E6156 – Topics in SW Engineering: Cloud Computing

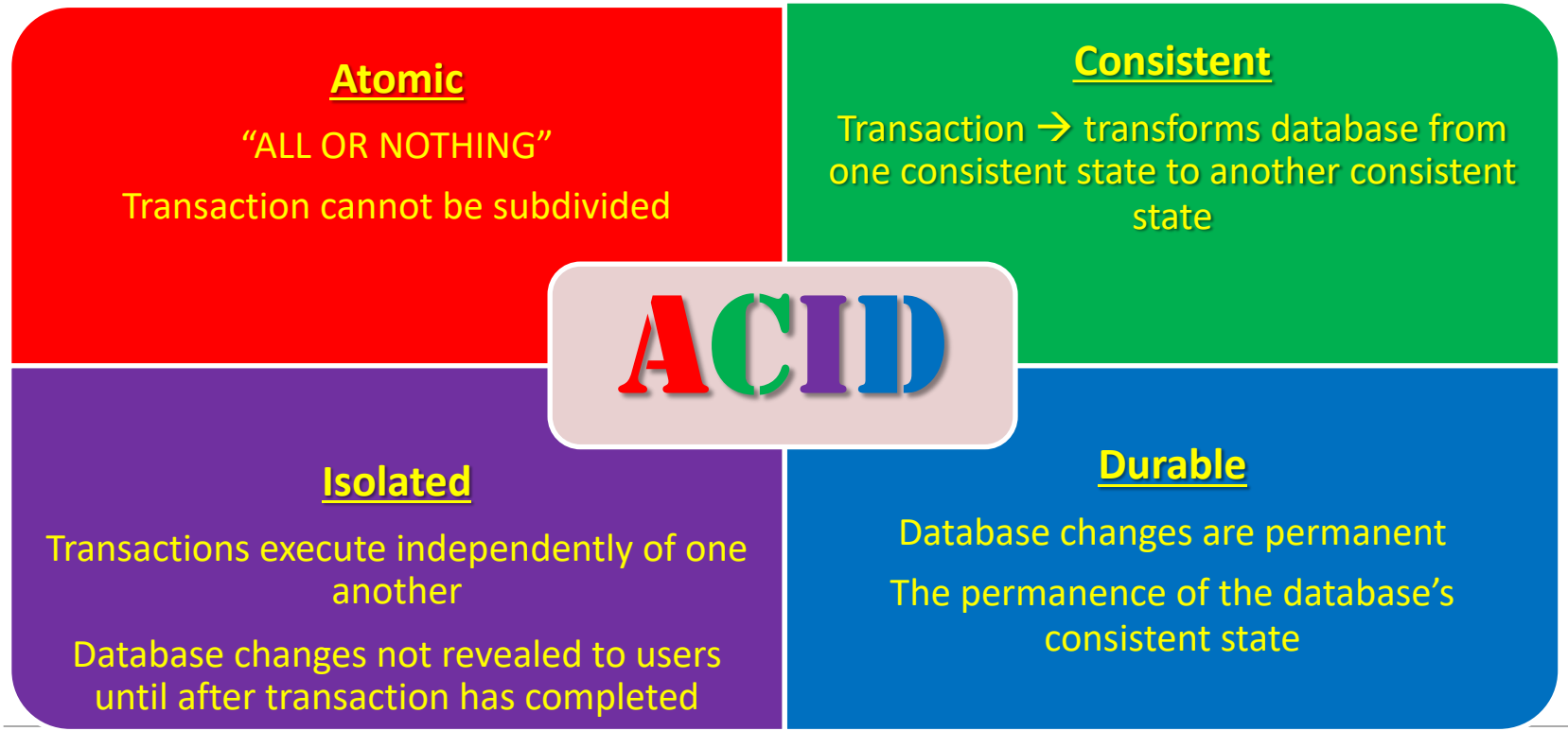
Lecture 7: CloudFront, NoSQL, Certificates, DNS, Service Composition

Come Code Walkthroughs
Smarty – Samples (Python, UI)
Notification
Lambda Functions ← SNS

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

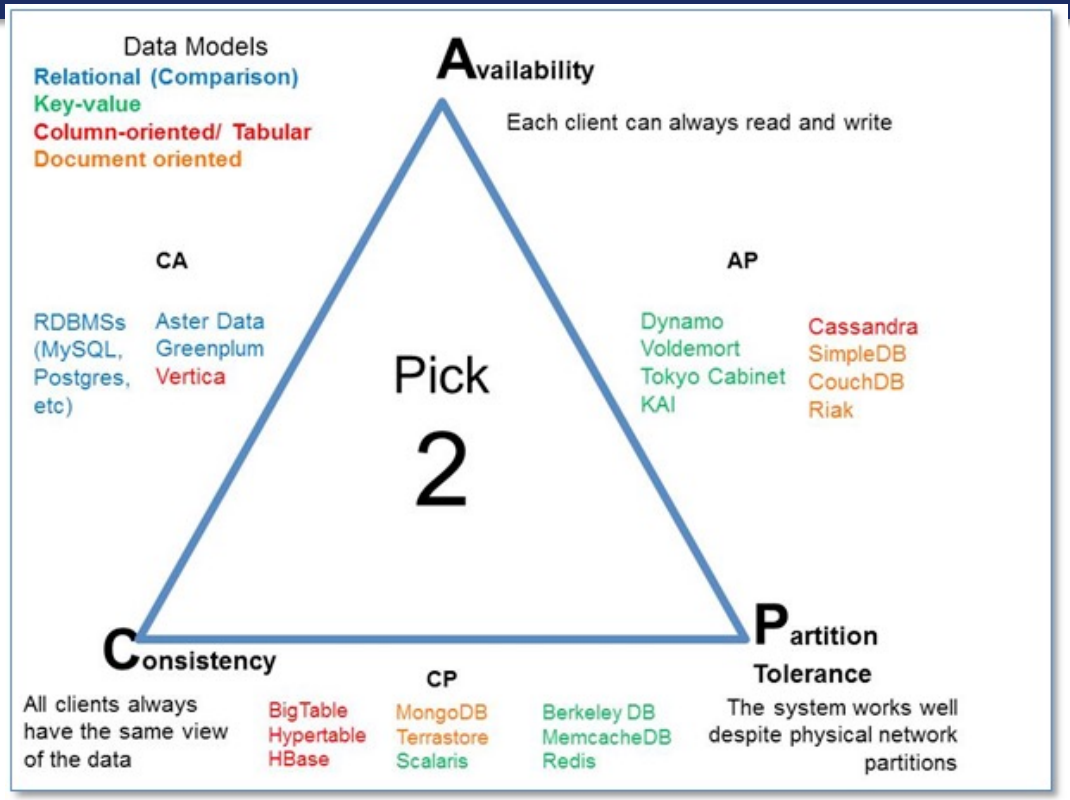
  Every read receives the most recent write or an error.

- 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.

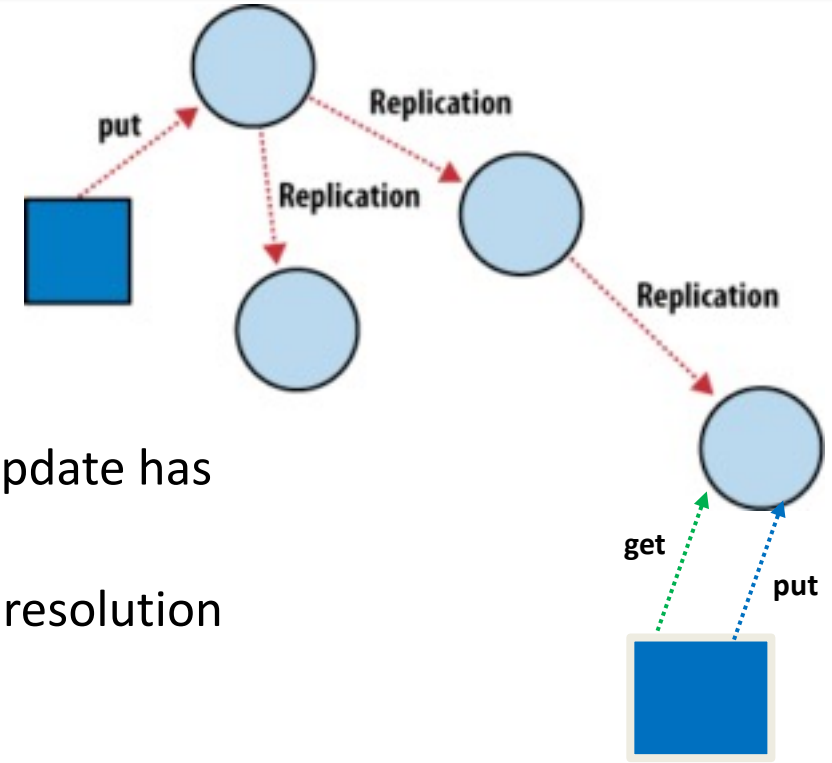
BASE

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)

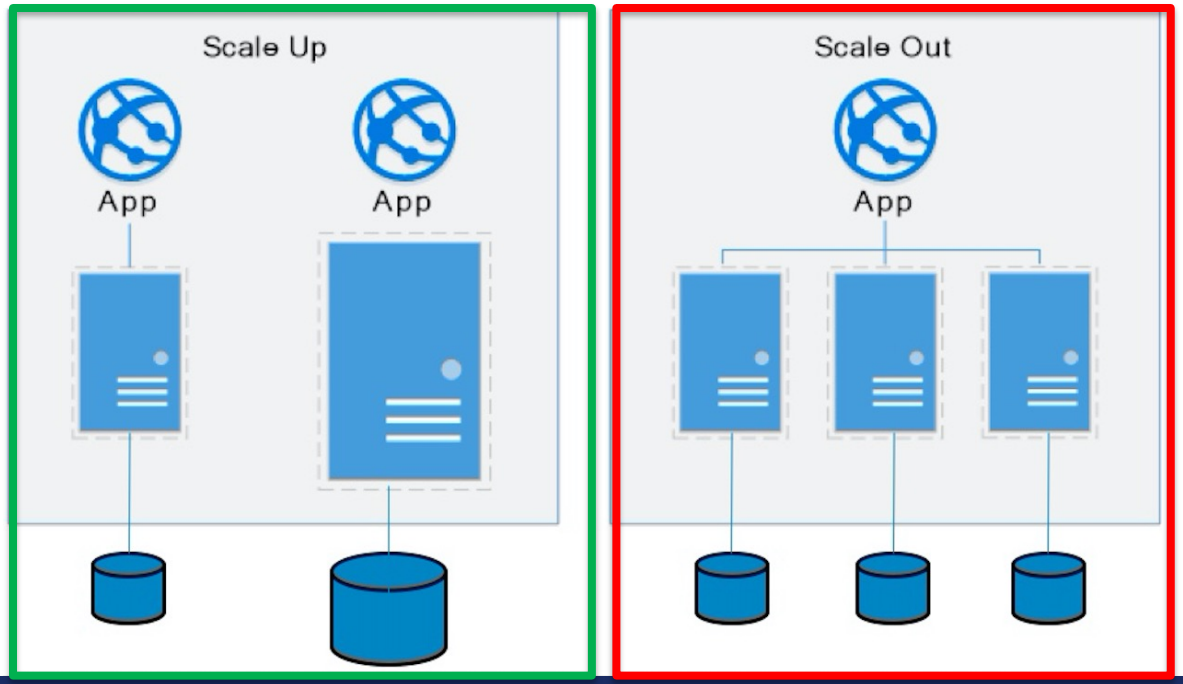
ACID (relational)	BASE (NoSQL)
Strong consistency	Weak consistency
Isolation	Last write wins (Or other strategy)
Transaction	Program managed
Robust database	Simple database
Simple code (SQL)	Complex code
Available and consistent	Available and partition-tolerant
Scale-up (limited)	Scale-out (unlimited)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

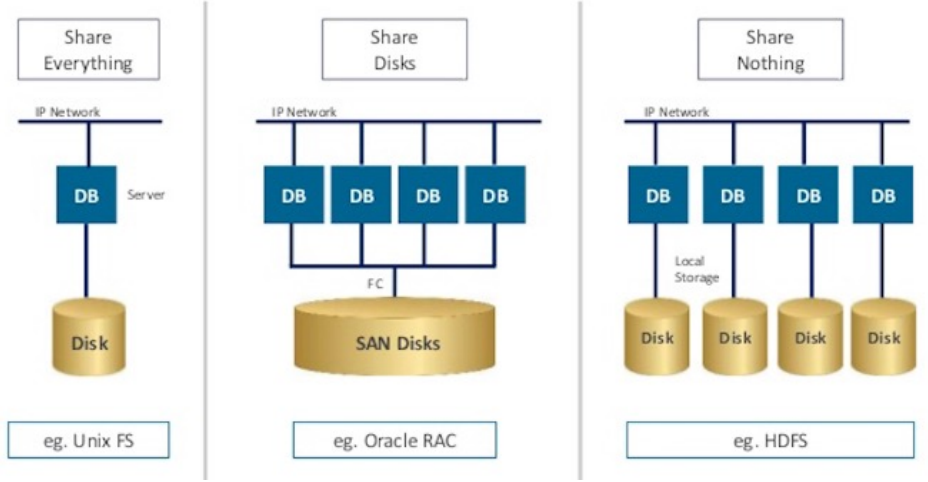
Add another system.



- 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, … …

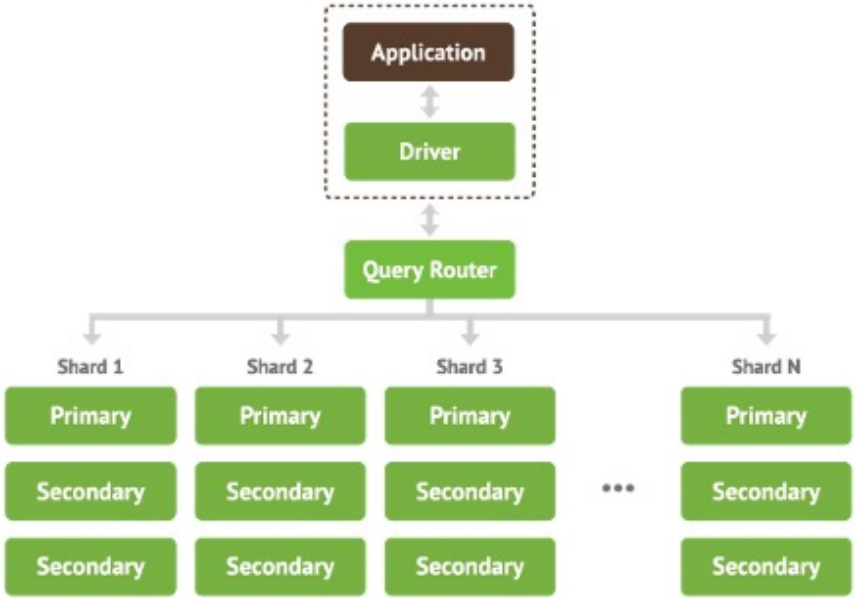
Disk Architecture for Scale-Out

- 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.

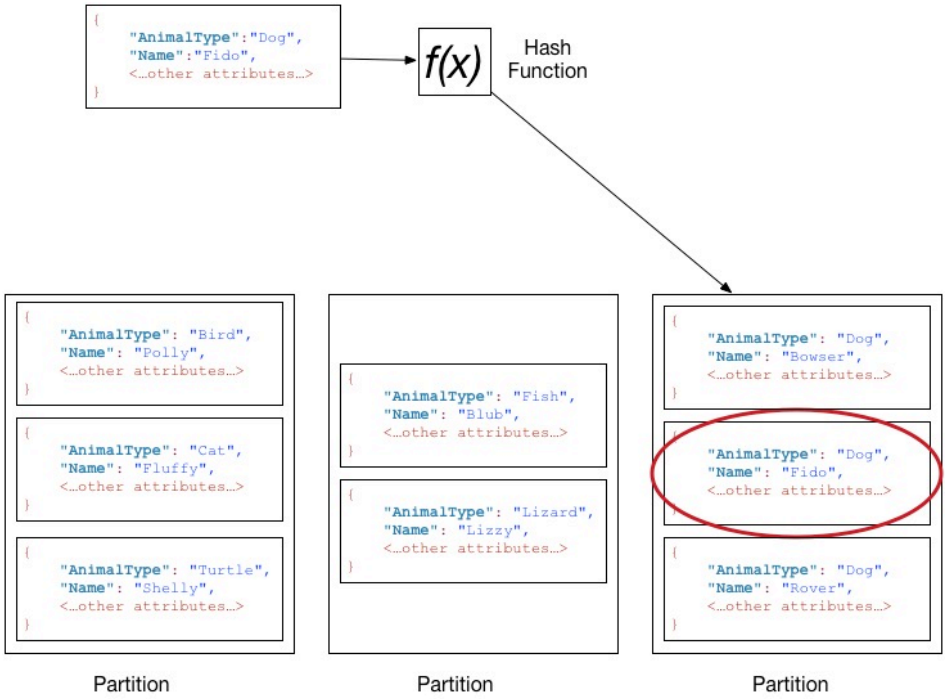


Shared Nothing, Scale-Out

MongoDB Sharding

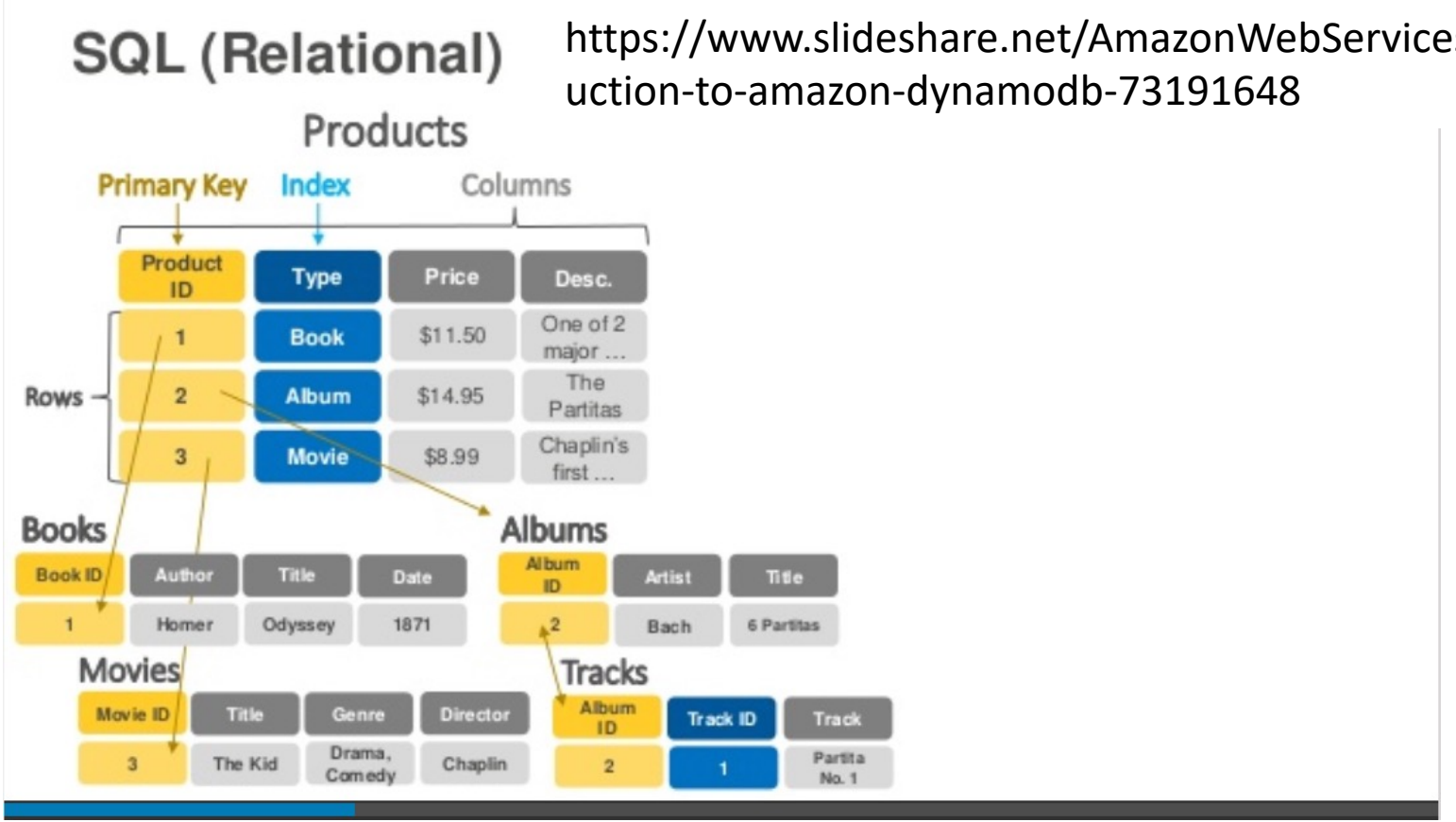


DynamoDB Partitioning



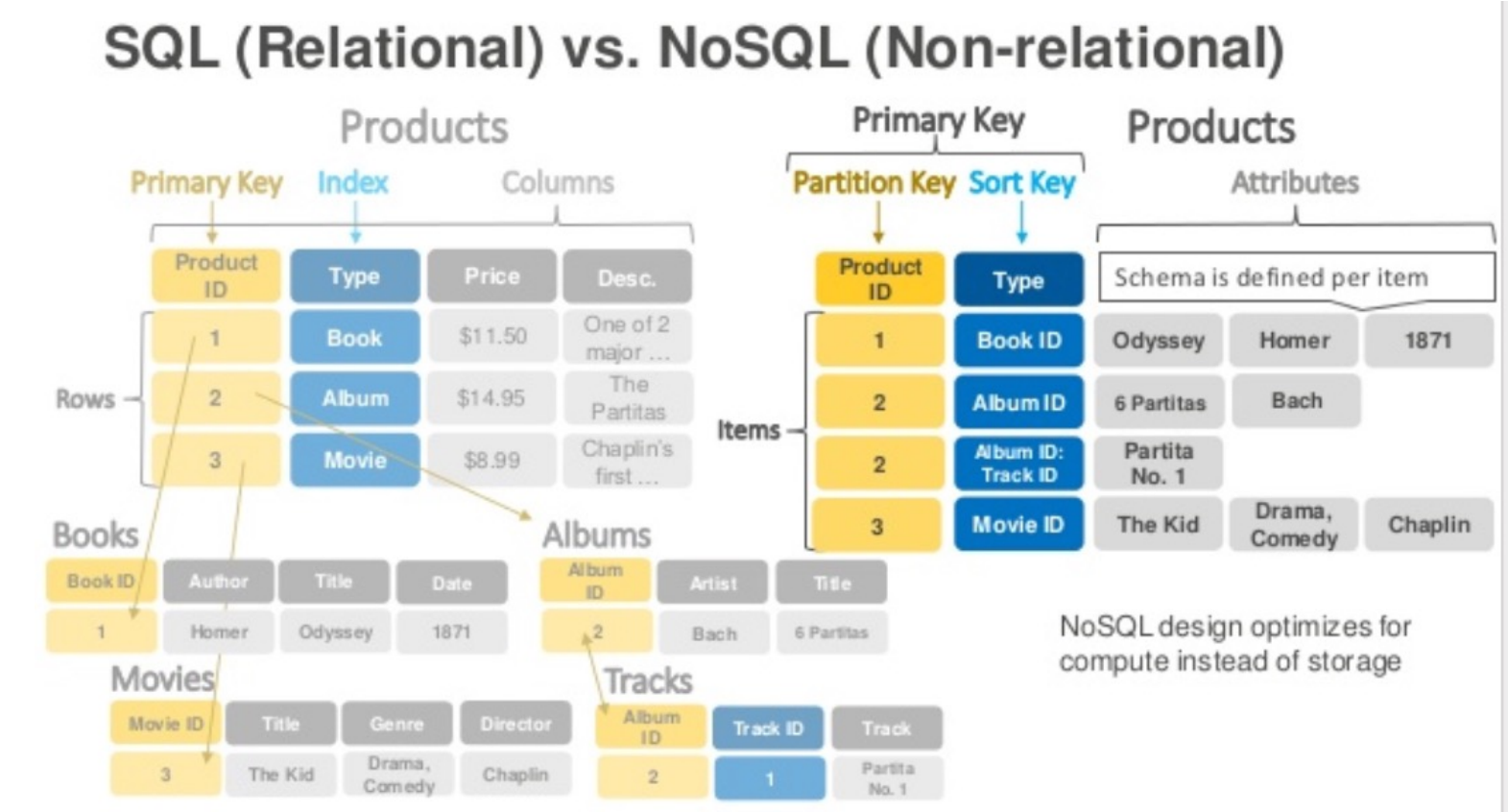
DynamoDB

DynamoDB – Relational



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

Dynamo DB – Relational



DynamoDB Benefits

DynamoDB Benefits


Fully managed


Fast, consistent performance

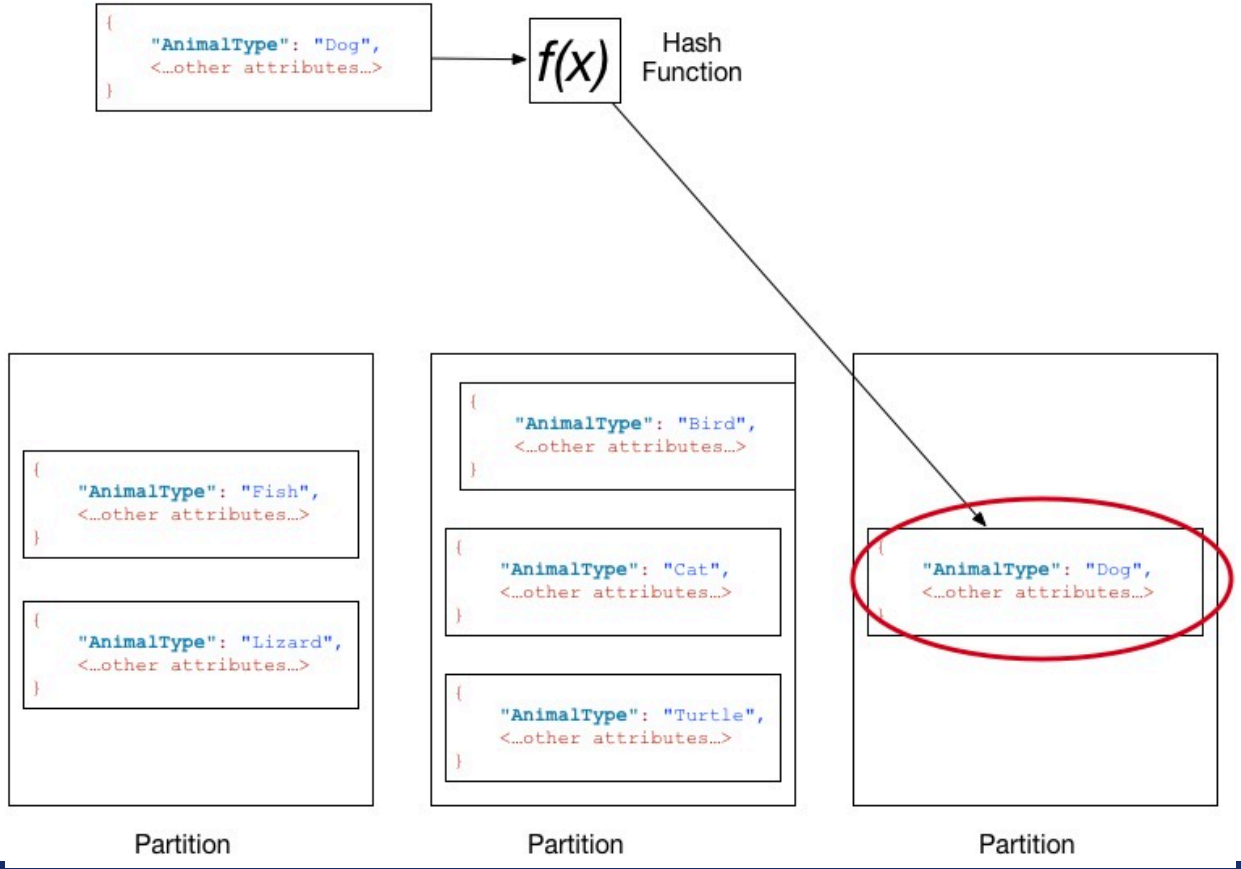

Highly scalable


Flexible

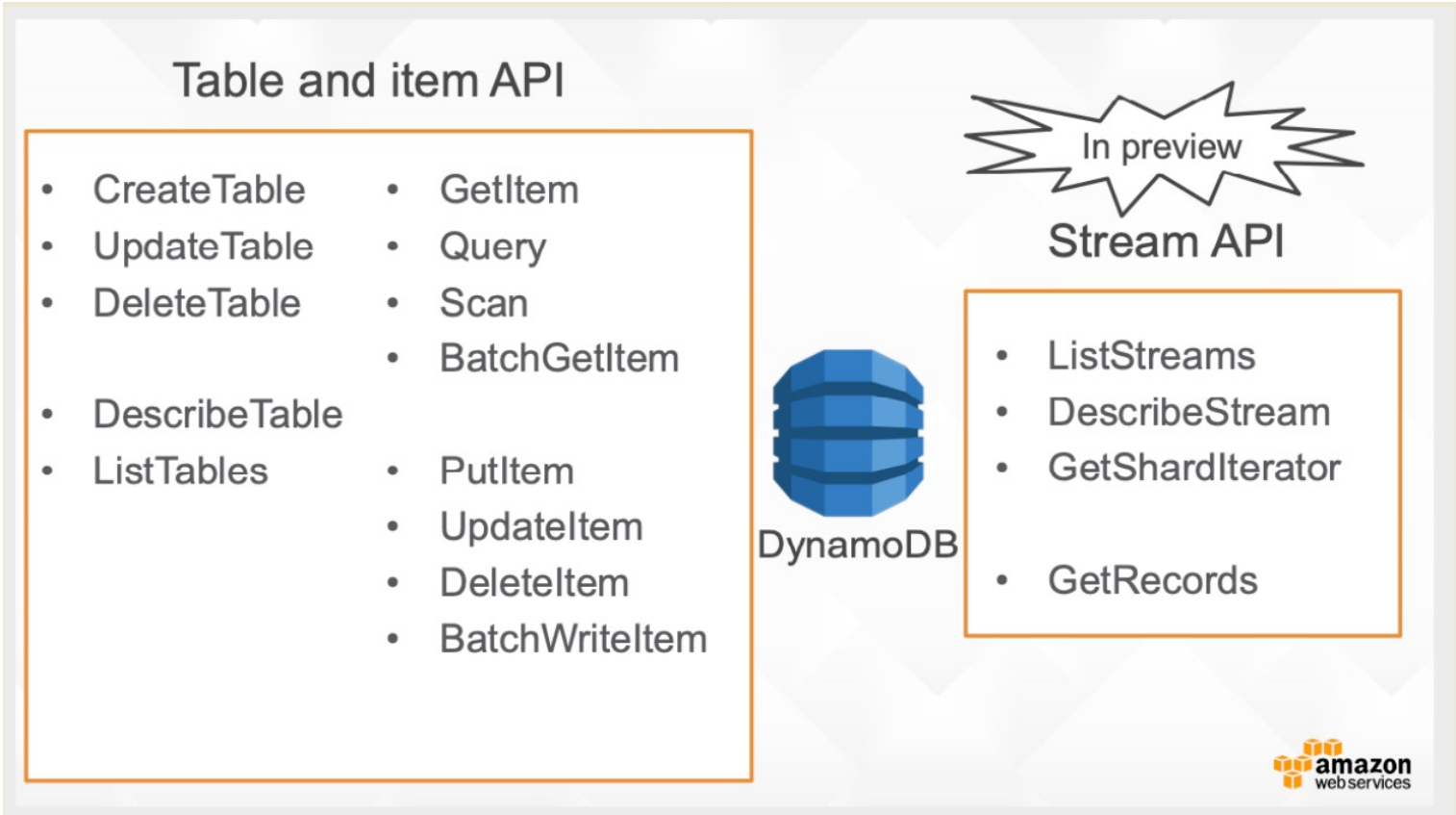

Event-driven programming


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)

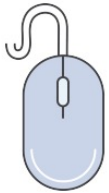
Conditional expression

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

Update expression

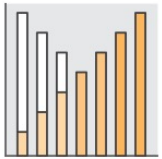
- UpdateItem: set Replies = Replies + :num

Amazon DynamoDB

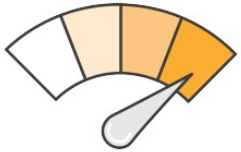


Fully Managed NoSQL

Document or Key-Value



Scales to Any Workload

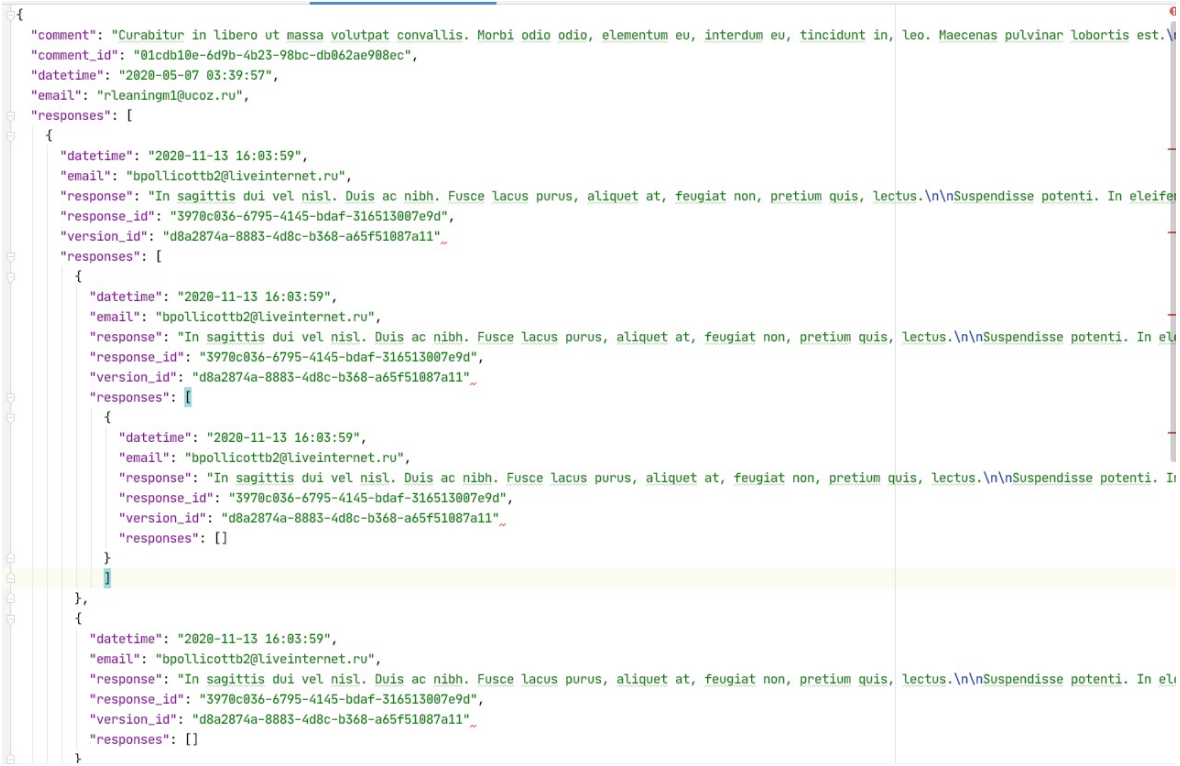


Fast and Consistent

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

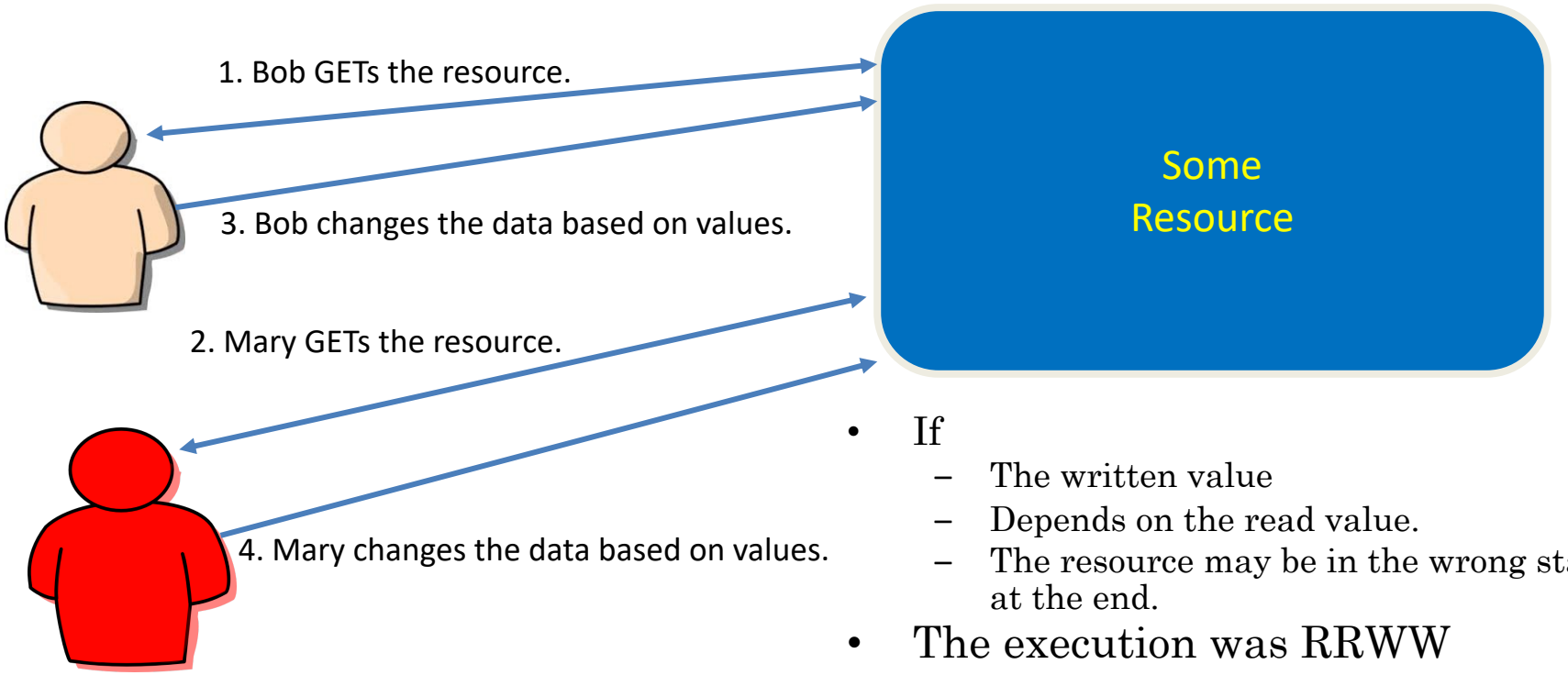
- Achieves much greater scalability and performance than RDBMs
- Does not support some RDBMs capabilities:
  - Referential integrity.
  - JOIN
  - Queries limited to key fields.
  - Non-key field queries are always scans.
- Data model better fit for *semi-structured* data models and good fit for JSON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

ETag, Conditional Update

REST and Isolation: Read then Update Conflict



- If
  - The written value
  - Depends on the read value.
  - The resource may be in the wrong state at the end.
- The execution was RRWW
- But the intent was RWRW.

Isolation/Concurrency Control

This requires conversation state, which would violate the REST Stateless principle.

- There are two basic approaches to implementing isolation
  - Locking/Pessimistic, e.g. cursor isolation
  - Optimistic: Before committing, each transaction verifies that no other transaction has modified the data it has read.
- How does this work in REST?
  - The server maintains an ETag (Entity Tag) for each resource.
  - Every time a resource's state changes, the server computes a new ETag.
  - The server includes the ETag in the header when returning data to the client.
  - The server may *optionally* maintain and return "Last Modified" time for resources.
- Semantics on updates
  - If-Match – value of a previous calls ETag response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412.
  - If-Modified-Since – value of a previous Last-Modified response used with a GET. Server should only provide a response if the resource was modified since the timestamp submitted. Use in conjunction with If-None-Match in case the change occurred within the same second. Otherwise provide a 304.
  - If-None-Match – value of a previous calls ETag response used with a GET. Server should only provide a response if the ETag doesn't match, i.e. the resource has been altered. Otherwise provide a 304.
  - If-Unmodified-Since – value of a previous Last-Modified response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412

Conditional Operations

What are DynamoDB Condition Expressions?

Before we learn the specifics of DynamoDB Condition Expressions, let's learn what they are and why you would want to use them.

A `ConditionExpression` is an optional parameter that you can use on write-based operations. If you include a Condition Expression in your write operation, it will be evaluated *prior to* executing the write. If the Condition Expression evaluates to false, the write will be aborted.

This can be useful in a number of common patterns, such as:

- Ensuring uniqueness,
- Validating business rules, and
- Confirming existance.

By using Condition Expressions, you can reduce the number of trips you make to DynamoDB and avoid race conditions when multiple clients are writing to DynamoDB at the same time.

By using Condition Expressions, you can reduce the number of trips you make to DynamoDB and avoid race conditions when multiple clients are writing to DynamoDB at the same time.

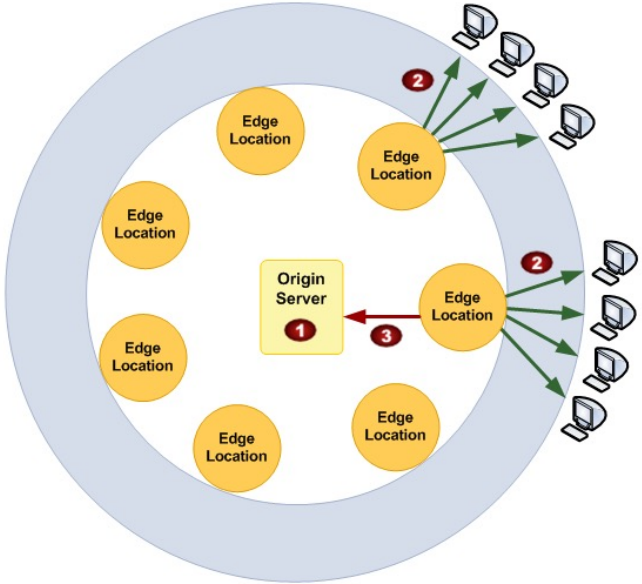
Condition Expressions can be used in the following write-based API operations:

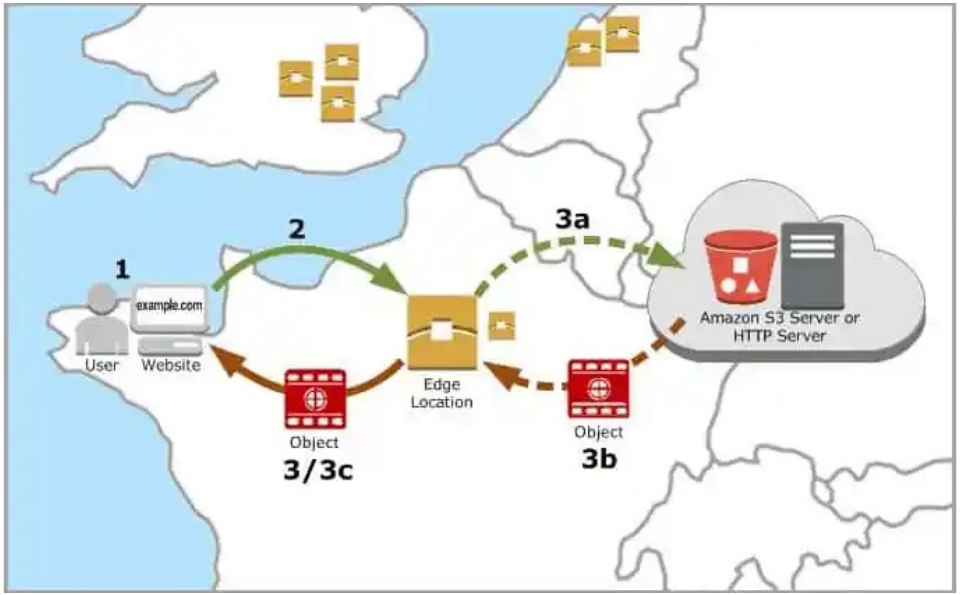
- PutItem
- UpdateItem
- DeleteItem
- TransactWriteItems

Hands-On CloudFront
DNS, Certificate, HTTPS

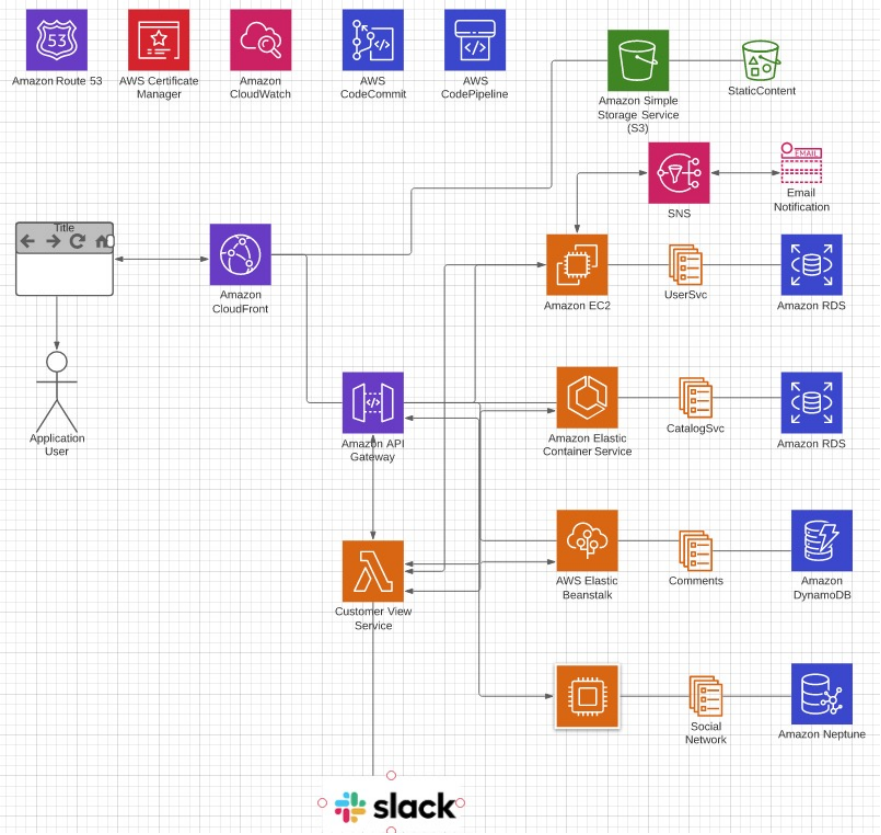
CloudFront

"A content delivery network, or content distribution network (CDN), is a geographically distributed network of proxy servers and their data centers. The goal is to provide high availability and performance by distributing the service spatially relative to end users." (https://en.wikipedia.org/wiki/Content_delivery_network)





Overall Architecture



- There may be multiple
  - Microservices
  - Web UI projects/content
- FrontPage
  - Multiple S3 content → one site.
  - Forward API requests to API GW
- API GW manages/monitors APIs
- Networking/HTTPS requires
  - Domain, DNS
  - Certificates