there are two separate tools for these steps – Pip is used for declaration and Virtualenv for isolation. Even C has Autoconf for dependency declaration, and static linking can provide dependency isolation. No matter what the toolchain, dependency declaration and isolation must always be used together – only one or the other is not sufficient to satisfy twelve-factor.

One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. The new developer can check out the app's codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. They will be able to set up everything needed to run the app's code with a deterministic *build command*. For example, the build command for Ruby/Bundler is `bundle install`, while for Clojure/Leiningen it is `lein deps`.

Twelve-factor apps also do not rely on the implicit existence of any system tools. Examples include shelling out to ImageMagick or `curl`. While these tools may exist on many or even most systems, there is no guarantee that they will exist on all systems where the app may run in the future, or whether the version found on a future system will be compatible with the app. If the app needs to shell out to a system tool, that tool should be vendored into the app.

- Most application frameworks have a dependency declaration concept.
- I have been showing examples with:
  - Python import
  - requirements.txt
  - etc.
- My examples also show encapsulating libraries and APIs with Python classes.
- II. Dependencies handles "the code," but we still need to handle the instance. There is a difference between, e.g.
  - pymysql
  - A specific connection.

III. Configurations

III. Config

Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires **strict separation of config from code**. Config varies substantially across deploys, code does not.

A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

Note that this definition of "config" does **not** include internal application config, such as `config/routes.rb` in Rails, or how code modules are connected in Spring. This type of config does not vary between deploys, and so is best done in the code.

Another approach to config is the use of config files which are not checked into revision control, such as `config/database.yml` in Rails. This is a huge improvement over using constants which are checked into the code repo, but still has weaknesses: it's easy to mistakenly check in a config file to the repo; there is a tendency for config files to be scattered about in different places and different formats, making it hard to see and manage all the config in one place. Further, these formats tend to be language- or framework-specific.

The twelve-factor app stores config in *environment variables* (often shortened to *env vars* or *env*). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

Another aspect of config management is grouping. Sometimes apps batch config into named groups (often called "environments") named after specific deploys, such as the `development`, `test`, and `production` environments in Rails. This method does not scale cleanly: as more deploys of the app are created, new environment names are necessary, such as `staging` or `qa`. As the project grows further, developers may add their own special environments like `joes-staging`, resulting in a combinatorial explosion of config which makes managing deploys of the app very brittle.

In a twelve-factor app, env vars are granular controls, each fully orthogonal to other env vars. They are never grouped together as "environments", but instead are independently managed for each deploy. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime.

- Each of the deployment environments we have seen has a method for setting environment variables.
  - https://docs.aws.amazon.com/cloud9/latest/user-guide/env-vars.html
  - https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/environments-cfg-softwaresettings.html
  - https://cloud.google.com/functions/docs/configuring/env-var
- CI/CD, GitHub actions, etc. can integrate environment variables with the development process.
- Vaults and secrets managers are a better approach for passwords.

IV. Backing Services

IV. Backing services

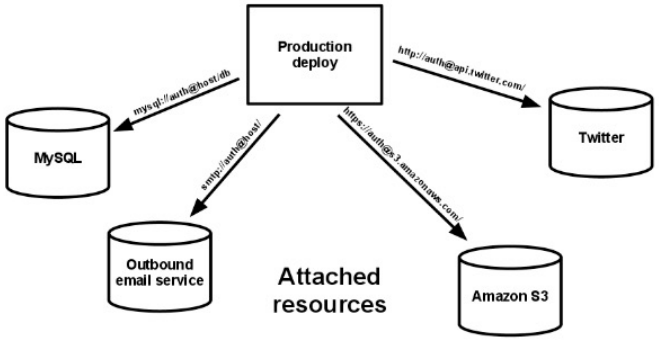
Treat backing services as attached resources

A *backing service* is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).

Backing services like the database are traditionally managed by the same systems administrators who deploy the app's runtime. In addition to these locally-managed services, the app may also have services provided and managed by third parties. Examples include SMTP services (such as Postmark), metrics-gathering services (such as New Relic or Loggly), binary asset services (such as Amazon S3), and even API-accessible consumer services (such as Twitter, Google Maps, or Last.fm).

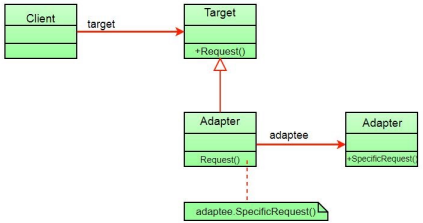
The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.

Each distinct backing service is a *resource*. For example, a MySQL database is a resource; two MySQL databases (used for sharding at the application layer) qualify as two distinct resources. The twelve-factor app treats these databases as *attached resources*, which indicates their loose coupling to the deploy they are attached to.



Resources can be attached to and detached from deploys at will. For example, if the app's database is misbehaving due to a hardware issue, the app's administrator might spin up a new database server restored from a recent backup. The current production database could be detached, and the new database attached – all without any code changes.

- My code has shown encapsulating backing services with a shallow adaptor.
- This is an example of the *adaptor pattern.*



- You seen examples of things like AbstractBaseDataService, … …
- The approach is also an application of several aspects of SOLID. (https://en.wikipedia.org/wiki/SOLID)

Structural Composition

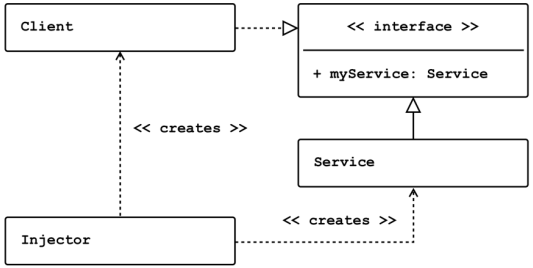
- We can use four patterns/principles for service composition ”in code.”
  - Dependency Injection
  - Service Factory
  - Service Locator
  - Metadata in the environment for configuration.

- I did some simple examples in https://github.com/donald-f-ferguson/E6156-Public-Examples/tree/master/simple_pattern_examples.

- Or actually, I asked ChatGPT to produce simple examples.

Dependency Injection

- Concepts (https://en.wikipedia.org/wiki/Dependency_injection):
  - In software engineering, dependency injection is a technique in which an object receives other objects that it depends on, called dependencies.
  - The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.



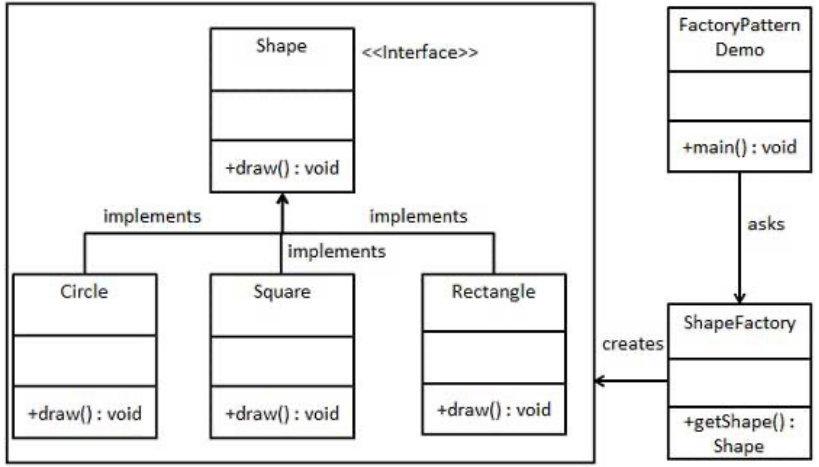

```
class UserResource(BaseApplicationResource):

    def __init__(self, config_info):
        super().__init__(config_info)
```

- There is no single approach. There are several frameworks, tools, etc.
- Like many other things, it is easy to get carried away and become dogmatic.
- I follow a single approach:
  - A Context class converts environment and other configuration and provides to application.
  - The top-level application injects a config_info object into services.

Service Factory

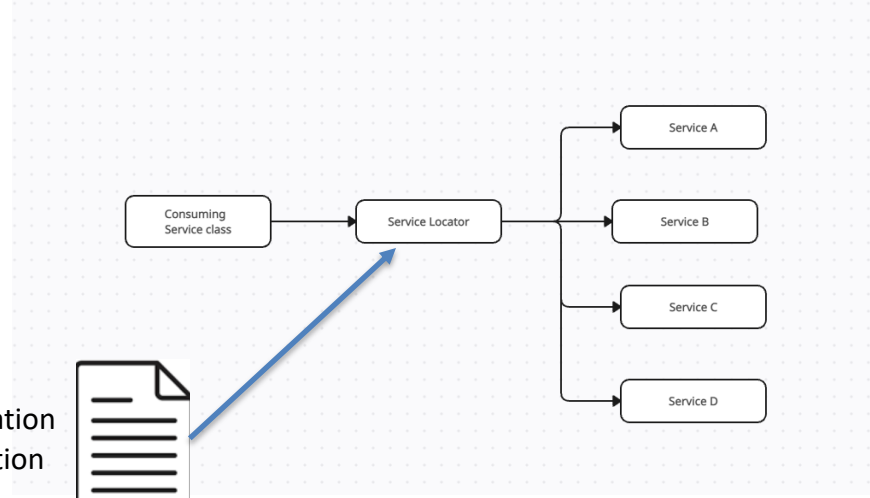
- "In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor." (https://en.wikipedia.org/wiki/Factory_method_pattern)
- You will sometimes see me use this for
  - Abstraction between a REST resource impl. and the data service.
  - Allows changing the database service, model, etc. without modifying the code.
  - Concrete implementation choices are configured via properties, metadata, …
- You can use in many situations.



https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Service Locator

- “The service locator pattern is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task. Proponents of the pattern say the approach simplifies component-based applications where all dependencies are cleanly listed at the beginning of the whole application design, …” (https://en.wikipedia.org/wiki/Service_locator_pattern)



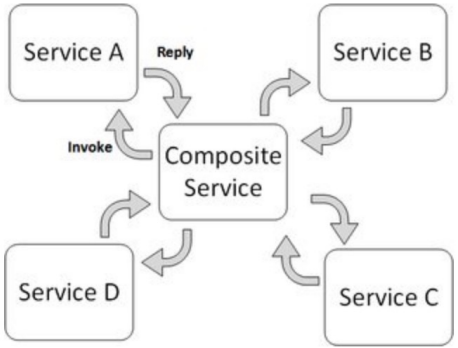
Behavioral Composition

Concepts

Service orchestration

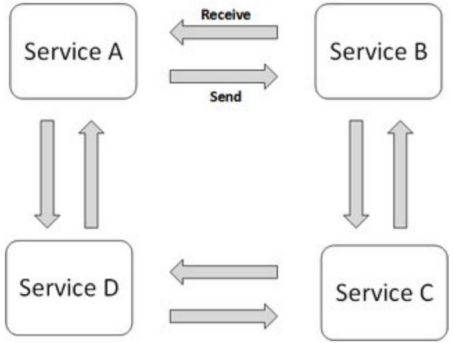
Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

The relationship between all the participating services are described by a single endpoint (i.e the composite service). The orchestration includes the management of transactions betweer individual services. Orchestration employs a centralized approach for service composition.



Service Choreography

Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints. Choreography employs a decentralized approach for service composition.

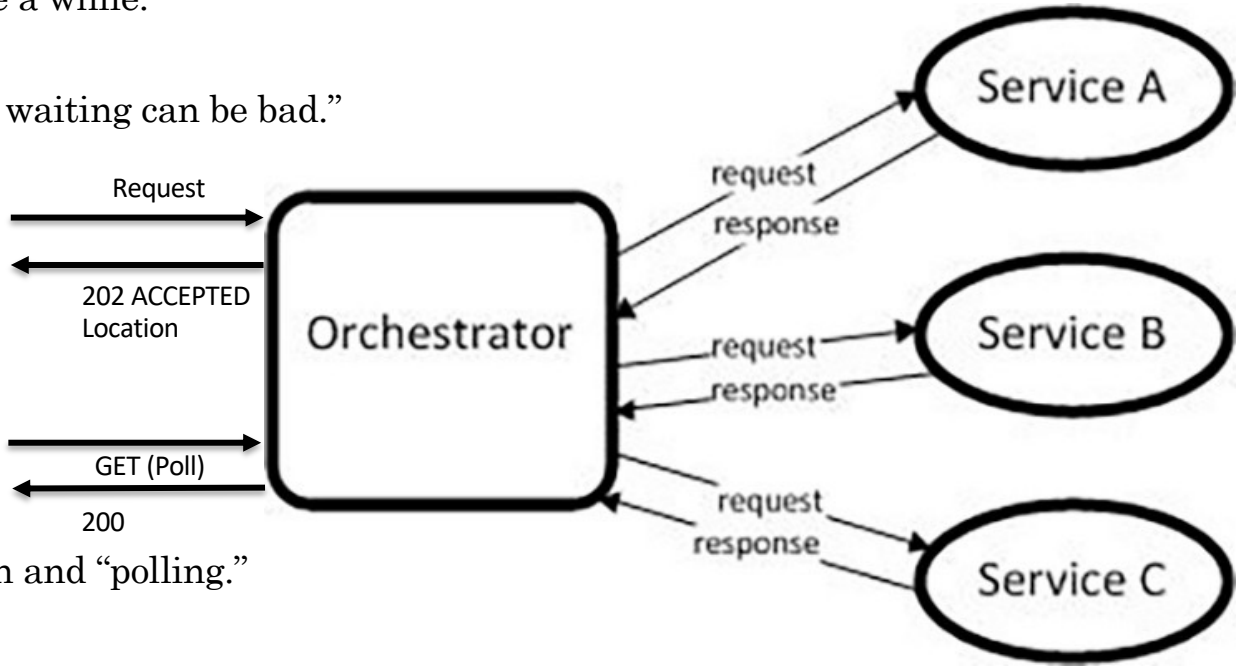


The choreography describes the interactions between multiple services, where as orchestration represents control from one party's perspective. This means that a **choreography** *differs* from an **orchestration** with respect to where the logic that controls the interactions between the services involved should reside.

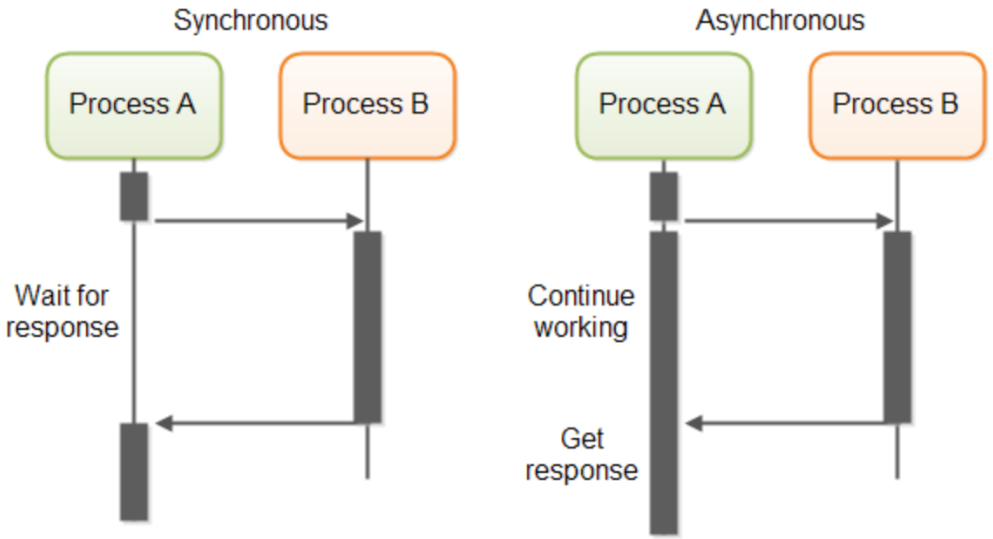
These are not formally, rigorously defined terms.

In Code: Composition and Asynchronous Method Invocation

- Composite operations can “take a while.”
- “Holding open connections and waiting can be bad.”
  - Consumes resources.
  - Connections can break.
  - … …
- There are two options:
  - Polling
  - Callbacks
- Our next pattern is composition and “polling.”
  Create a new user requires:
  - Updating the user database.
  - Verifying a submitted address.
  - Creating an account in the catalog service.



In Code: Service Orchestration/Composition

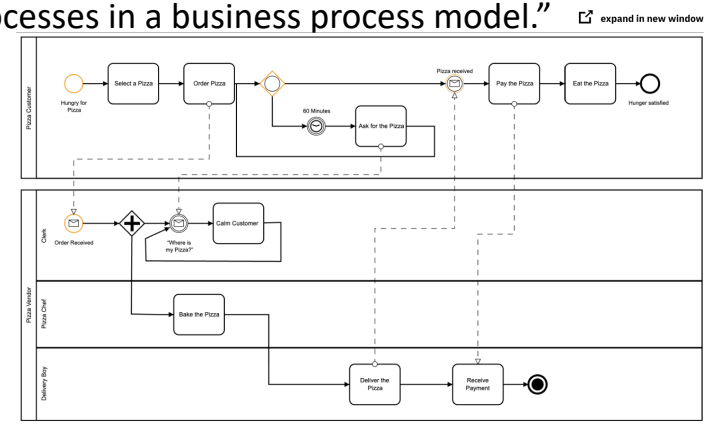


RESTFul HTTP calls can be implemented in both a synchronous and asynchronous fashion at an IO level.

- One call to a composite service
  - May drive calls to multiple other services.
  - Which in turn, may drive multiple calls to other services.
- Synchronous (Wait) and calling one at a time, in order is
  - Inefficient
  - Fragile
  - … …
- Asynchronous has benefits but can result in complex code.

Service Orchestration

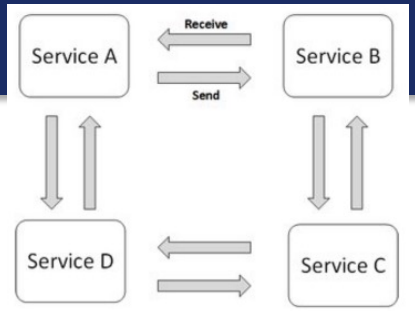
- Implementing a complex orchestration in code can become complex.
- High layer abstractions and tools have emerged, e.g. BPMN.
  - "Business Process Model and Notation (BPMN) is a graphical representation for specifying business processes in a business process model."

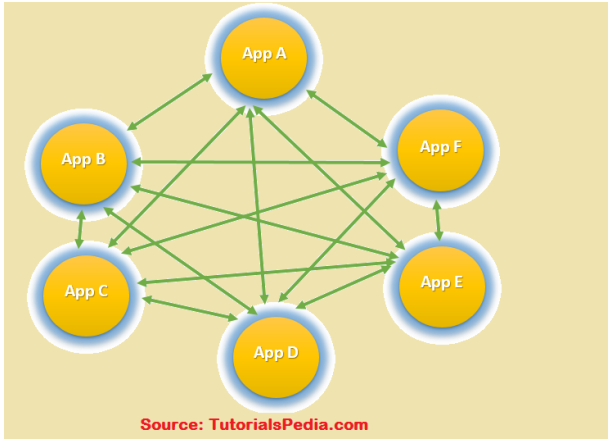


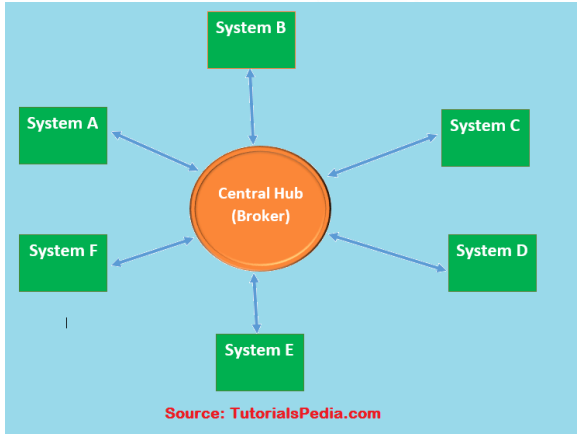
  - There are products for defining BPMN processes, generating code, executing, … …, e.g. Camunda: https://camunda.com/bpmn/
- There are other languages, tools, execution engines, … …

Choreography

- We saw the basic diagram, but …
  this is an anti-pattern and does not scale.

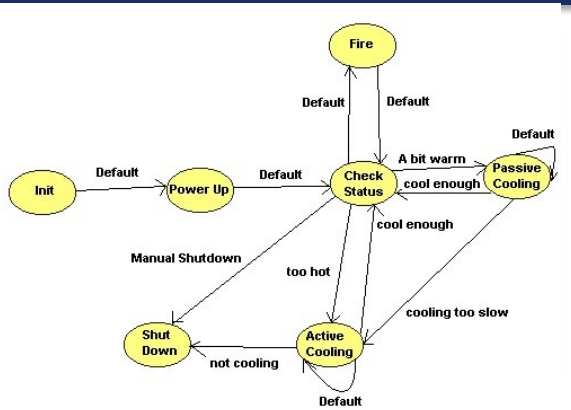


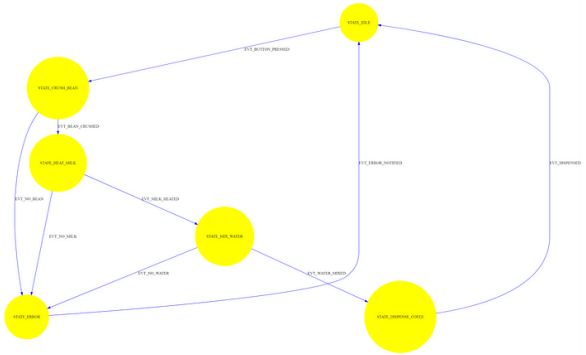


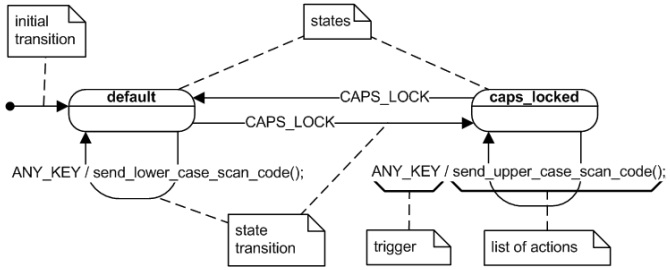


- But, how do you write the event driven microservices?
  - Well, you can just write code … …
  - Or use a state machines abstraction.

State Machine







- A state machine is an abstraction.
- There are many models for defining state machines.
- There are also development tools, runtimes, … …
- We will look at a specific example: AWS Step Functions

Step Functions
Switch to Amazon Presentation

NoSQL and DynamoDB

Cloud Concepts – One Perspective

Categorizing and Comparing the Cloud Landscape

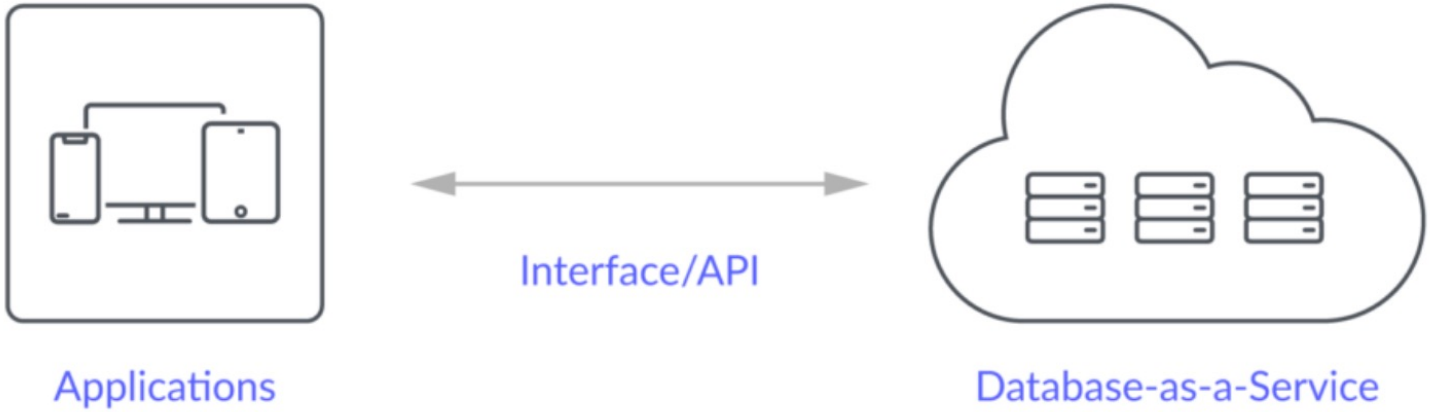
http://www.theenterprisearchitect.eu/blog/2013/10/12/the-cloud-landscape-described-categorized-and-compared/

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)

Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

A **NoSQL** (originally referring to "non SQL" or "non relational")[1] database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,[2] triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.[3][4][5] NoSQL databases are increasingly used in big data and real-time web applications.[6] NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.[7][8]

Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases),[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the data structures used by NoSQL databases are also viewed as "more flexible" than relational database tables.[9]

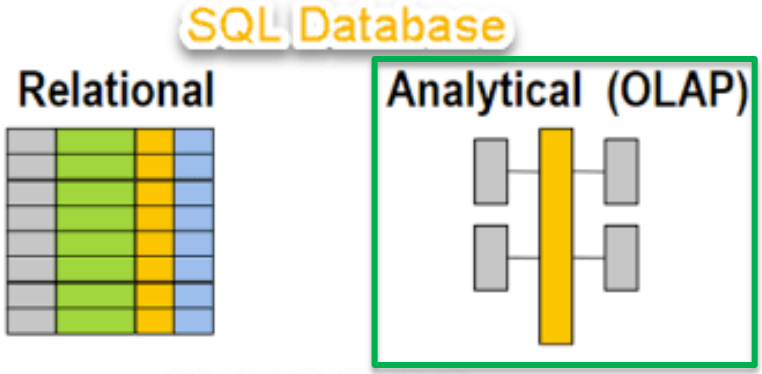
Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.[10] Most NoSQL stores lack true ACID transactions, … …

Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.[11] Additionally, some NoSQL systems may exhibit lost writes and other forms of data loss.[12] Fortunately, some NoSQL systems provide concepts such as write-ahead logging to avoid data loss.[13] For distributed transaction processing across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."[14]

Simplistic Classification (https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b)

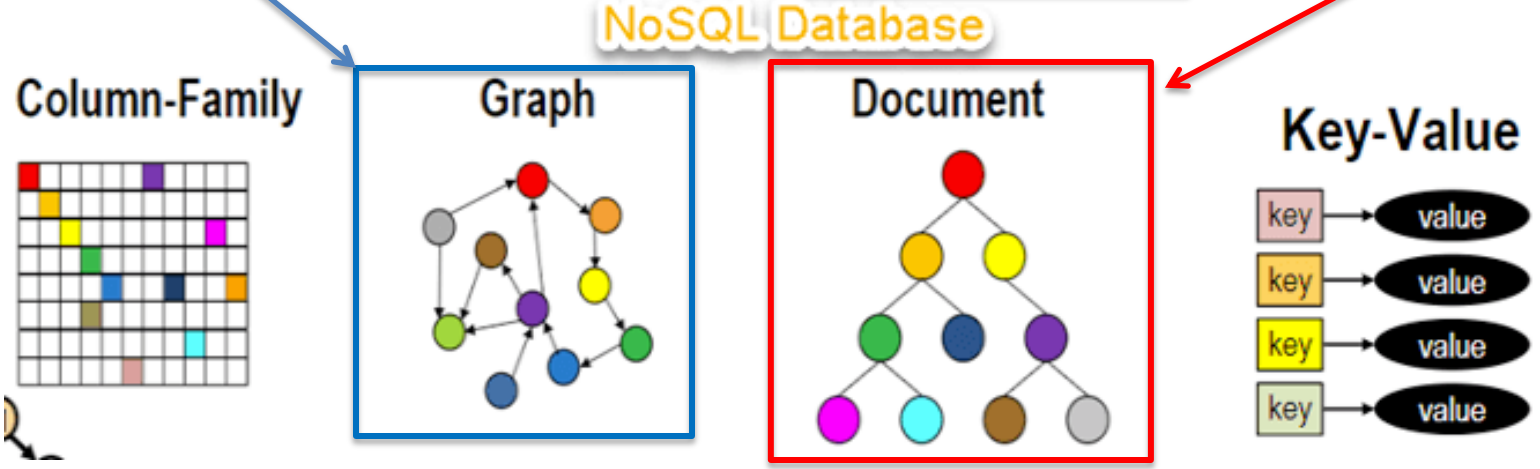
Relational is the foundational model.



We will see OLAP in a future lecture.

We covered graphs and examples.

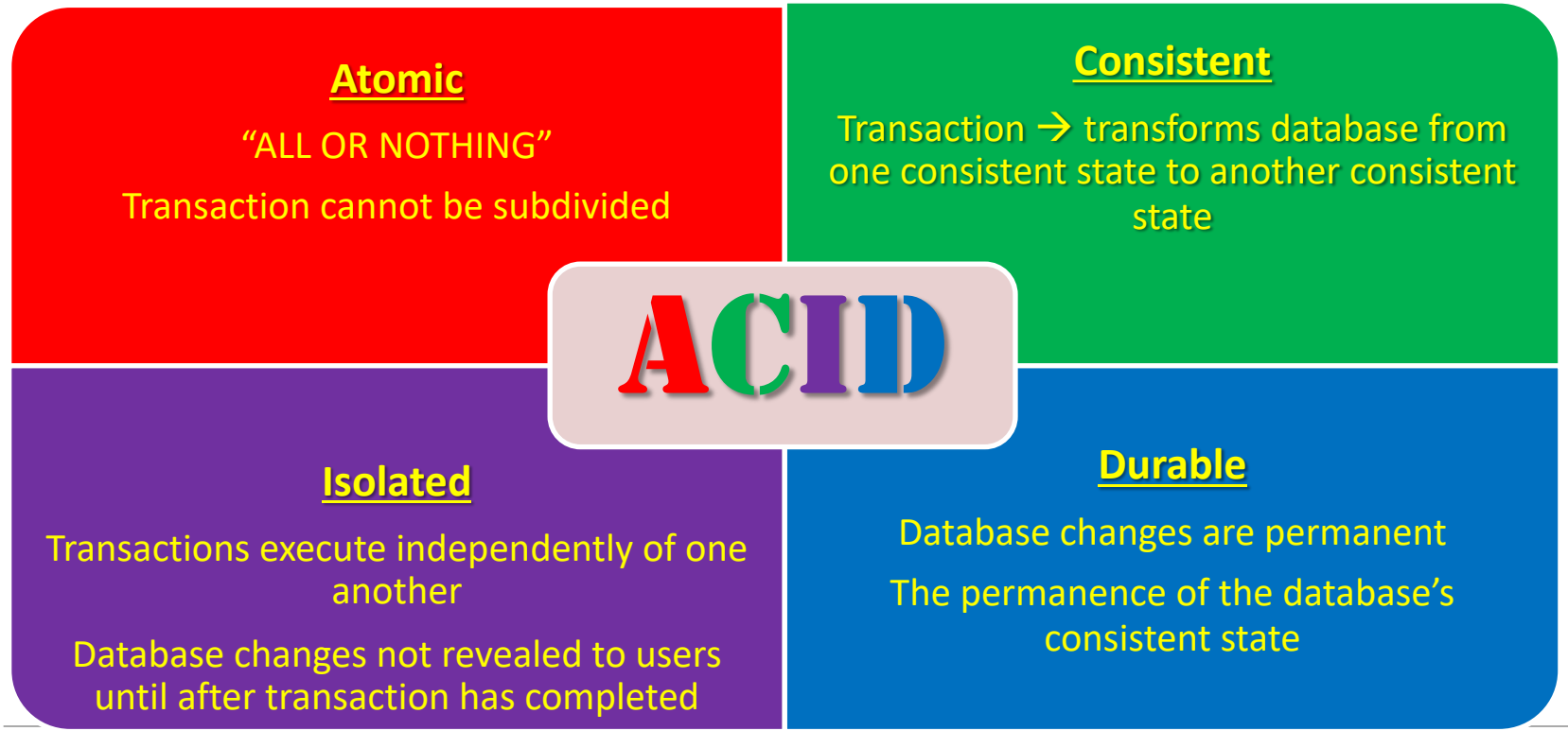
Subject of this lecture and part of HW4



One Taxonomy

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

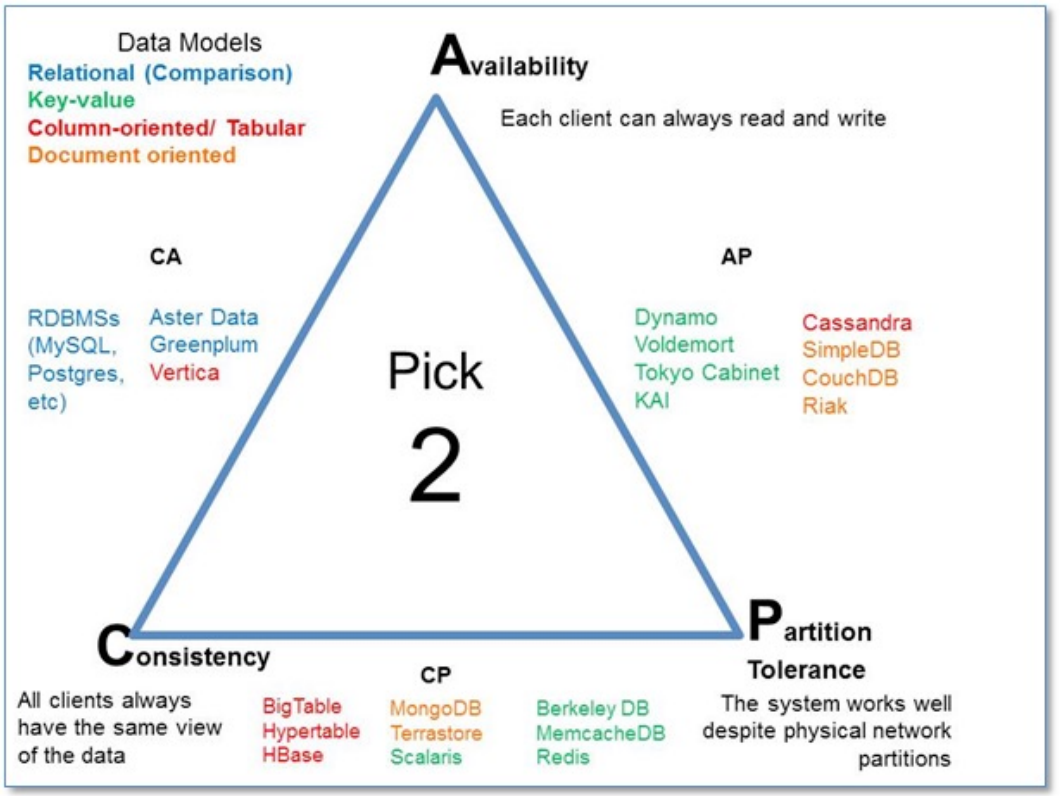
  Every read receives the most recent write or an error.

- 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.

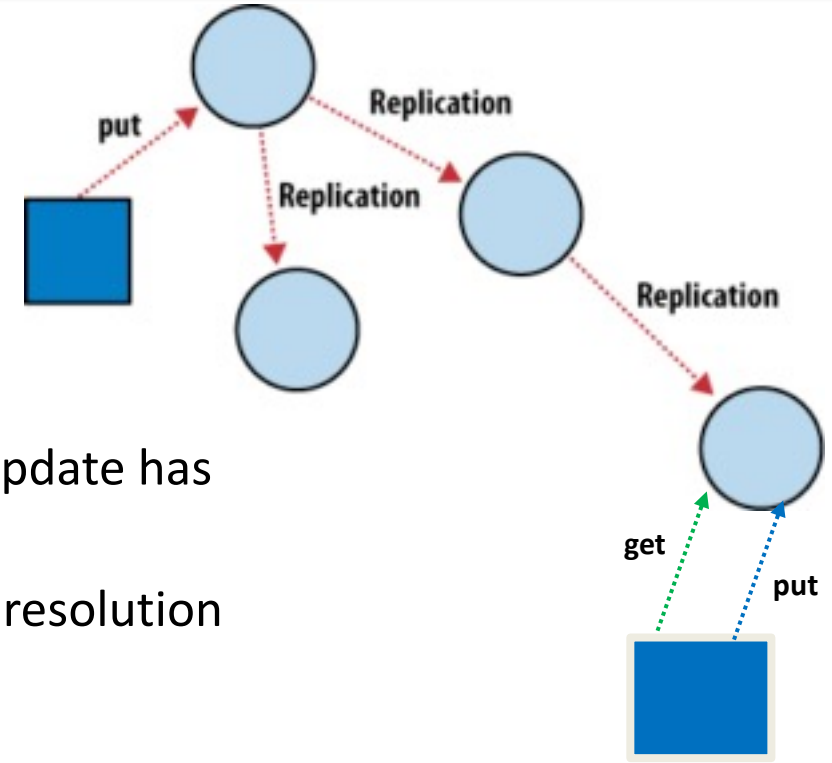
BASE

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)

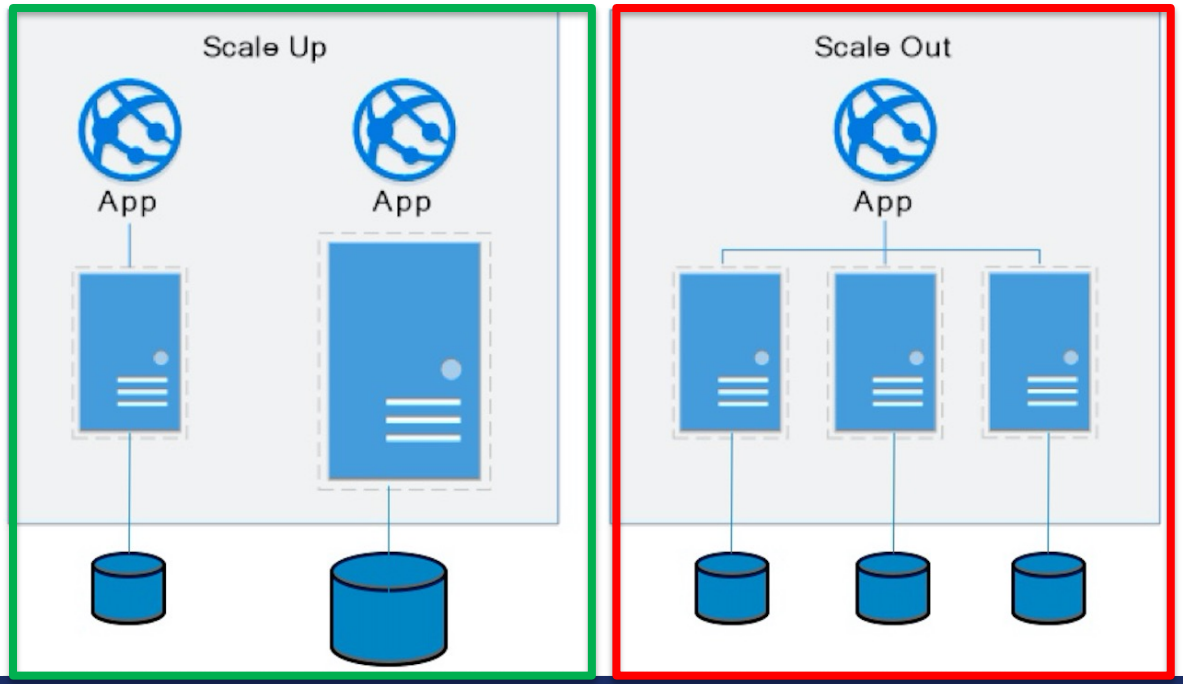
ACID (relational)	BASE (NoSQL)
Strong consistency	Weak consistency
Isolation	Last write wins (Or other strategy)
Transaction	Program managed
Robust database	Simple database
Simple code (SQL)	Complex code
Available and consistent	Available and partition-tolerant
Scale-up (limited)	Scale-out (unlimited)
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, … …

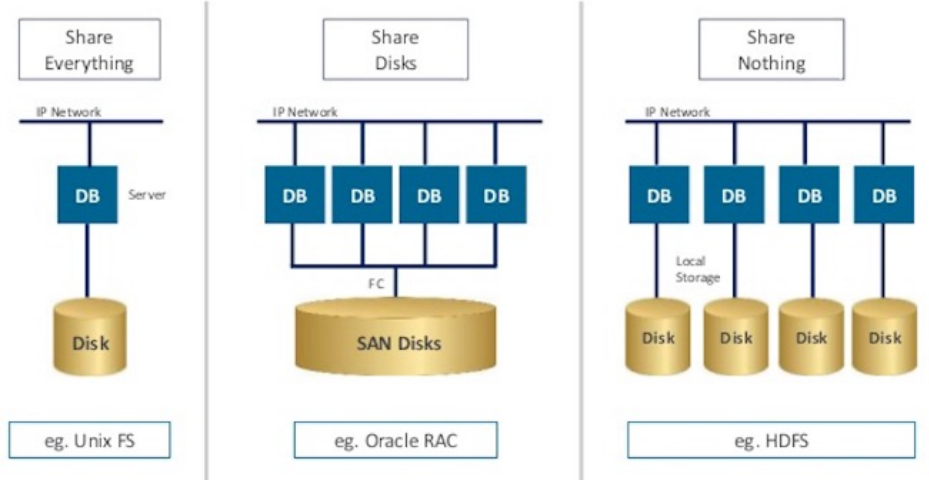
Add another system.



- 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, … …

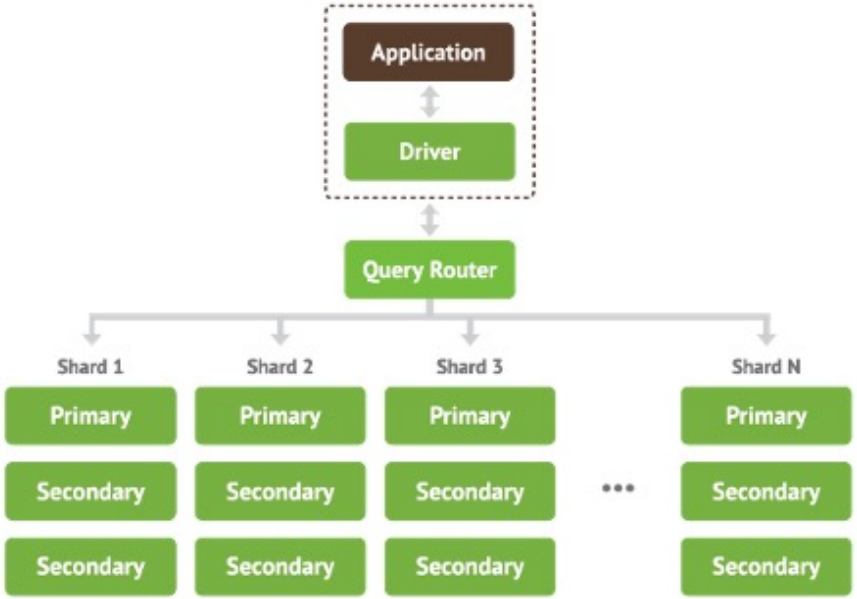
Disk Architecture for Scale-Out

- 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.

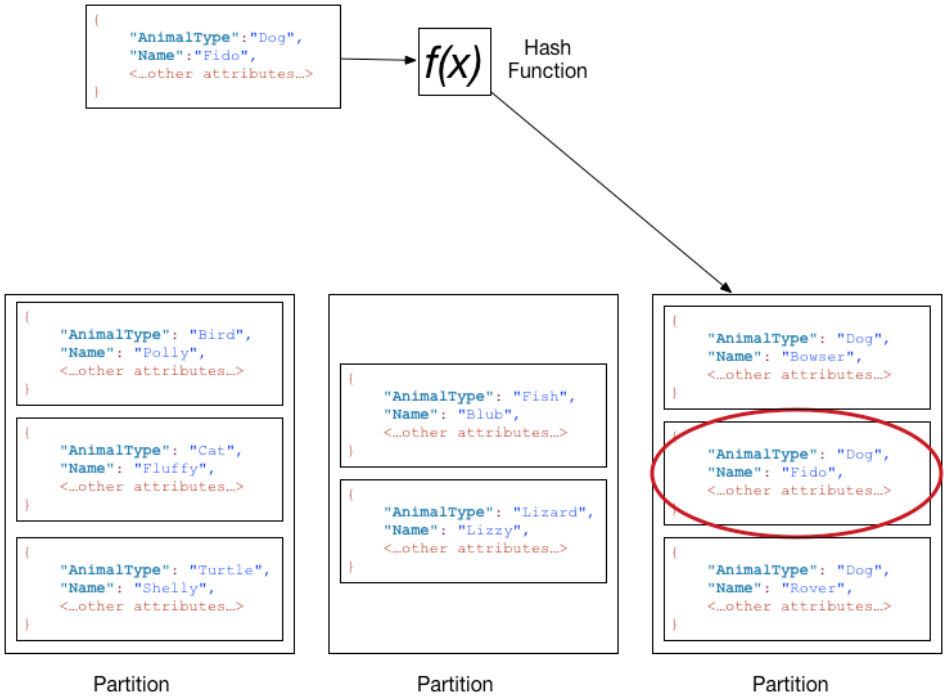


Shared Nothing, Scale-Out

MongoDB Sharding

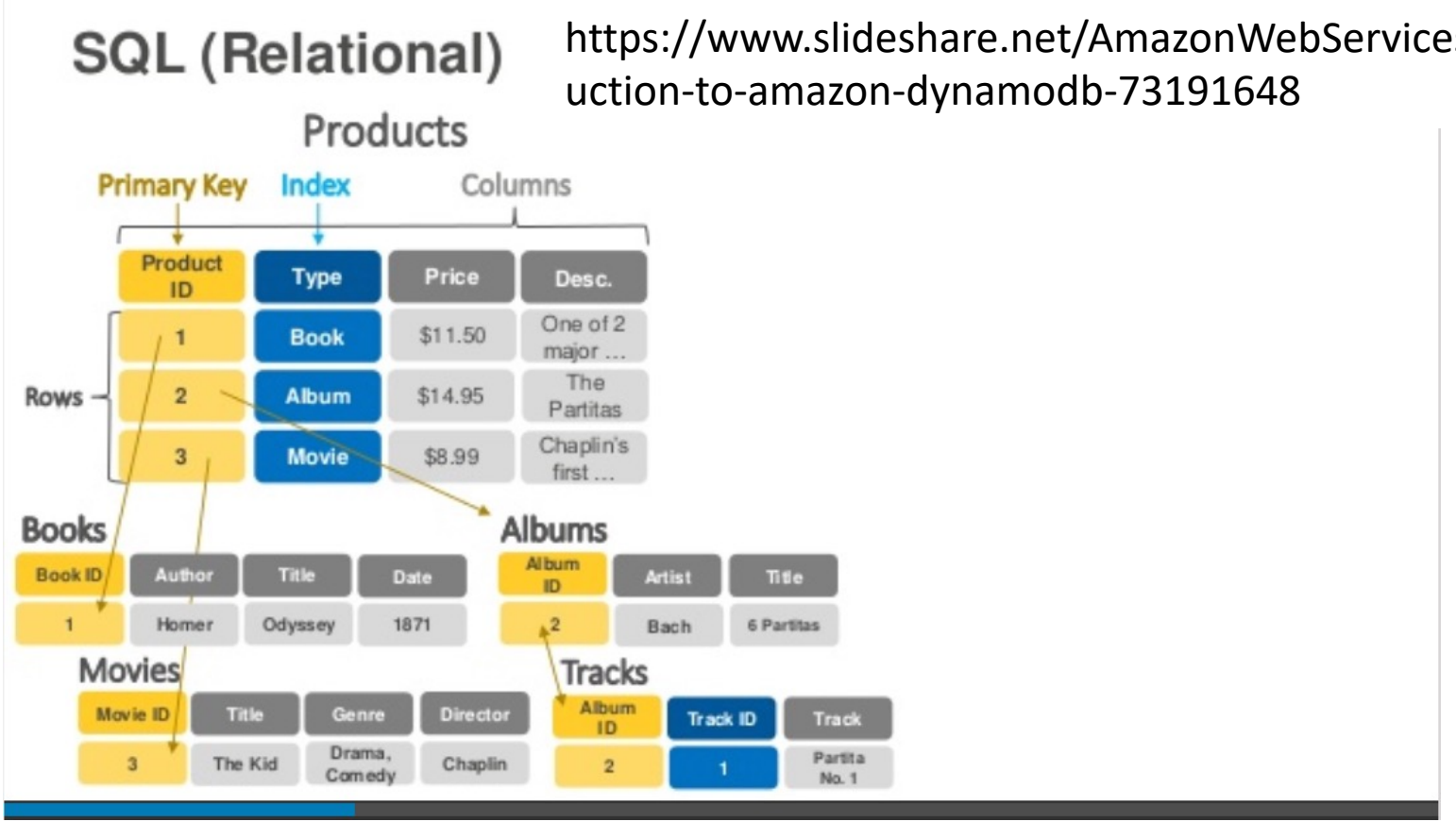


DynamoDB Partitioning



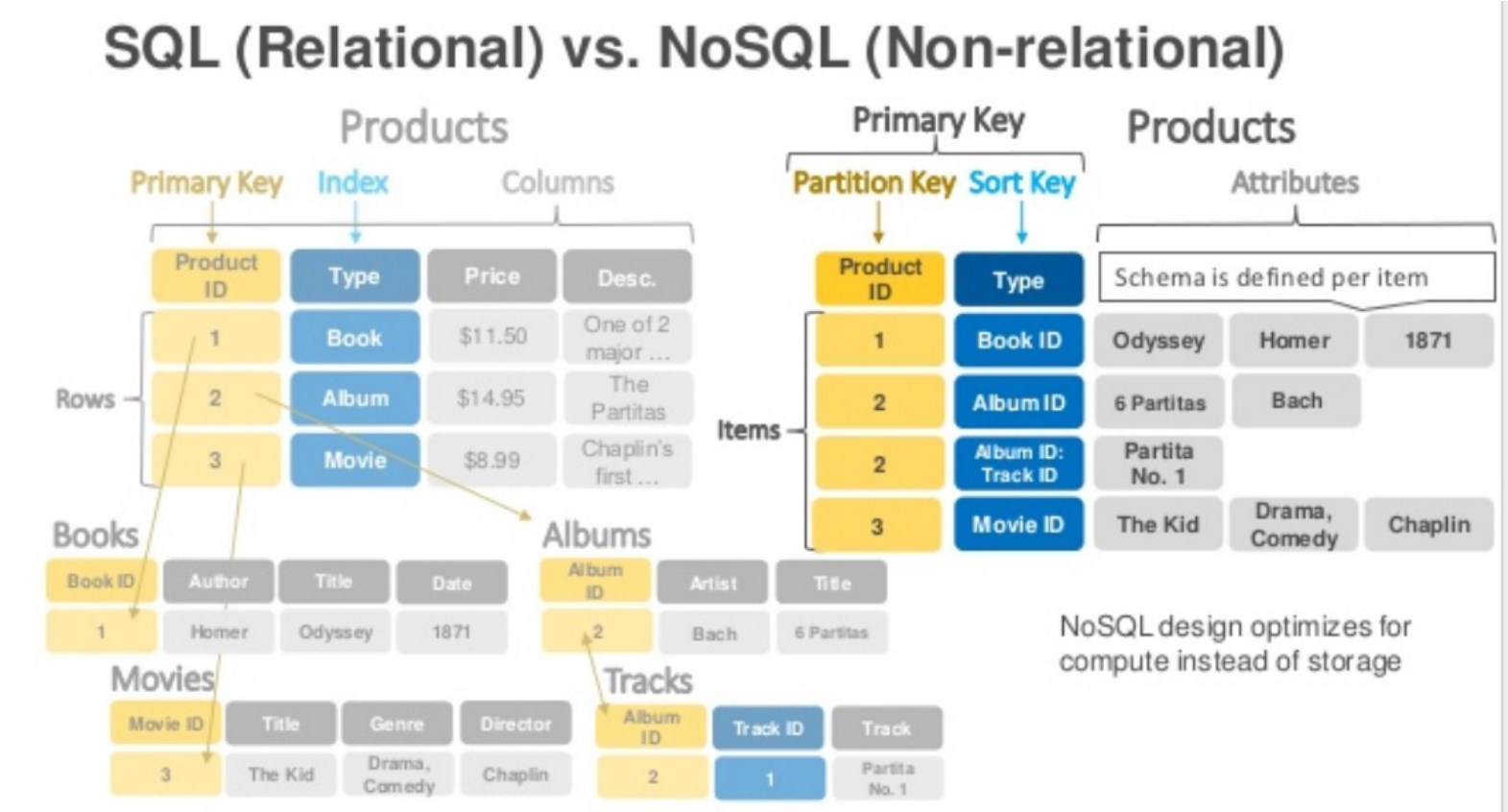
DynamoDB

DynamoDB – Relational



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

Dynamo DB – Relational



DynamoDB Benefits

DynamoDB Benefits

Fully managed

Fast, consistent performance

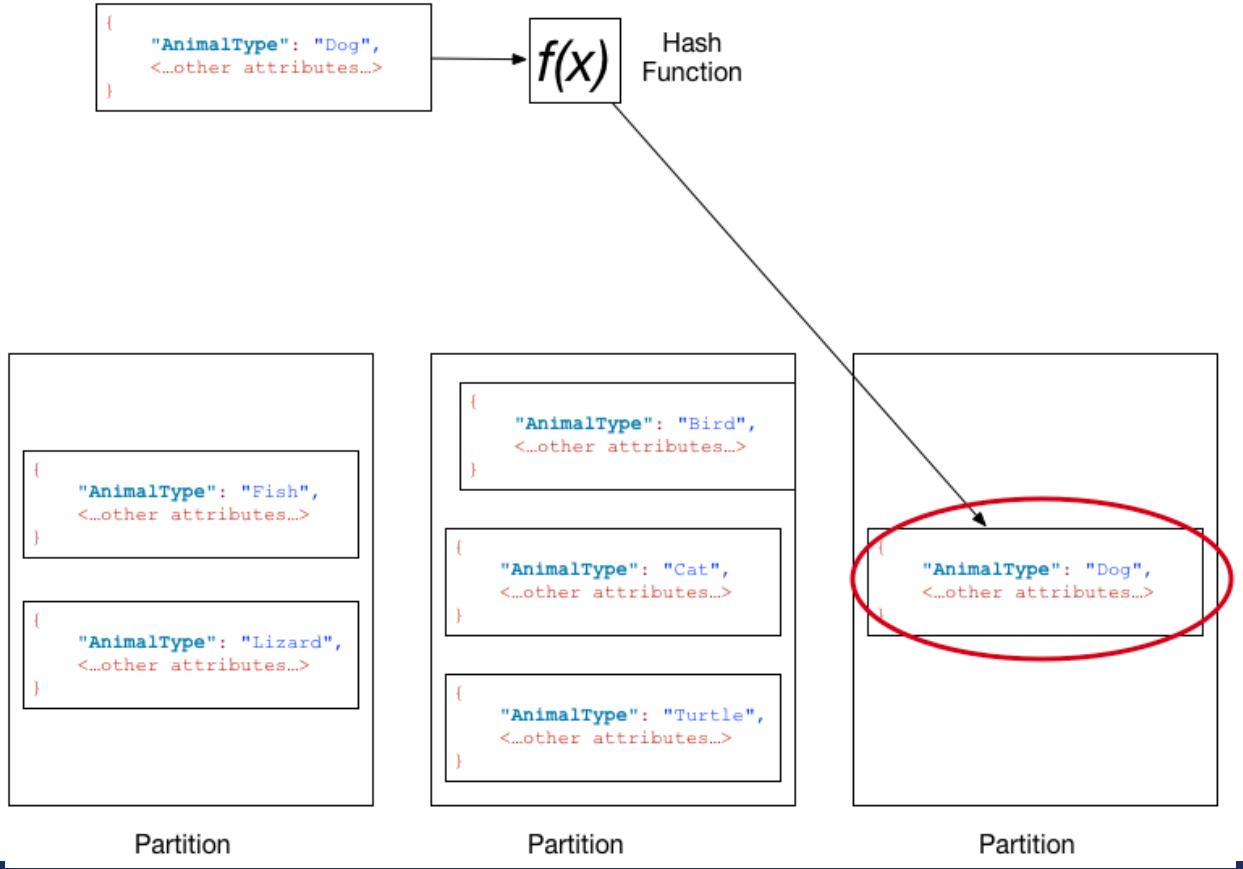
Highly scalable

Flexible

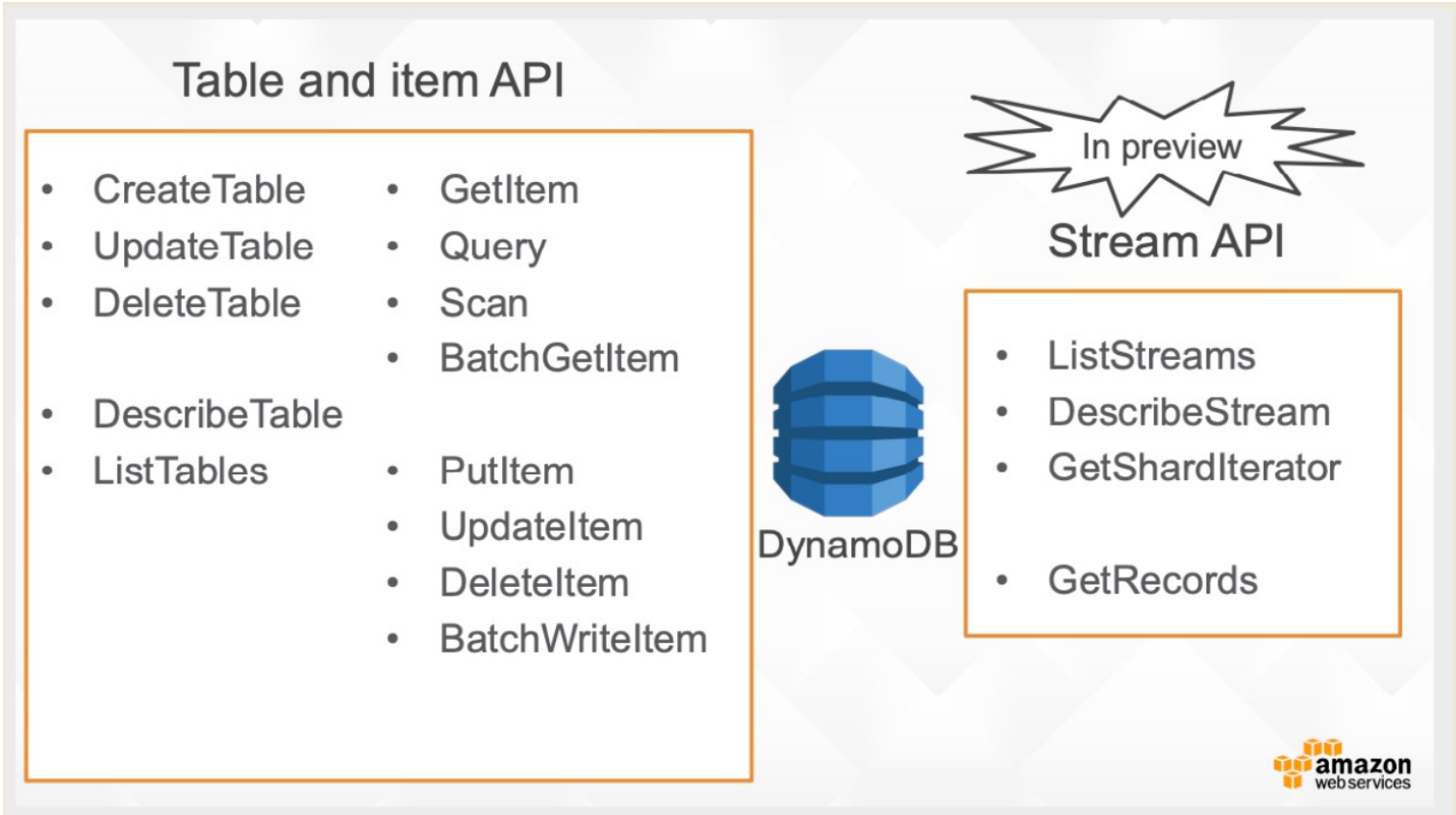
Event-driven programming

Fine-grained access control

DynamoDB – Hash Key



DynamoDB



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

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)

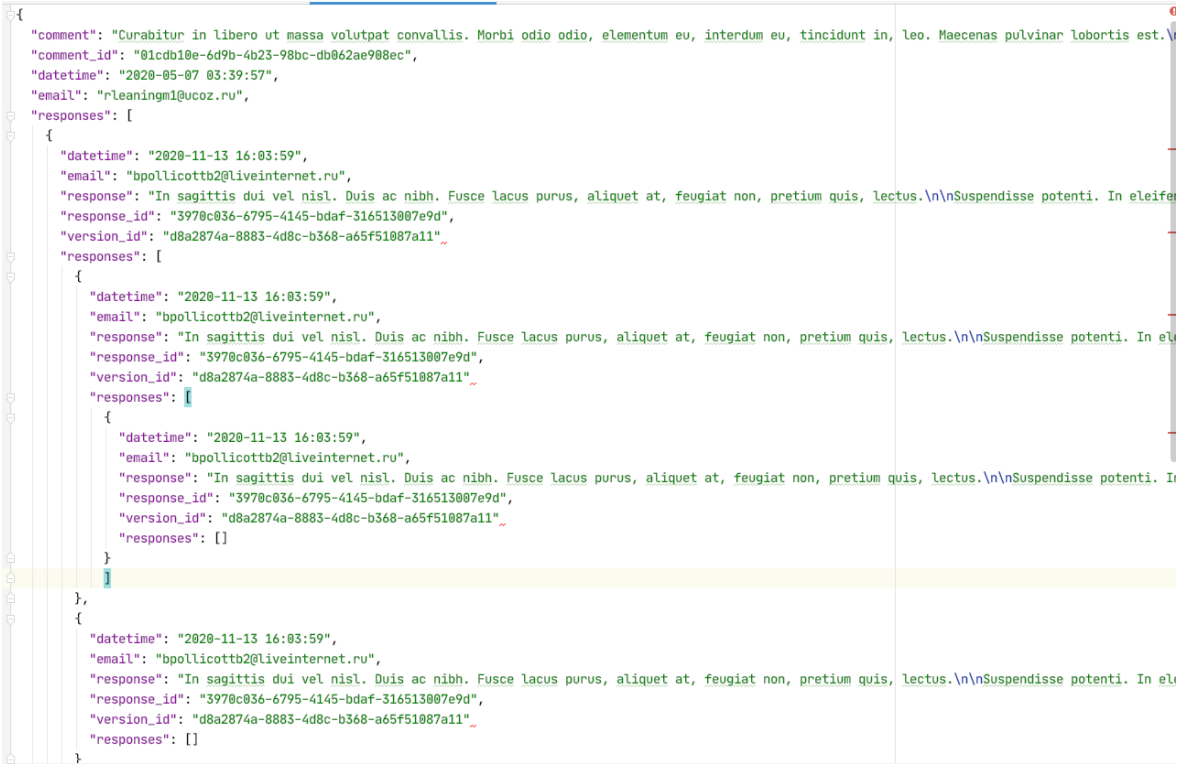
Conditional expression

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

Update expression

- UpdateItem: set Replies = Replies + :num

Document Databases – Semi-Structured Schema



The relation model has difficulties with some types of data:

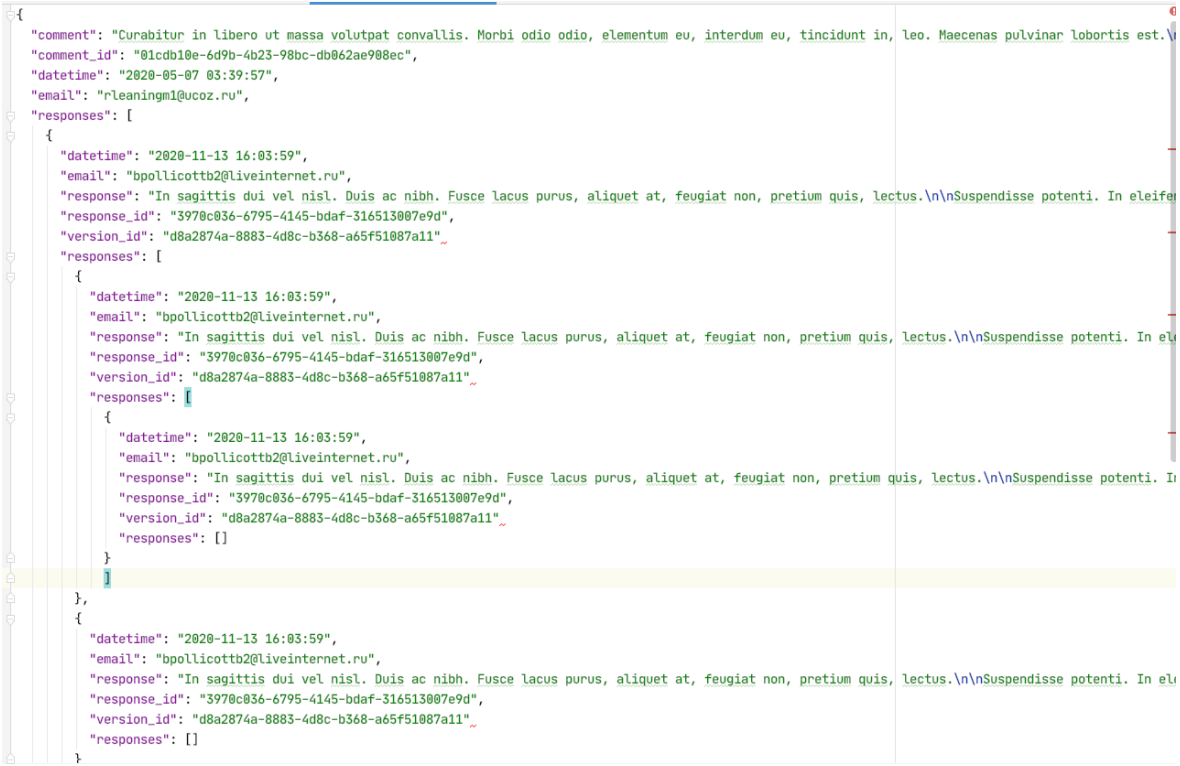
- Columns that are lists or maps
- Nesting – things inside things
- Discover you need a modified schema when you get a piece of data.
- etc.

All of these are common for documented like data:

- Threaded discussions.
- Documents.
- … …

- Document DBs emerged to handle these scenarios.

DynamoDB and Documents



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

DynamoDB Summary

- There are some very interesting properties.
- I will provide some code examples.
- I am trying to decide which cloud DBs you should use, if any.