DynamoDB

The art of single table design

Diachenko Illia

2022

Destiny developer Bungie publisher Activision

GameWallpapers.com hosted by JTLnet.com
Please do not post this wallpaper online. Wallpaper © GameWallpapers.com. Artwork © developer, publisher cum suis. No part of this wallpaper may be reproduced or transmitted in any form or by any means without prior written consent of GameWallpapers.com.

AGENDA

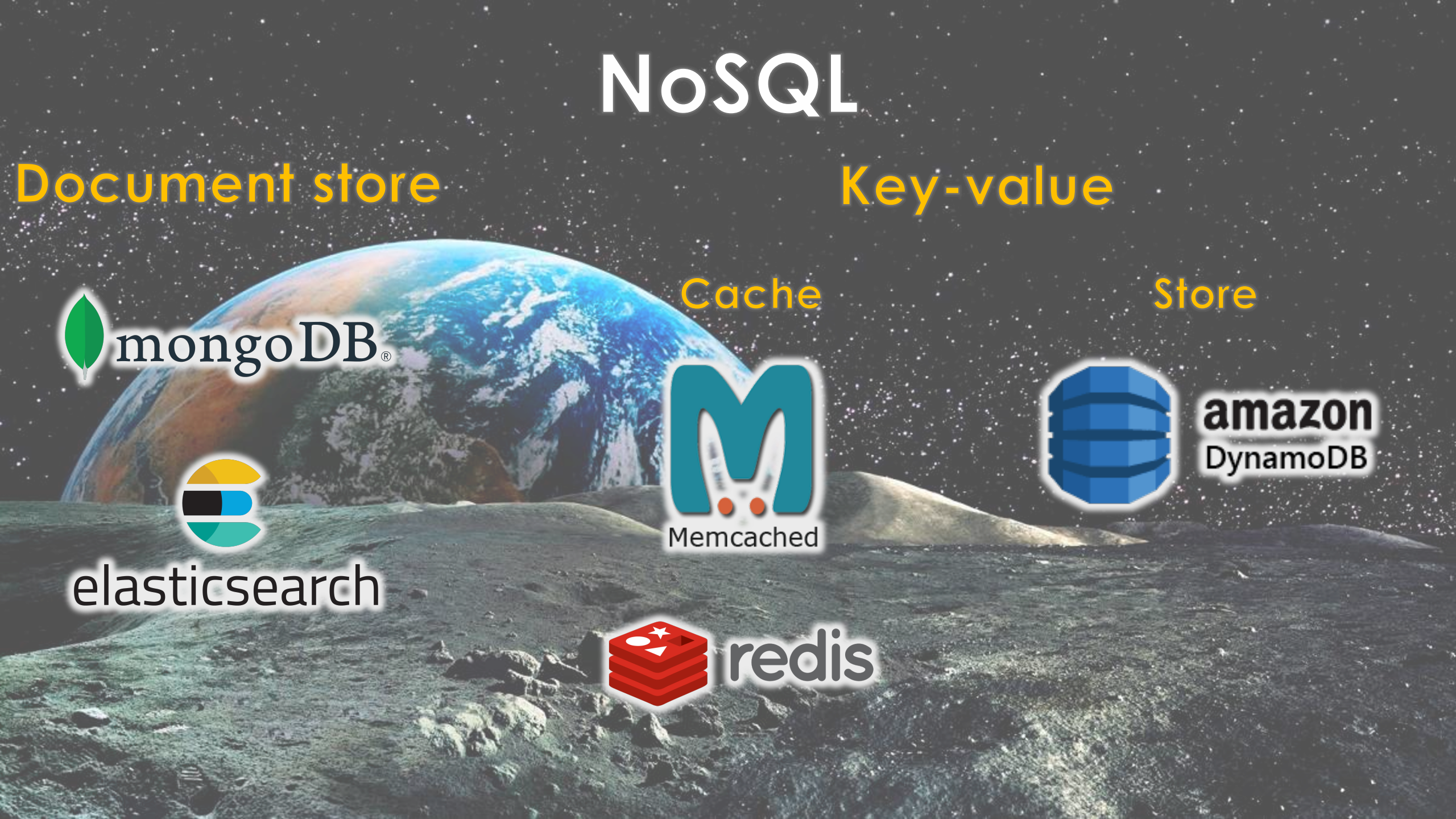
- NoSQL world overview
- CAP Theorem
- Pricing
- DynamoDB Data modelling
- Partitioning and primary key
- Indexes
- Single table design
- ACID Transactions
- Best practices
- Demo

Amazon DynamoDB is a fully managed proprietary NoSQL database service that supports key–value and document data structures and is offered by Amazon.com as part of the Amazon Web Services portfolio. DynamoDB exposes a similar data model to and derives its name from Dynamo, but has a different underlying implementation. Dynamo had a multi-leader design requiring the client to resolve version conflicts and DynamoDB uses synchronous replication across multiple data centers for high durability and availability. DynamoDB was announced by Amazon CTO Werner Vogels on January 18, 2012, and is presented as an evolution of Amazon SimpleDB.

Destiny developer Bungie publisher Activision
GameWallpapers.com hosted by JTLnet.com

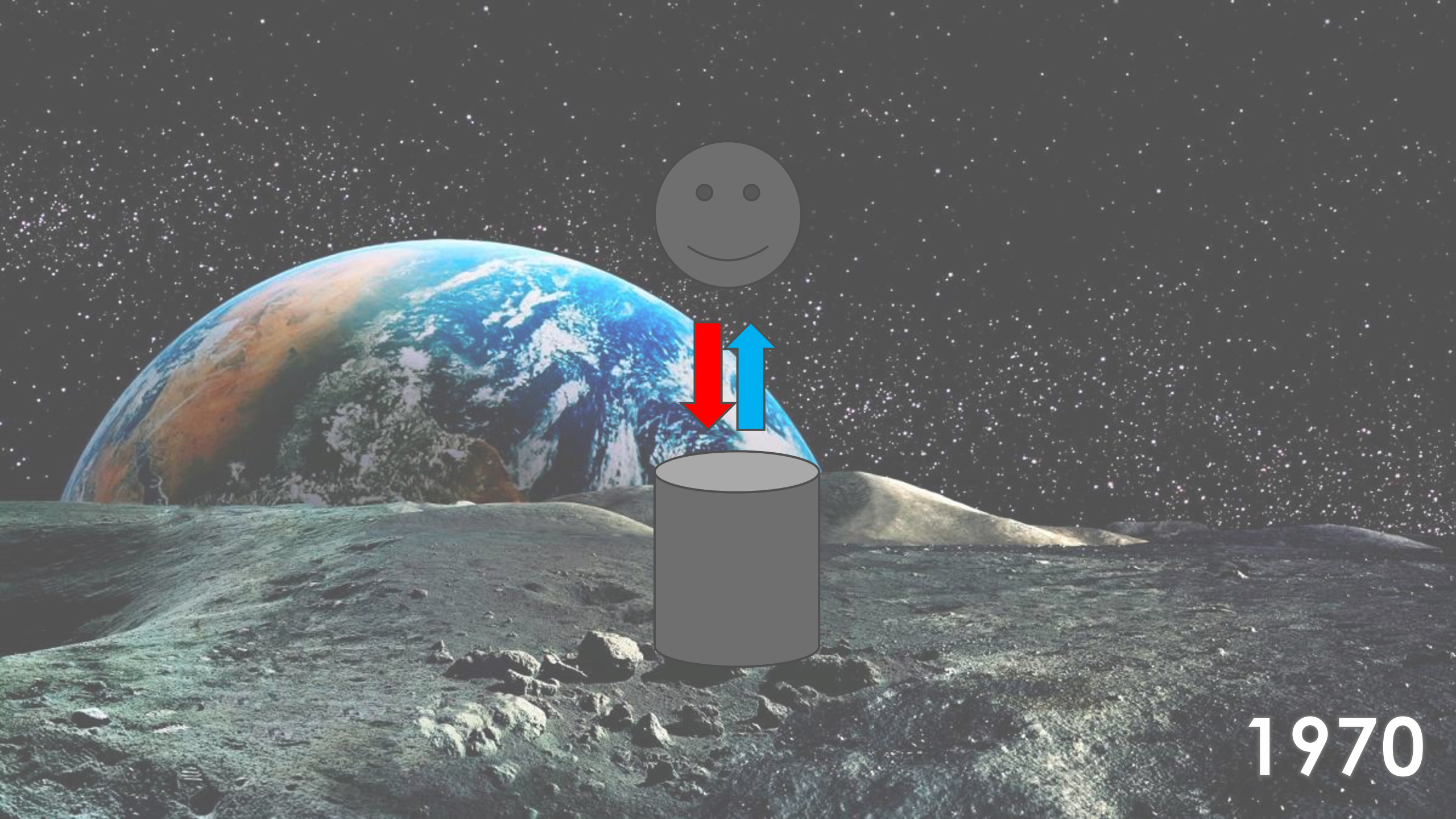
DIFFERENCES BETWEEN SQL AND NOSQL

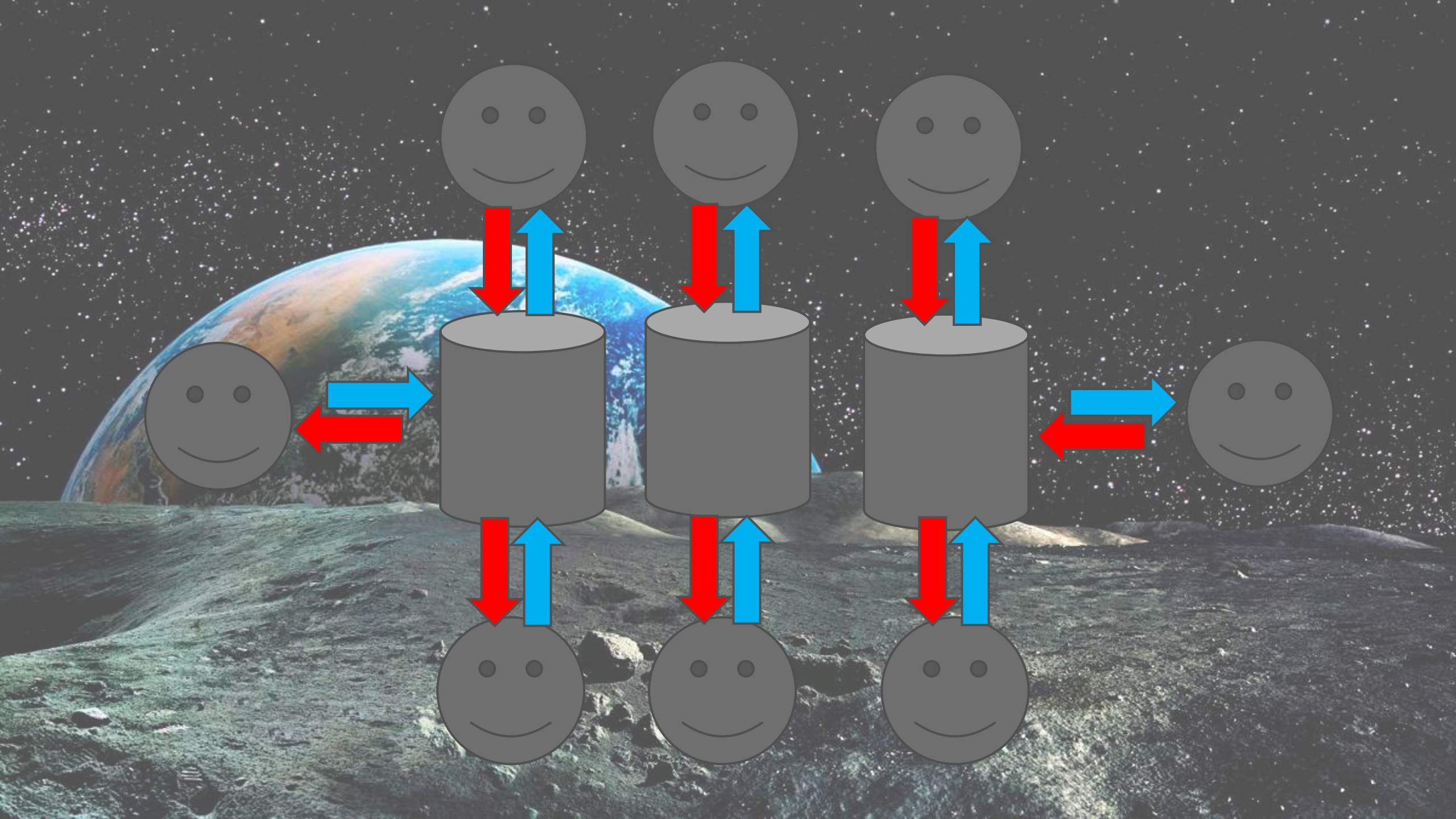
	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling.
Primary Purpose	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
ACID Transactions	Supported	Most do not support multi-record ACID transactions
Joins	Typically required	Typically not required
Data to Object Mapping	Requires ORM (object-relational mapping)	Many do not require ORMs.


NoSQL
Document store
Key-value
Cache
Store
mongoDB
Memcached
amazon
DynamoDB
elasticsearch
redis

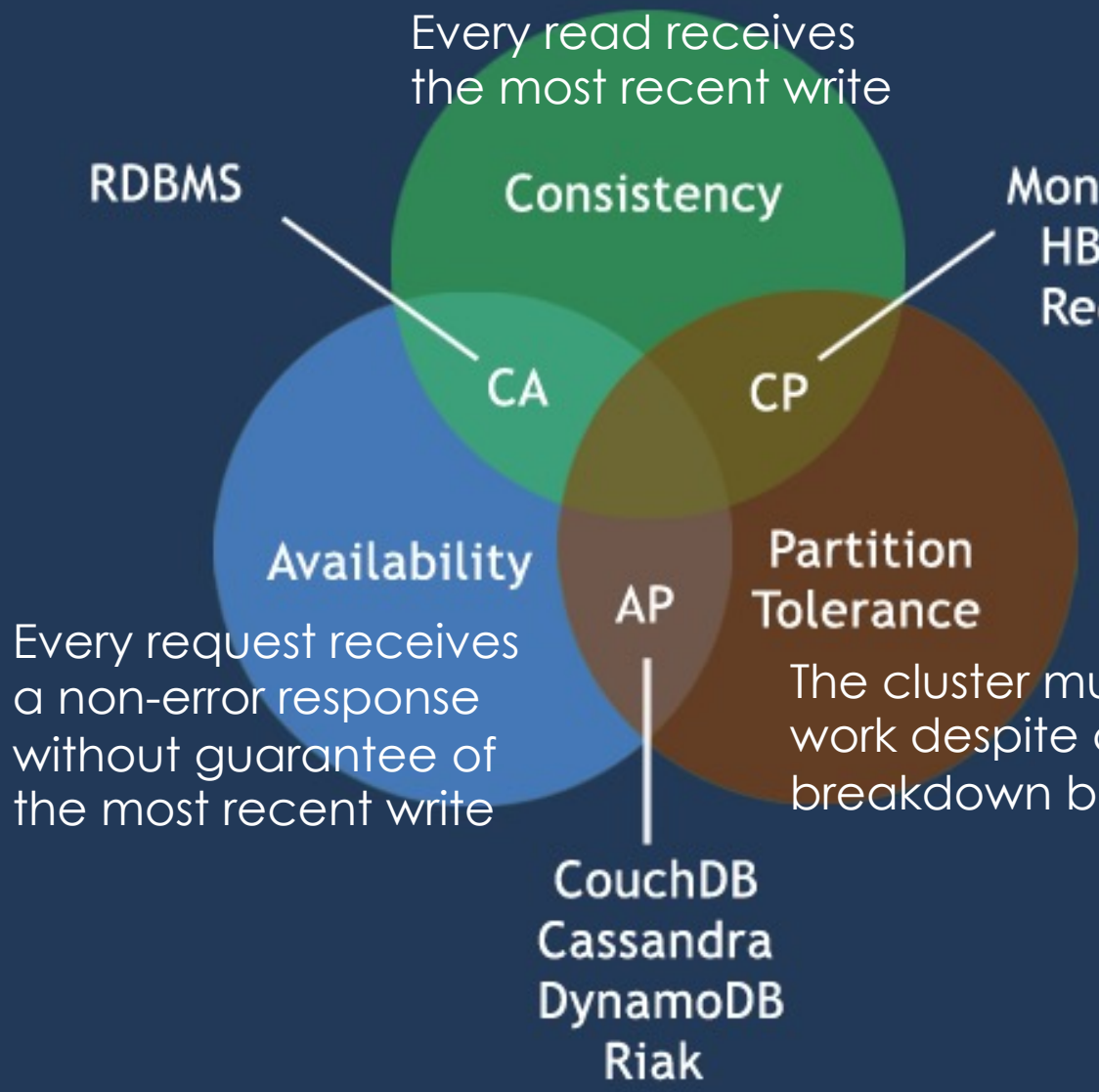
NoSQL

Type	Notable examples of this type
Key–value cache	Apache Ignite, Couchbase, Coherence, eXtreme Scale, Hazelcast, Infinispan, Memcached, Redis, Velocity
Key–value store	Azure Cosmos DB, ArangoDB, Amazon DynamoDB, Aerospike, Couchbase
Key–value store (eventually consistent)	Azure Cosmos DB, Oracle NoSQL Database, Riak, Voldemort
Key–value store (ordered)	FoundationDB, InfinityDB, LMDB, MemcacheDB
Tuple store	Apache River, GigaSpaces, Tarantool, TIBCO ActiveSpaces, OpenLink Virtuoso
Triplestore	AllegroGraph, MarkLogic, Ontotext-OWLIM, Oracle NoSQL database, Profium Sense, Virtuoso Universal Server
Object database	Objectivity/DB, Perst, ZopeDB, db4o, GemStone/S, InterSystems Caché, JADE, ObjectDatabase++, ObjectDB, ObjectStore, ODABA, Realm, OpenLink Virtuoso, Versant Object Database, ZODB
Document store	Azure Cosmos DB, ArangoDB, BaseX, Clusterpoint, Couchbase, CouchDB, DocumentDB, eXist-db, IBM Domino, MarkLogic, MongoDB, RavenDB, Qizx, RethinkDB, Elasticsearch, OrientDB
Wide Column Store	Azure Cosmos DB, Amazon DynamoDB, Bigtable, Cassandra, Google Cloud Datastore, HBase, Hypertable, ScyllaDB
Native multi-model database	ArangoDB, Azure Cosmos DB, OrientDB, MarkLogic, Apache Ignite,[22][23] Couchbase, FoundationDB, MarkLogic, Oracle Database
Graph database	Azure Cosmos DB, AllegroGraph, ArangoDB, InfiniteGraph, Apache Giraph, MarkLogic, Neo4J, OrientDB, Virtuoso
Multivalue database	D3 Pick database, Extensible Storage Engine (ESE/NT), InfinityDB, InterSystems Caché, jBASE Pick database, mvBase Rocket Software, mvEnterprise Rocket Software, Northgate Information Solutions Reality (the original Pick/MV Database), OpenQM, Revelation Software's OpenInsight (Windows) and Advanced Revelation (DOS), UniData Rocket U2, UniVerse Rocket U2


1970



CAP Theorem



2000

Rank Feb 2020	Rank Jan 2020	Rank Feb 2019	DBMS	Database Model	Score Feb 2020	Score Jan 2020	Score Feb 2019
1.	1.	1.	Oracle	Relational, Multi-model	1344.75	-1.93	+80.73
2.	2.	2.	MySQL	Relational, Multi-model	1267.65	-7.00	+100.36
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1093.75	-4.80	+53.69
4.	4.	4.	PostgreSQL	Relational, Multi-model	506.94	-0.25	+33.38
5.	5.	5.	MongoDB	Document, Multi-model	433.33	+6.37	+38.24
6.	6.	6.	IBM Db2	Relational, Multi-model	165.55	-3.15	-13.87
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	152.16	+0.72	+6.91
8.	8.	↓ 7.	Redis	Key-value, Multi-model	151.42	+2.67	+1.97
9.	9.	9.	Microsoft Access	Relational	128.06	-0.52	-15.96
10.	10.	10.	SQLite	Relational	123.36	+1.22	-2.81
11.	11.	11.	Cassandra	Wide column	120.36	-0.31	-3.02
12.	12.	↑ 13.	Splunk	Search engine	88.77	+0.10	+5.96
13.	13.	↓ 12.	MariaDB	Relational, Multi-model	87.34	-0.11	+3.91
14.	14.	↑ 15.	Hive	Relational	83.53	-0.71	+11.25
15.	15.	↓ 14.	Teradata	Relational, Multi-model	76.81	-1.48	+0.84
16.	16.	↑ 21.	Amazon DynamoDB	Multi-model	62.14	+0.12	+7.19
17.	17.	↓ 16.	Solr	Search engine	56.16	-0.41	-4.81
18.	↑ 19.	↑ 19.	SAP HANA	Relational, Multi-model	54.97	+0.28	-1.58
19.	↓ 18.	↓ 18.	FileMaker	Relational	54.88	-0.23	-2.91
20.	↑ 21.	↓ 17.	HBase	Wide column	52.95	-0.39	-7.33

MongoDB vs. DynamoDB

- MongoDB is more popular than DynamoDB
- DynamoDB has a more hands-off operations model than MongoDB
- At high scale, both have very similar data modelling principles (pre-joins)
- At lower scale, MongoDB has a more flexible data model
- DynamoDB's connection and security model make it a popular choice for use with serverless compute like AWS Lambda

0.
PREPARE ENVIRONMENT

DATA MODEL

```
[
  {
    "Title": "Avatar",
    "Year": "2009",
    "Rating": "7.9"
  },
  {
    "Title": "I Am Legend",
    "Year": "2007",
    "Rating": "7.2"
  }
]
```

TABLE – STORE OF DATA

ITEM – GROUP OF ATTRIBUTES

ATTRIBUTES – SCALAR AND COMPOSITE DATA TYPES.

Title	Year	Rating
Avatar	2009	7.9
I Am Legend	2007	7.2

SCHEMALESS

Title	Year	Rating	Type	Runtime	Plot
Avatar	2009	7.9	Movie	162	
I Am Legend	2007	7.2	Movie	101	
Breaking Bad		9.5	TV show		A high school chemistry teacher diagnosed with inoperable lung cancer turns to manufacturing and selling methamphetamine in order to secure his family's financial future.
The Avengers	2012	8.1	Movie	143	Earth's mightiest heroes must come together and learn to fight as a team if they are to stop the mischievous Loki and his alien army from enslaving humanity.
Luke Cage			TV show		

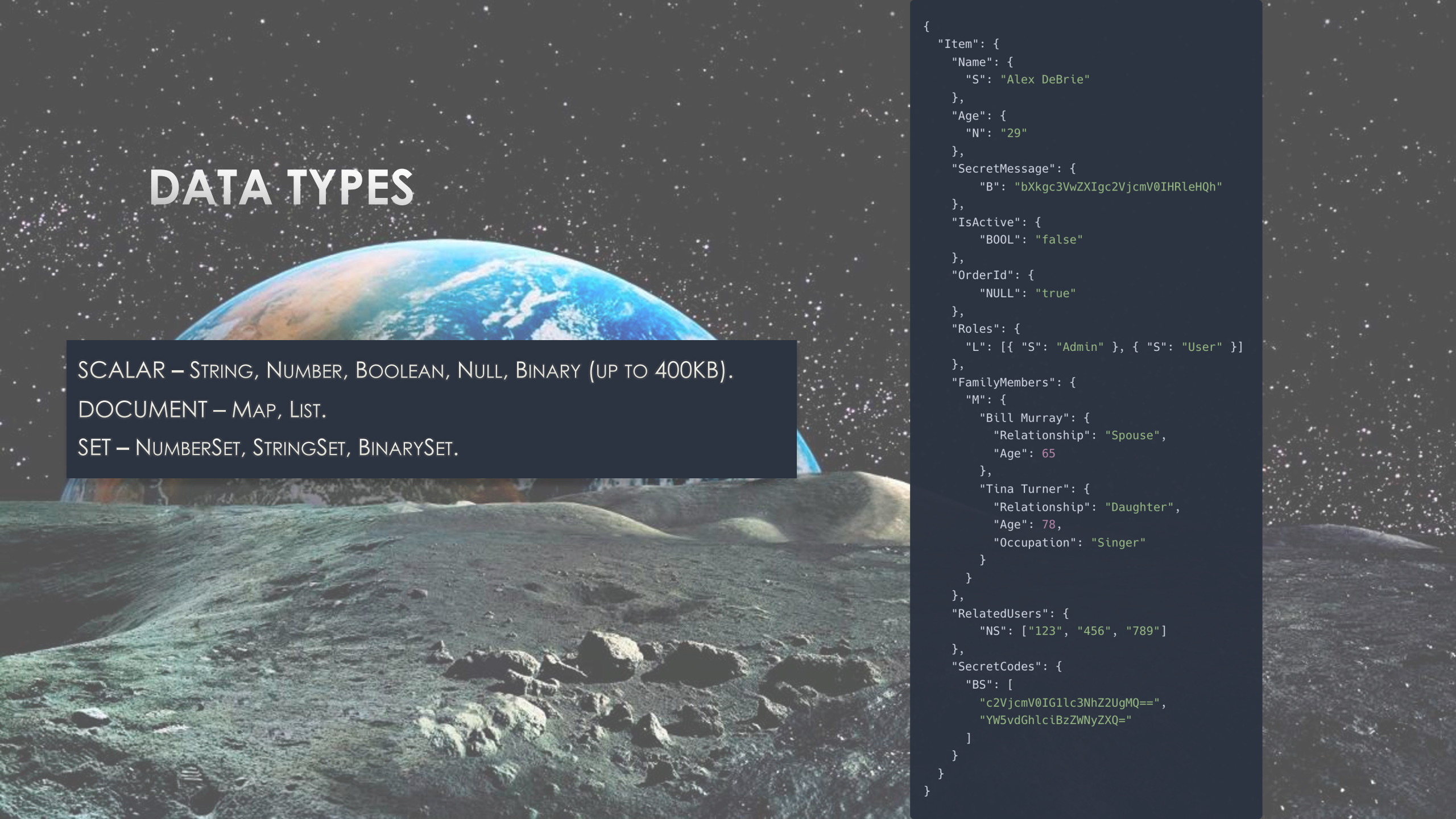
1. DATA MODELLING

DATA TYPES

SCALAR – String, Number, Boolean, Null, Binary (up to 400KB).

DOCUMENT – Map, List.

SET – NumberSet, StringSet, BinarySet.



```
{
  "Item": {
    "Name": {
      "S": "Alex DeBrie"
    },
    "Age": {
      "N": "29"
    },
    "SecretMessage": {
        "B": "bXkgc3VwZXIgc2VjcmV0IHRleHQh"
    },
    "IsActive": {
        "BOOL": "false"
    },
    "OrderId": {
        "NULL": "true"
    },
    "Roles": {
      "L": [{ "S": "Admin" }, { "S": "User" }]
    },
    "FamilyMembers": {
      "M": {
        "Bill Murray": {
          "Relationship": "Spouse",
          "Age": 65
        },
        "Tina Turner": {
          "Relationship": "Daughter",
          "Age": 78,
          "Occupation": "Singer"
        }
      }
    },
    "RelatedUsers": {
        "NS": ["123", "456", "789"]
    },
    "SecretCodes": {
      "BS": [
        "c2VjcmV0IG1lc3NhZ2UgMQ==",
        "YW5vdGhlciBzZWNyZXQ="
      ]
    }
  }
}
```

LIST

```
FavoriteThings: ["Cookies", "Coffee", 3.14159]
```

There are no restrictions on the data types that can be stored in a list element, and the elements in a list element do not have to be of the same type

SET

```
["Black", "Green", "Red"]

[42.2, -19, 7.5, 3.14]

["U3Vubnk=", "UmFpbnk=", "U25vd3k="]
```

Each value within a set must be unique. The order of the values within a set is not preserved.

MAP

```
{
    Day: "Monday",
    UnreadEmails: 42,
    ItemsOnMyDesk: [
        "Coffee Cup",
        "Telephone",
        {
            Pens: { Quantity : 3},
            Pencils: { Quantity : 2},
            Erasers: { Quantity : 1}
        }
    ]
}
```

Maps are ideal for storing JSON documents in DynamoDB.

PRIMARY KEY

UNIQUE IDENTIFIER (STRING, NUMBER, OR BINARY) FOR AN ITEM

MUST BE DEFINED AT THE CREATION OF THE TABLE

MUST BE PROVIDED WHEN INSERTING A NEW ITEM

TWO TYPES:

- PARTITION KEY (ONE ATTRIBUTE)
- PARTITION KEY AND SORT KEY (RANGE KEY)

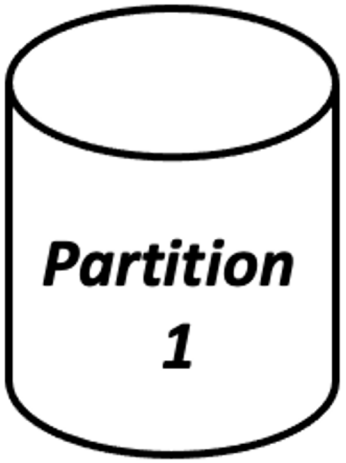
HORIZONTAL SCALING AND THE PARTITION KEY

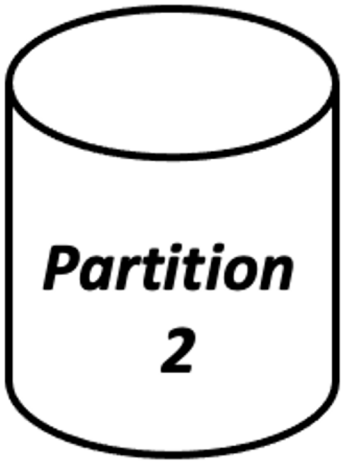
PutItem:

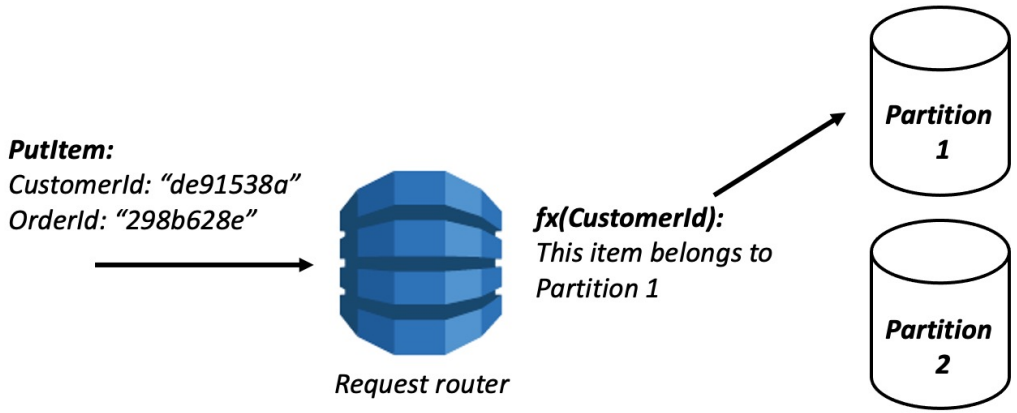
CustomerId: "de91538a"

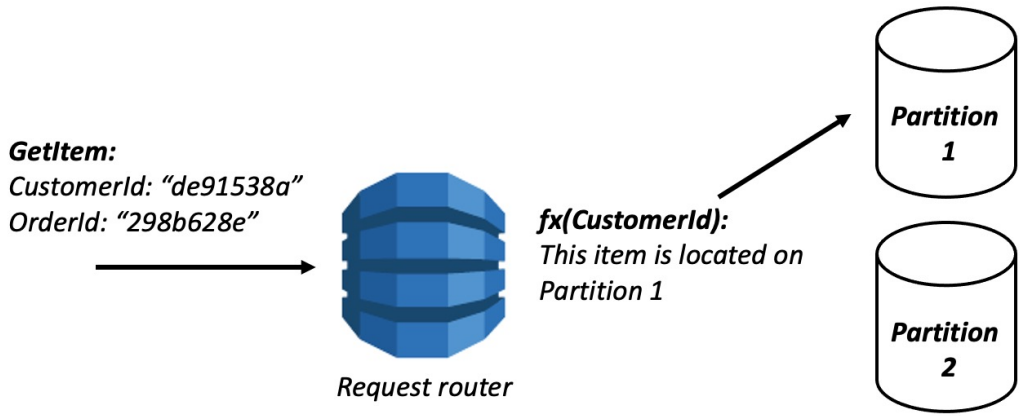
OrderId: "298b628e"

Request router



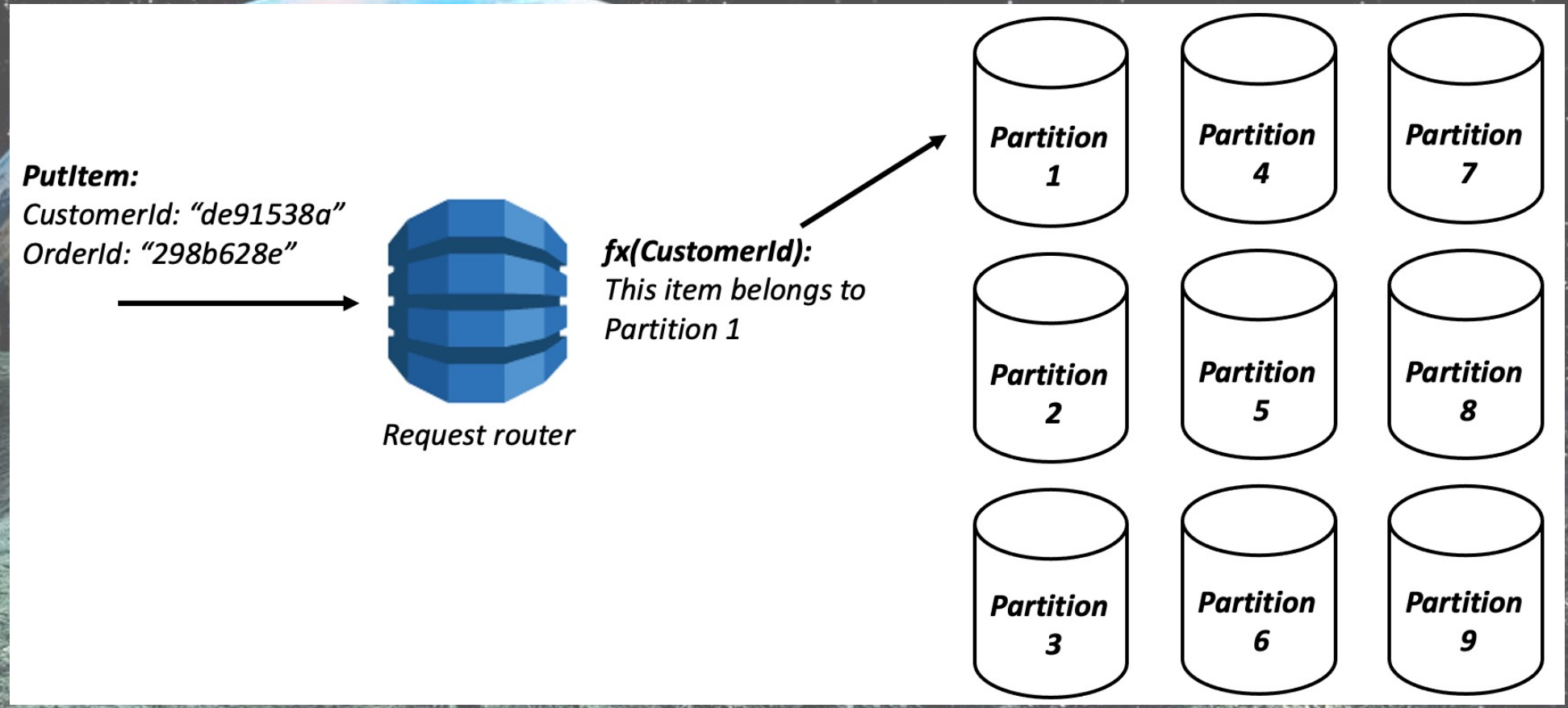



PutItem:
CustomerId: “de91538a”
OrderId: “298b628e”
Request router
fx(CustomerId):
This item belongs to
Partition 1
Partition
1
Partition
2


GetItem:
CustomerId: “de91538a”
OrderId: “298b628e”
Request router
fx(CustomerId):
This item is located on
Partition 1
Partition
1
Partition
2

EACH PARTITION IS ROUGHLY **10GB** IN SIZE

LARGE TABLE COULD HAVE THOUSANDS OF PARTITIONS



2.
ITEMS RETRIEVING

Get

```
{
  TableName : 'Orders',
  Key: {
    id: '12345'
  }
}
```

Query

```
{
  TableName: 'Orders',
  IndexName: 'some-index',
  KeyConditionExpression: '#name = :value',
  ExpressionAttributeValues: { ':value': 'shoes' },
  ExpressionAttributeNames: { '#name': 'name' }
}
```

Scan

```
{
  TableName: "Movies",
  FilterExpression: "isAdult = :isAdult",
  ExpressionAttributeValues: {
    ":isAdult": { BOOL: true },
  },
}
```

- **GetItem** retrieve via hash and range key is a 1:1 fit, the time it takes (hence performance) to retrieve it is limited by the hash and sharding internally
- **Query** results in a search on "all" range keys. It adds computational work, thus considered slower
- **Scan** operations are less efficient than other operations

GOOD PRACTICE to issue a GET instead of QUERY when you have enough informations to do so

CONDITIONS

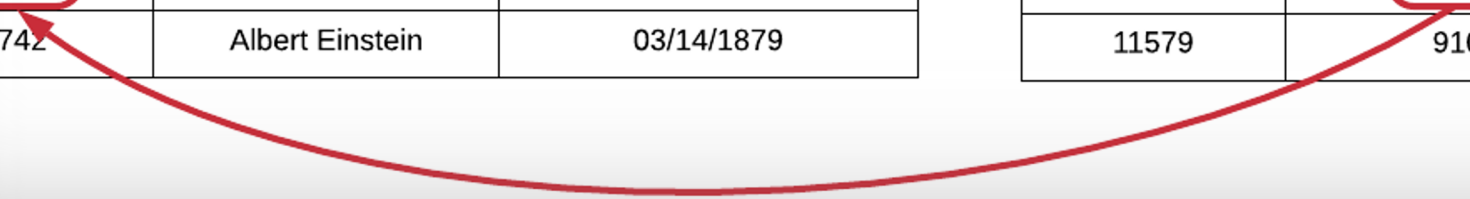
- EQ : Equal
- NE : Not equal
- LE : Less than or equal
- LT : Less than
- GE : Greater than or equal
- GT : Greater than
- NOT_NULL : The attribute exists
- NULL : The attribute does not exist
- CONTAINS : Checks for a subsequence, or value in a set
- NOT_CONTAINS : Checks for absence of a subsequence, or absence of a value in a set
- BEGINS_WITH : Checks for a prefix
- IN : Checks for matching elements in a list
- BETWEEN : Greater than or equal to the first value, and less than or equal to the second value

SQL MODELLING & JOINS

ABILITY TO GET MULTIPLE, HETEROGENOUS ITEMS FROM YOUR DATABASE IN A SINGLE REQUEST

Customers		
CustomerId	**CustomerName**	**CustomerBirthdate**
741	Alex DeBrie	05/26/1988
742	Albert Einstein	03/14/1879

Orders		
OrderId	**CustomerId**	**OrderDate**
11578	741	12/20/2019
11579	910	12/21/2019

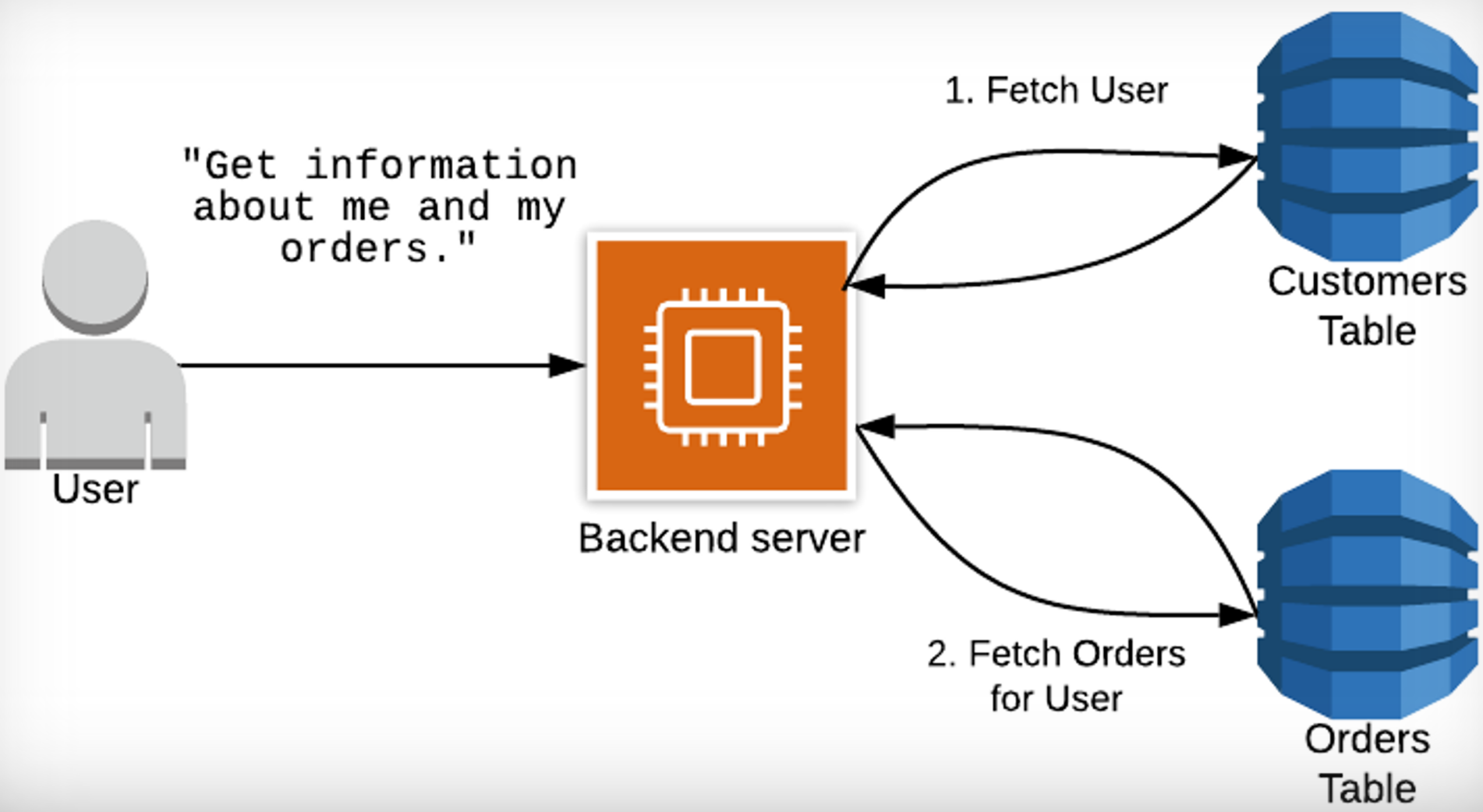


SELECT * FROM customers INNER JOIN orders

NORMALIZATION IS THE PROCESS OF **ORGANIZING THE DATA** IN THE DATABASE.

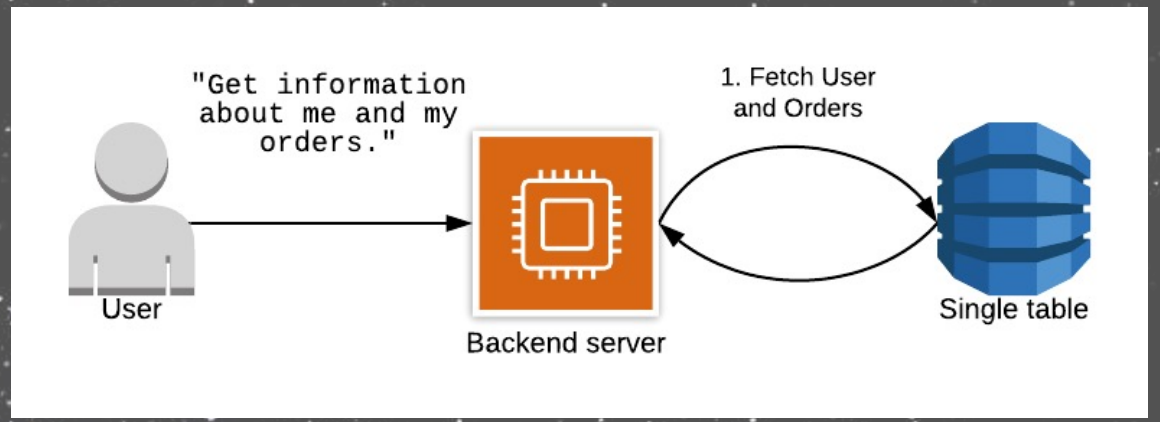
NORMALIZATION IS USED TO MINIMIZE THE REDUNDANCY FROM A RELATION OR SET OF RELATIONS.

NORMALIZATION DIVIDES THE LARGER TABLE INTO SMALLER AND LINKS THEM USING RELATIONSHIPS.


"Get information about me and my orders."
User
Backend server
1. Fetch User
Customers Table
2. Fetch Orders for User
Orders Table

SINGLE TABLE DESIGN

PRE-JOIN YOUR DATA INTO ITEM COLLECTIONS



Primary Key		Attributes				
PK	**SK**					
USER#alexdebrie	#PROFILE#alexdebrie	**Username**	**FullName**	**Email**	**CreatedAt**	**Addresses**
		alexdebrie	Alex DeBrie	alexdebrie1@gmail.com	03/23/2018	{"Home":{"StreetAddress":"1111 1st St","State":"Nebr
	ORDER#5e7272b7	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	5e7272b7	PLACED	04/21/2019	{"StreetAddress":"1111 1st St","State":"Nebraska","C
	ORDER#42ef295e	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	42ef295e	PLACED	04/25/2019	{"StreetAddress":"1111 1st St","State":"Nebraska","C
	ORDER#2e7abecc	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	2e7abecc	SHIPPED	12/25/2018	{"StreetAddress":"1111 1st St","State":"Nebraska","C
USER#nedstark	#PROFILE#nedstark	**Username**	**FullName**	**Email**	**CreatedAt**	**Addresses**
		nedstark	Eddard Stark	lord@winterfell.com	02/27/2016	{"Home":{"StreetAddress":"1234 2nd Ave","City":"Wir
	ORDER#2eae1dee	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		nedstark	2eae1dee	SHIPPED	01/15/2019	{"StreetAddress":"Suite 200, Red Keep","City":"King's L
	ORDER#f4f80a91	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		nedstark	f4f80a91	PLACED	05/12/2019	{"StreetAddress":"Suite 200, Red Keep","City":"King's L

PARTITION KEY

CustomerId (Partition Key)	**DateTime**	**OrderId**
CID-123	1653665965826	1
CID-123	1653665990056	2
CID-123	1653666002601	3
CID-456	1653666014190	2

SORT KEY

CustomerId (Partition Key)	**OrderDate** (Sort key)	**OrderId**
CID-123	1653665965826	1
CID-123	1653665990056	2
CID-123	1653666002601	3
CID-456	1653666014190	2

SYNTAX FOR FILTER AND CONDITION EXPRESSIONS

ATTRIBUTE_EXISTS / ATTRIBUTE_NOT_EXISTS / ATTRIBUTE_TYPE / BEGINS_WITH / CONTAINS/ SIZE

```
{
  TableName: "Users",
  KeyConditionExpression: "pk = :pkey and begins_with(sk, :skey)",
  ExpressionAttributeValues: {
    ":pkey": "USER#alexderbie",
    ":skey": "ORDER#",
}
```

3. ADD DIRECTORS

BENEFITS AND DOWNSIDES OF SINGLE-TABLE DESIGN

"WELL-OPTIMIZED SINGLE-TABLE DYNAMODB LAYOUT LOOKS MORE LIKE MACHINE CODE THAN A SIMPLE SPREADSHEET" (C)

BENEFITS:

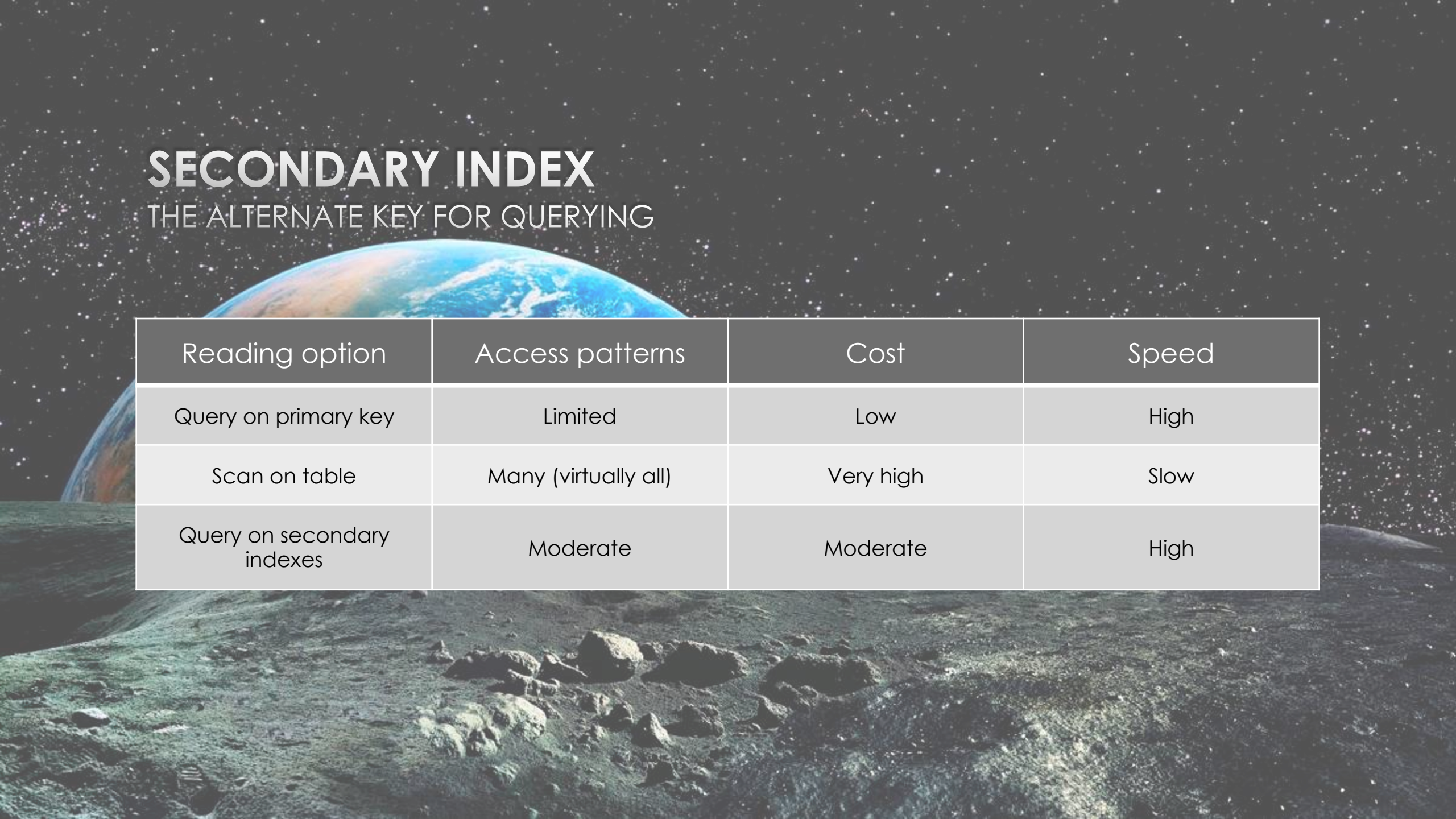
- RETRIEVING MULTIPLE, HETEROGENOUS ITEM TYPES USING A SINGLE REQUEST
- REDUCING THE NUMBER OF ALARMS AND METRICS TO WATCH
- PREDICTABLE PROVISIONING OF READ AND WRITE CAPACITY UNITS (NOT FOR ON-DEMAND PRICING)

DOWNSIDES:

- THE LEARNING CURVE TO UNDERSTAND SINGLE-TABLE DESIGN
- THE INFLEXIBILITY OF ADDING NEW ACCESS PATTERNS
- THE DIFFICULTY OF EXPORTING YOUR TABLES FOR ANALYTICS

WHEN NOT TO USE:

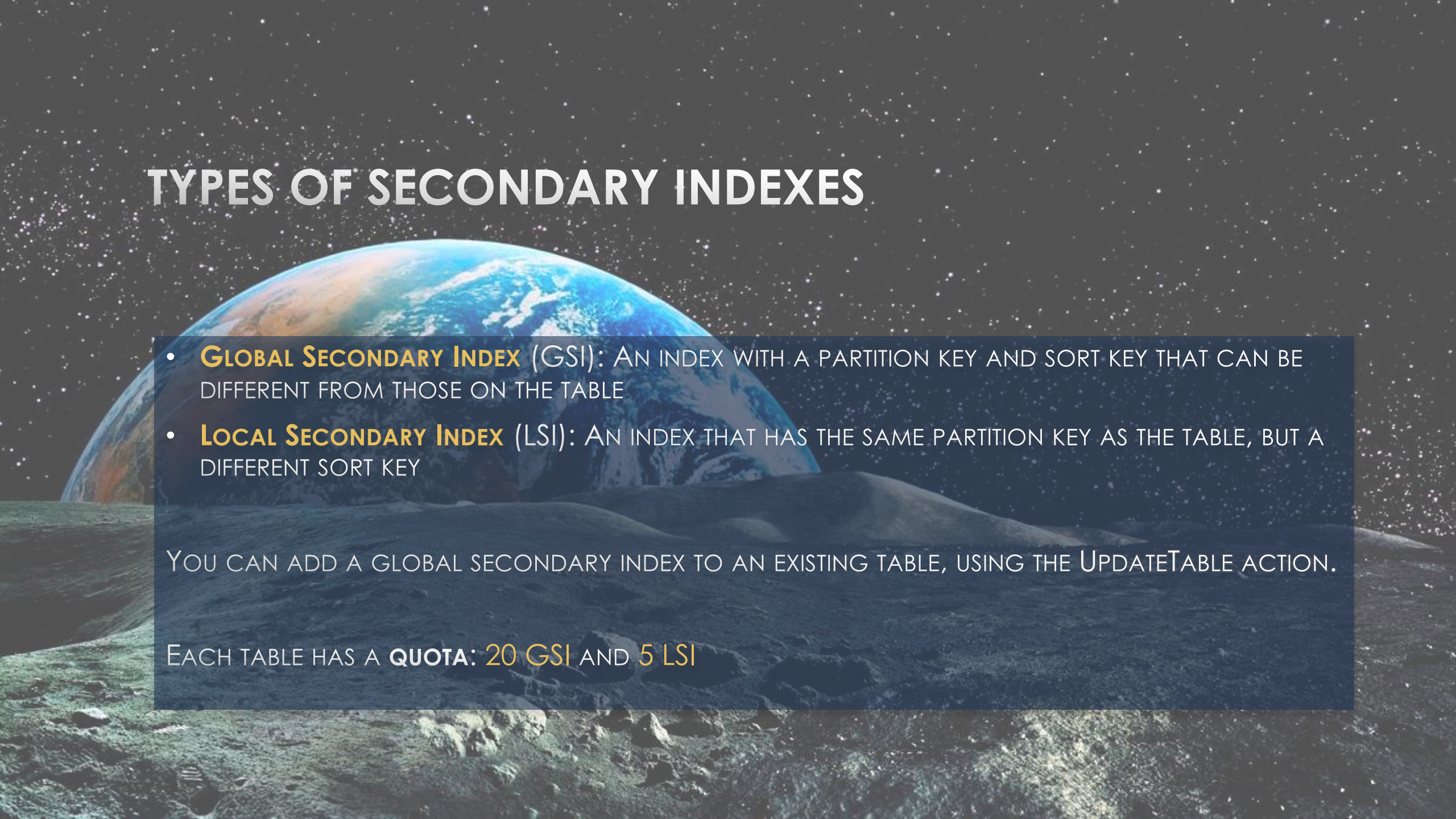
- WHEN DEVELOPER AGILITY IS MORE IMPORTANT THAN APPLICATION PERFORMANCE
- IN APPLICATIONS USING GRAPHQL



SECONDARY INDEX

THE ALTERNATE KEY FOR QUERYING

Reading option	Access patterns	Cost	Speed
Query on primary key	Limited	Low	High
Scan on table	Many (virtually all)	Very high	Slow
Query on secondary indexes	Moderate	Moderate	High

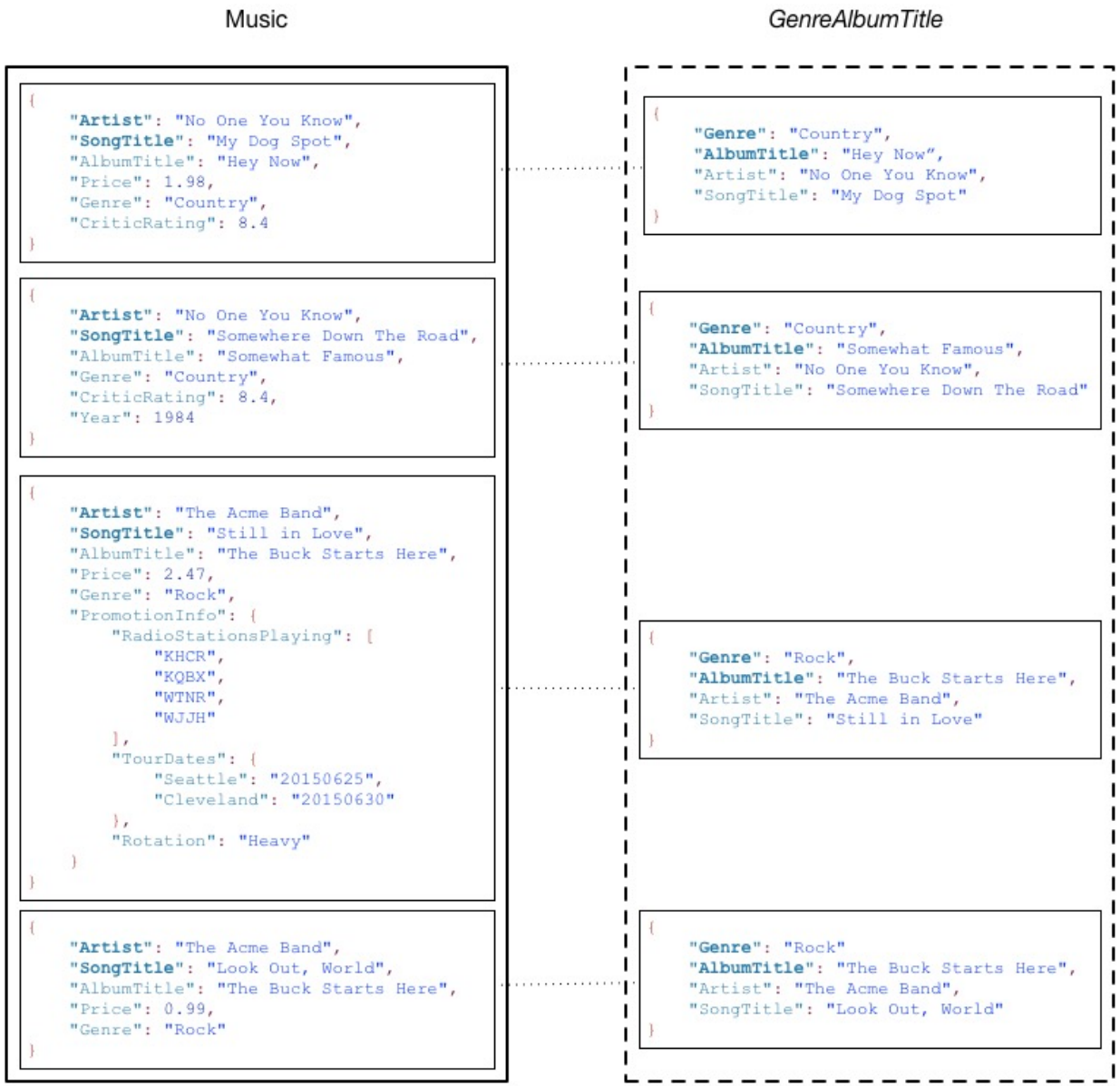


TYPES OF SECONDARY INDEXES

- **Global Secondary Index** (GSI): An index with a partition key and sort key that can be different from those on the table
- **Local Secondary Index** (LSI): An index that has the same partition key as the table, but a different sort key

You can add a global secondary index to an existing table, using the UpdateTable action.

Each table has a **quota**: 20 GSI and 5 LSI



MUSIC TABLE:

PARTITION KEY – ARTIST

SORT KEY – SONGTITLE

WHAT IF YOU ALSO WANTED TO QUERY THE DATA BY *GENRE* AND *ALBUMTITLE*?

GENREALBUMTITLE INDEX:

PARTITION KEY – GENRE

SORT KEY - ALBUMTITLE

4.
SORT
BY RATING

5.AKAS

RESERVED WORDS

ABORT ABSOLUTE ACTION ADD AFTER AGENT AGGREGATE ALL ALLOCATE ALTER ANALYZE AND ANY ARCHIVE ARE ARRAY AS ASC ASCII ASENSITIVE ASSERTION ASYMMETRIC AT ATOMIC ATTACH ATTRIBUTE AUTH AUTHORIZATION AUTHORIZE AUTO AVG BACK BACKUP BASE BATCH BEFORE BEGIN BETWEEN BIGINT BINARY BIT BLOB BLOCK BOOLEAN BOTH BREADTH BUCKET BULK BY BYTE CALL CALLED CALLING CAPACITY CASCADE CASCADED CASE CAST CATALOG CHAR CHARACTER CHECK CLASS CLOB CLOSE CLUSTER CLUSTERED CLUSTERING CLUSTERS COALESCE COLLATE COLLATION COLLECTION COLUMN COLUMNS COMBINE COMMENT COMMIT COMPACT COMPILE COMPRESS CONDITION CONFLICT CONNECT CONNECTION CONSISTENCY CONSISTENT CONSTRAINT CONSTRAINTS CONSTRUCTOR CONSUMED CONTINUE CONVERT COPY CORRESPONDING COUNT COUNTER CREATE CROSS CUBE CURRENT CURSOR CYCLE DATA DATABASE DATE DATETIME DAY DEALLOCATE DEC DECIMAL DECLARE DEFAULT DEFERRABLE DEFERRED DEFINE DEFINED DEFINITION DELETE DELIMITED DEPTH DEREF DESC DESCRIBE DESCRIPTOR DETACH DETERMINISTIC DIAGNOSTICS DIRECTORIES DISABLE DISCONNECT DISTINCT DISTRIBUTE DO DOMAIN DOUBLE DROP DUMP DURATION DYNAMIC EACH ELEMENT ELSE ELSEIF EMPTY ENABLE END EQUAL EQUALS ERROR ESCAPE ESCAPED EVAL EVALUATE EXCEEDED EXCEPT EXCEPTION EXCEPTIONS EXCLUSIVE EXEC EXECUTE EXISTS EXIT EXPLAIN EXPLODE EXPORT EXPRESSION EXTENDED EXTERNAL EXTRACT FAIL FALSE FAMILY FETCH FIELDS FILE FILTER FILTERING FINAL FINISH FIRST FIXED FLATTERN FLOAT FOR FORCE FOREIGN FORMAT FORWARD FOUND FREE FROM FULL FUNCTION FUNCTIONS GENERAL GENERATE GET GLOB GLOBAL GO GOTO GRANT GREATER GROUP GROUPING HANDLER HASH HAVE HAVING HEAP HIDDEN HOLD HOUR IDENTIFIED IDENTITY IF IGNORE IMMEDIATE IMPORT IN INCLUDING INCLUSIVE INCREMENT INCREMENTAL INDEX INDEXED INDEXES INDICATOR INFINITE INITIALLY INLINE INNER INNTER INOUT INPUT INSENSITIVE INSERT INSTEAD INT INTEGER INTERSECT INTERVAL INTO INVALIDATE IS ISOLATION ITEM ITEMS ITERATE JOIN KEY KEYS LAG LANGUAGE LARGE LAST LATERAL LEAD LEADING LEAVE LEFT LENGTH LESS LEVEL LIKE LIMIT LIMITED LINES LIST LOAD LOCAL LOCALTIME LOCALTIMESTAMP LOCATION LOCATOR LOCK LOCKS LOG LOGED LONG LOOP LOWER MAP MATCH MATERIALIZED MAX MAXLEN MEMBER MERGE METHOD METRICS MIN MINUS MINUTE MISSING MOD MODE MODIFIES MODIFY MODULE MONTH MULTI MULTISET NAME NAMES NATIONAL NATURAL NCHAR NCLOB NEW NEXT NO NONE NOT NULL NULLIF NUMBER NUMERIC OBJECT OF OFFLINE OFFSET OLD ON ONLINE ONLY OPAQUE OPEN OPERATOR OPTION OR ORDER ORDINALITY OTHER OTHERS OUT OUTER OUTPUT OVER OVERLAPS OVERRIDE OWNER PAD PARALLEL PARAMETER PARAMETERS PARTIAL PARTITION PARTITIONED PARTITIONS PATH PERCENT PERCENTILE PERMISSION PERMISSIONS PIPE PIPELINED PLAN POOL POSITION PRECISION PREPARE PRESERVE PRIMARY PRIOR PRIVATE PRIVILEGES PROCEDURE PROCESSED PROJECT PROJECTION PROPERTY PROVISIONING PUBLIC PUT QUERY QUIT QUORUM RAISE RANDOM RANGE RANK RAW READ READS REAL REBUILD RECORD RECURSIVE REDUCE REF REFERENCE REFERENCES REFERENCING REGEXP REGION REINDEX RELATIVE RELEASE REMAINDER RENAME REPEAT REPLACE REQUEST RESET RESIGNAL RESOURCE RESPONSE RESTORE RESTRICT RESULT RETURN RETURNING RETURNS REVERSE REVOKE RIGHT ROLE ROLES ROLLBACK ROLLUP ROUTINE ROW ROWS RULE RULES SAMPLE SATISFIES SAVE SAVEPOINT SCAN SCHEMA SCOPE SCROLL SEARCH SECOND SECTION SEGMENT SEGMENTS SELECT SELF SEMI SENSITIVE SEPARATE SEQUENCE SERIALIZABLE SESSION SET SETS SHARD SHARE SHARED SHORT SHOW SIGNAL SIMILAR SIZE SKEWED SMALLINT SNAPSHOT SOME SOURCE SPACE SPACES SPARSE SPECIFIC SPECIFICTYPE SPLIT SQL SQLCODE SQLERROR SQLEXCEPTION SQLSTATE SQLWARNING START STATE STATIC STATUS STORAGE STORE STORED STREAM STRING STRUCT STYLE SUB SUBMULTISET SUBPARTITION SUBSTRING SUBTYPE SUM SUPER SYMMETRIC SYNONYM SYSTEM TABLE TABLESAMPLE TEMP TEMPORARY TERMINATED TEXT THAN THEN THROUGHPUT TIME TIMESTAMP TIMEZONE TINYINT TO TOKEN TOTAL TOUCH TRAILING TRANSACTION TRANSFORM TRANSLATE TRANSLATION TREAT TRIGGER TRIM TRUE TRUNCATE TTL TUPLE TYPE UNDER UNDO UNION UNIQUE UNIT UNKNOWN UNLOGGED UNNEST UNPROCESSED UNSIGNED UNTIL UPDATE UPPER URL USAGE USE USER USERS USING UUID VACUUM VALUE VALUED VALUES VARCHAR VARIABLE VARIANCE VARINT VARYING VIEW VIEWS VIRTUAL VOID WAIT WHEN WHENEVER WHERE WHILE WINDOW WITH WITHIN WITHOUT WORK WRAPPED WRITE YEAR ZONE

RESERVED WORDS

```
{
  Key={
    'id': item_id
  },
  UpdateExpression='SET #timestamp = :val1',
  ExpressionAttributeValues={
    ":val": new_timestamp
  }
}
```

ABORT ABSOLUTE ACTION ADD AFTER
ASYMMETRIC AT ATOMIC ATTACH AT
BLOB BLOCK BOOLEAN BOTH BREADT
CHECK CLASS CLOB CLOSE CLUSTER
COMMIT COMPACT COMPILE COMP
CONSUMED CONTINUE CONVERT CO
DEALLOCATE DEC DECIMAL DECLAR
DETACH DETERMINISTIC DIAGNOSTIC
ELEMENT ELSE ELSEIF EMPTY ENABLE E
EXECUTE EXISTS EXIT EXPLAIN EXPLOD
FIXED FLATTERN FLOAT FOR FORCE FO
GOTO GRANT GREATER GROUP GRO
INCLUDING INCLUSIVE INCREMENT IN
INSTEAD INT INTEGER INTERSECT INTER
LEAVE LEFT LENGTH LESS LEVEL LIKE LI
LOOP LOWER MAP MATCH MATERIAL
MONTH MULTI MULTISET NAME NAMES
OLD ON ONLINE ONLY OPAQUE OPE
PARALLEL PARAMETER PARAMETERS F
POSITION PRECISION PREPARE PRESER

Y AS ASC ASCII ASENSITIVE ASSERTION
FORE BEGIN BETWEEN BIGINT BINARY BIT
ASE CAST CATALOG CHAR CHARACTER
OLUMN COLUMNS COMBINE COMMENT
ONSTRAINT CONSTRAINTS CONSTRUCTOR
YCLE DATA DATABASE DATE DATETIME DAY
TH DEREF DESC DESCRIBE DESCRIPTOR
P DUMP DURATION DYNAMIC EACH
CEPTION EXCEPTIONS EXCLUSIVE EXEC
FILE FILTER FILTERING FINAL FINISH FIRST
L GENERATE GET GLOB GLOBAL GO
Y IF IGNORE IMMEDIATE IMPORT IN
ER INOUT INPUT INSENSITIVE INSERT
GE LARGE LAST LATERAL LEAD LEADING
OR LOCK LOCKS LOG LOGED LONG
MOD MODE MODIFIES MODIFY MODULE
R NUMERIC OBJECT OF OFFLINE OFFSET
ER OVERLAPS OVERRIDE OWNER PAD
PERMISSIONS PIPE PIPELINED PLAN POOL
TION PROPERTY PROVISIONING PUBLIC PUT

QUERY QUIT QUORUM RAISE RANDOM RANGE RANK RAW READ READS REAL REBUILD RECORD RECURSIVE REDUCE REF REFERENCE REFERENCES REFERENCING
REGEXP REGION REINDEX RELATIVE RELEASE REMAINDER RENAME REPEAT REPLACE REQUEST RESET RESIGNAL RESOURCE RESPONSE RESTORE RESTRICT RESULT
RETURN RETURNING RETURNS REVERSE REVOKE RIGHT ROLE ROLES ROLLBACK ROLLUP ROUTINE ROW ROWS RULE RULES SAMPLE SATISFIES SAVE SAVEPOINT SCAN
SCHEMA SCOPE SCROLL SEARCH SECOND SECTION SEGMENT SEGMENTS SELECT SELF SEMI SENSITIVE SEPARATE SEQUENCE SERIALIZABLE SESSION SET SETS SHARD
SHARE SHARED SHORT SHOW SIGNAL SIMILAR SIZE SKEWED SMALLINT SNAPSHOT SOME SOURCE SPACE SPACES SPARSE SPECIFIC SPECIFICTYPE SPLIT SQL SQLCODE
SQLERROR SQLEXCEPTION SQLSTATE SQLWARNING START STATE STATIC STATUS STORAGE STORE STORED STREAM STRING STRUCT STYLE SUB SUBMULTISET
SUBPARTITION SUBSTRING SUBTYPE SUM SUPER SYMMETRIC SYNONYM SYSTEM TABLE TABLESAMPLE TEMP TEMPORARY TERMINATED TEXT THAN THEN THROUGHPUT
TIME TIMESTAMP TIMEZONE TINYINT TO TOKEN TOTAL TOUCH TRAILING TRANSACTION TRANSFORM TRANSLATE TRANSLATION TREAT TRIGGER TRIM TRUE TRUNCATE TTL
TUPLE TYPE UNDER UNDO UNION UNIQUE UNIT UNKNOWN UNLOGGED UNNEST UNPROCESSED UNSIGNED UNTIL UPDATE UPPER URL USAGE USE USER USERS USING
UUID VACUUM VALUE VALUED VALUES VARCHAR VARIABLE VARIANCE VARINT VARYING VIEW VIEWS VIRTUAL VOID WAIT WHEN WHENEVER WHERE WHILE WINDOW
WITH WITHIN WITHOUT WORK WRAPPED WRITE YEAR ZONE

RESERVED WORDS

```
{
  Key={
    'id': item_id
  },
  UpdateExpression='SET #timestamp = :val1',
  ExpressionAttributeValues={
    ":val": new_timestamp
  }
}
```

```
{
  Key={
    'id': item_id
  },
  UpdateExpression='SET #ts = :val1',
  ExpressionAttributeValues={
    ":val1": new_timestamp
  },
  ExpressionAttributeNames={
    "#ts": "timestamp"
  }
}
```

ABORT ABSOLUTE ACTION ADD AFTER AGENT AGGREGATE ALL ALLOCATE ALTER A
SUBPARTITION SUBSTRING SUBTYPE SUM SUPER SYMMETRIC SYNONYM SYSTEM TABLE
TIME TIMESTAMP TIMEZONE TINYINT TO TOKEN TOTAL TOUCH TRAILING TRANSACTIO
TUPLE TYPE UNDER UNDO UNION UNIQUE UNIT UNKNOWN UNLOGGED UNNEST UNP
UUID VACUUM VALUE VALUED VALUES VARCHAR VARIABLE VARIANCE VARINT VARYING VIEW VIEWS VIRTUAL VOID WAIT WHEN WHENEVER WHERE WHILE WINDOW
WITH WITHIN WITHOUT WORK WRAPPED WRITE YEAR ZONE

ACID TRANSACTIONS

Atomicity

All changes are performed as if they are a single operation.

All changes are performed, or none of them are.

Consistency

Data is in a consistent state when a transaction starts and when it ends

Isolation

The intermediate state of a transaction is invisible to other transactions

Durability

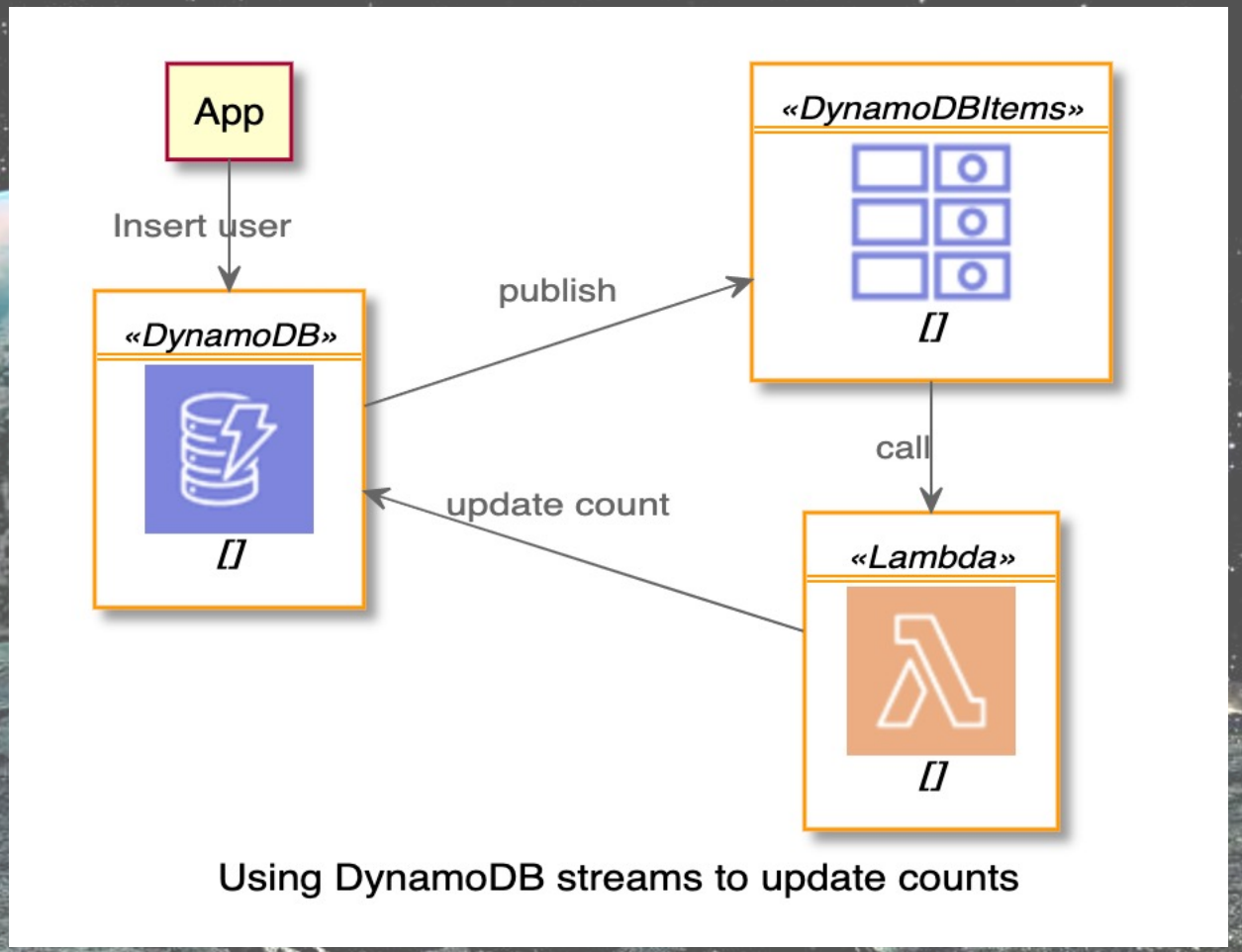
After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure

HOW TO KEEP ACCURATE COUNTS IN DYNAMODB

SELECT COUNT(*) FROM users

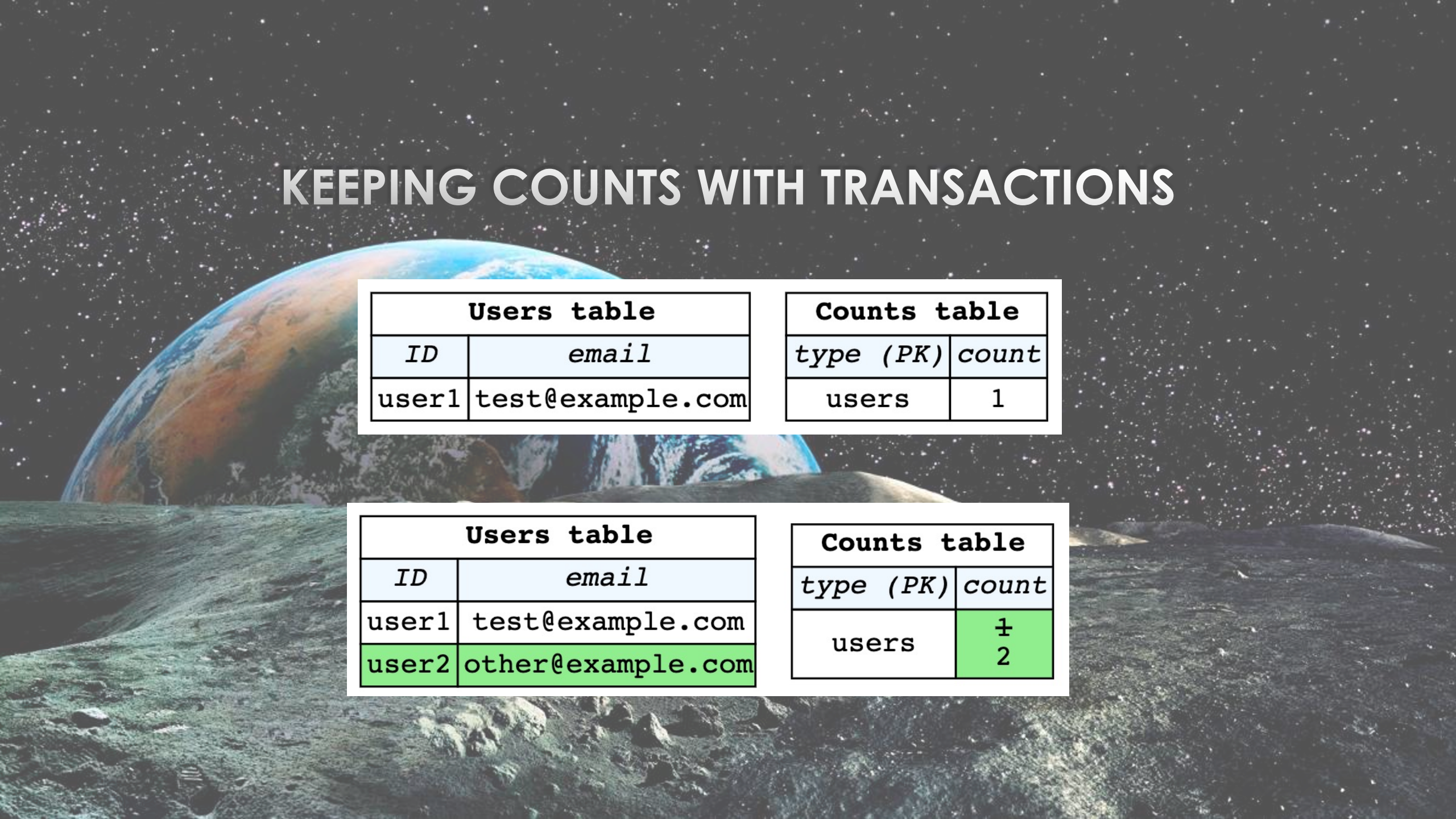
aws dynamodb scan --table-name users --select "COUNT"

```
{
    "Count": 123,
    "ScannedCount": 123,
    "ConsumedCapacity": null
}
```



Using DynamoDB streams to update counts

KEEPING COUNTS WITH TRANSACTIONS



Users table	
ID	*email*
user1	test@example.com

Counts table	
type (PK)	*count*
users	1

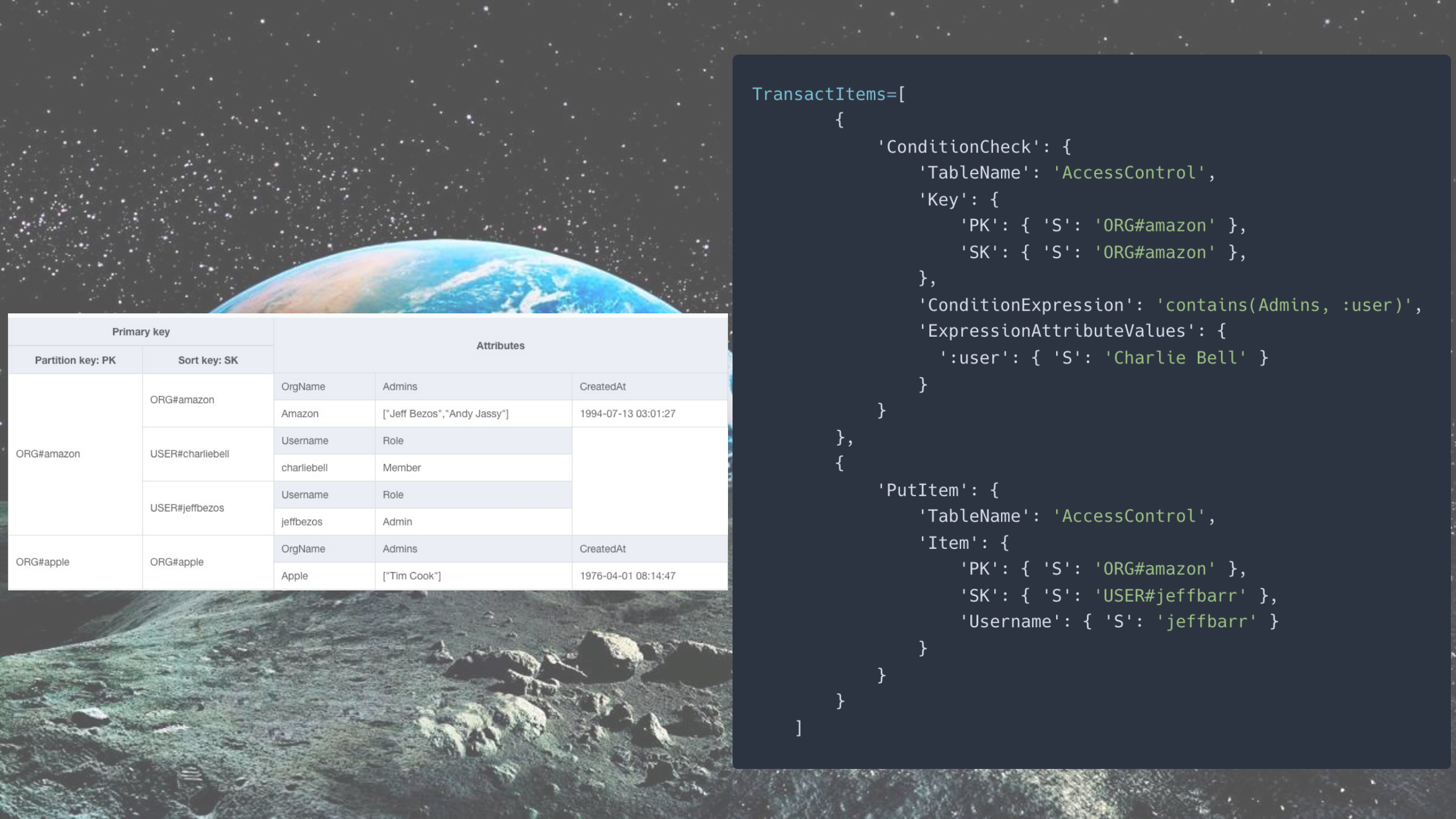
Users table	
ID	*email*
user1	test@example.com
user2	other@example.com

Counts table	
type (PK)	*count*
users	1 + 2

```
TransactItems: [
        {
            Update: {
                TableName: 'items',
                Key: { id: { S: itemId } },
                ConditionExpression: 'available = :true',
                UpdateExpression: 'set available = :false, ' +
                    'ownedBy = :player',
                ExpressionAttributeValues: {
                    ':true': { BOOL: true },
                    ':false': { BOOL: false },
                    ':player': { S: playerId }
                }
            }
        },
        {
            Update: {
                TableName: 'players',
                Key: { id: { S: playerId } },
                ConditionExpression: 'coins >= :price',
                UpdateExpression: 'set coins = coins - :price, ' +
                    'inventory = list_append(inventory, :items)',
                ExpressionAttributeValues: {
                    ':items': { L: [{ S: itemId }] },
                    ':price': { N: itemPrice.toString() }
                }
            }
        }
    ]
```

TRANSACTIONS: ACCESS CONTROL USE CASE

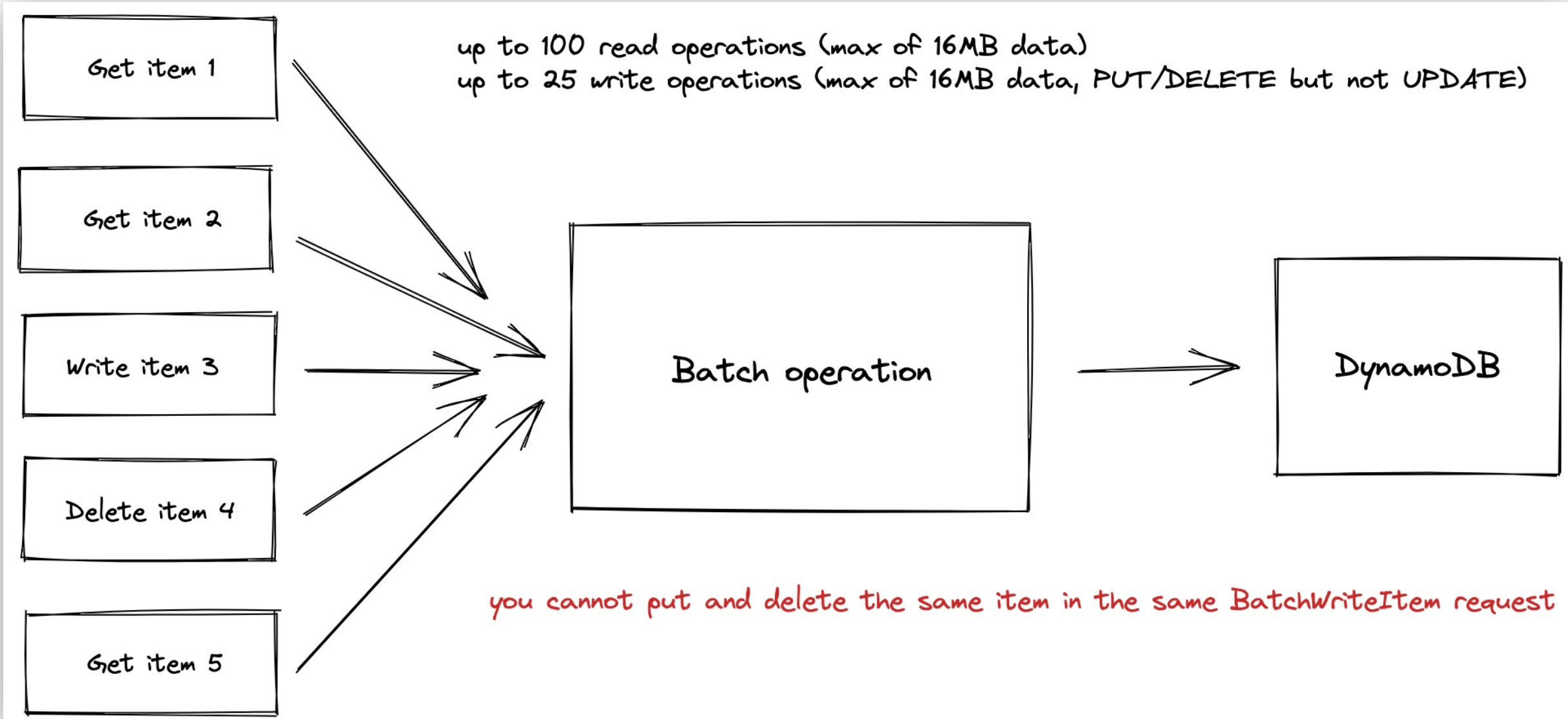
Primary key		Attributes		
Partition key: PK	**Sort key: SK**			
ORG#amazon	ORG#amazon	OrgName	Admins	CreatedAt
		Amazon	["Jeff Bezos","Andy Jassy"]	1994-07-13 03:01:27
	USER#charliebell	Username	Role	
		charliebell	Member	
	USER#jeffbezos	Username	Role	
		jeffbezos	Admin	
ORG#apple	ORG#apple	OrgName	Admins	CreatedAt
		Apple	["Tim Cook"]	1976-04-01 08:14:47

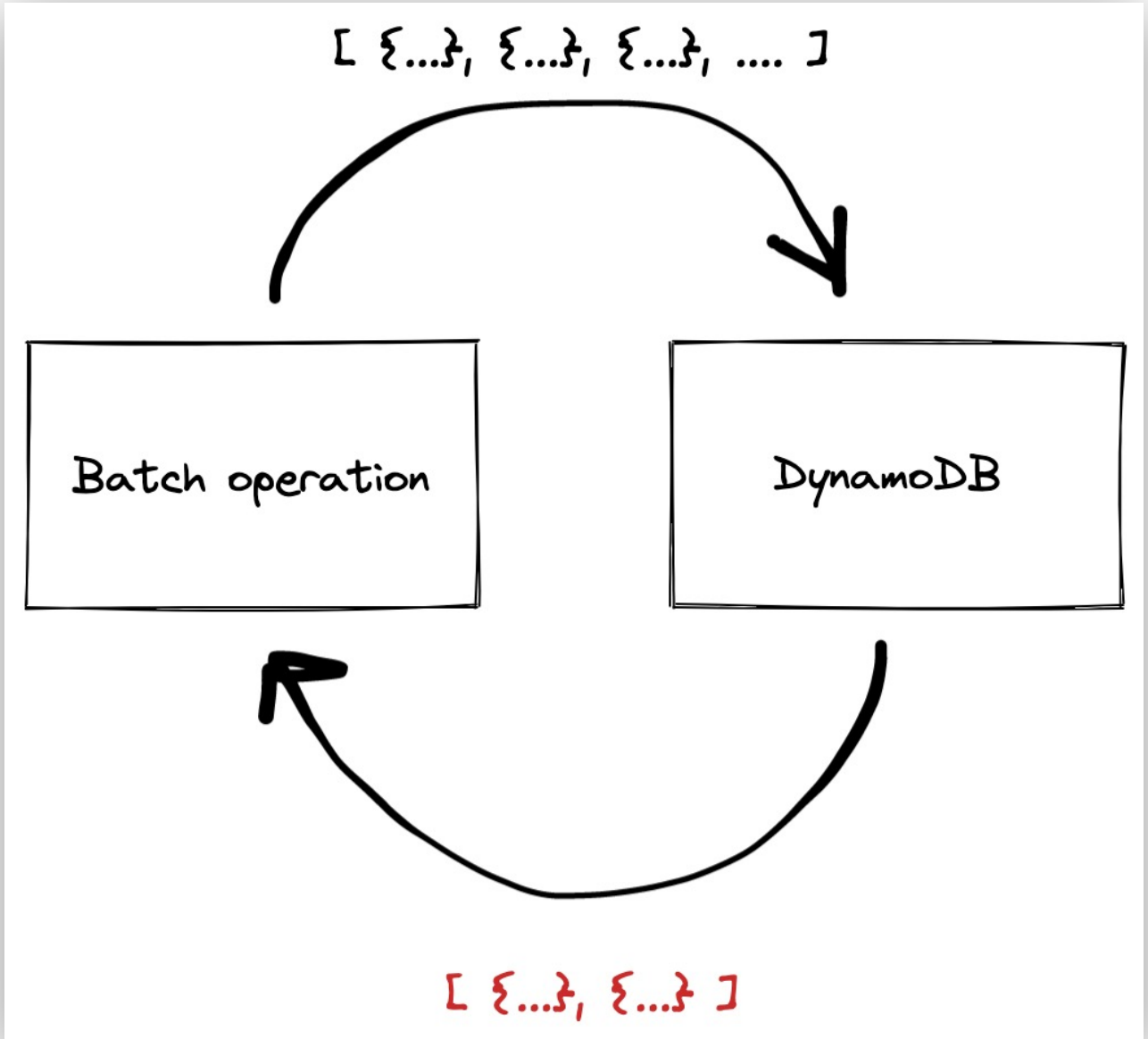


Primary key		Attributes		
Partition key: PK	Sort key: SK			
ORG#amazon	ORG#amazon	OrgName	Admins	CreatedAt
		Amazon	["Jeff Bezos","Andy Jassy"]	1994-07-13 03:01:27
	USER#charliebell	Username	Role	
		charliebell	Member	
	USER#jeffbezos	Username	Role	
		jeffbezos	Admin	
ORG#apple	ORG#apple	OrgName	Admins	CreatedAt
		Apple	["Tim Cook"]	1976-04-01 08:14:47

```
TransactItems=[
        {
            'ConditionCheck': {
                'TableName': 'AccessControl',
                'Key': {
                    'PK': { 'S': 'ORG#amazon' },
                    'SK': { 'S': 'ORG#amazon' },
                },
                'ConditionExpression': 'contains(Admins, :user)',
                'ExpressionAttributeValues': {
                  ':user': { 'S': 'Charlie Bell' }
                }
            }
        },
        {
            'PutItem': {
                'TableName': 'AccessControl',
                'Item': {
                    'PK': { 'S': 'ORG#amazon' },
                    'SK': { 'S': 'USER#jeffbarr' },
                    'Username': { 'S': 'jeffbarr' }
                }
            }
        }
    ]
```

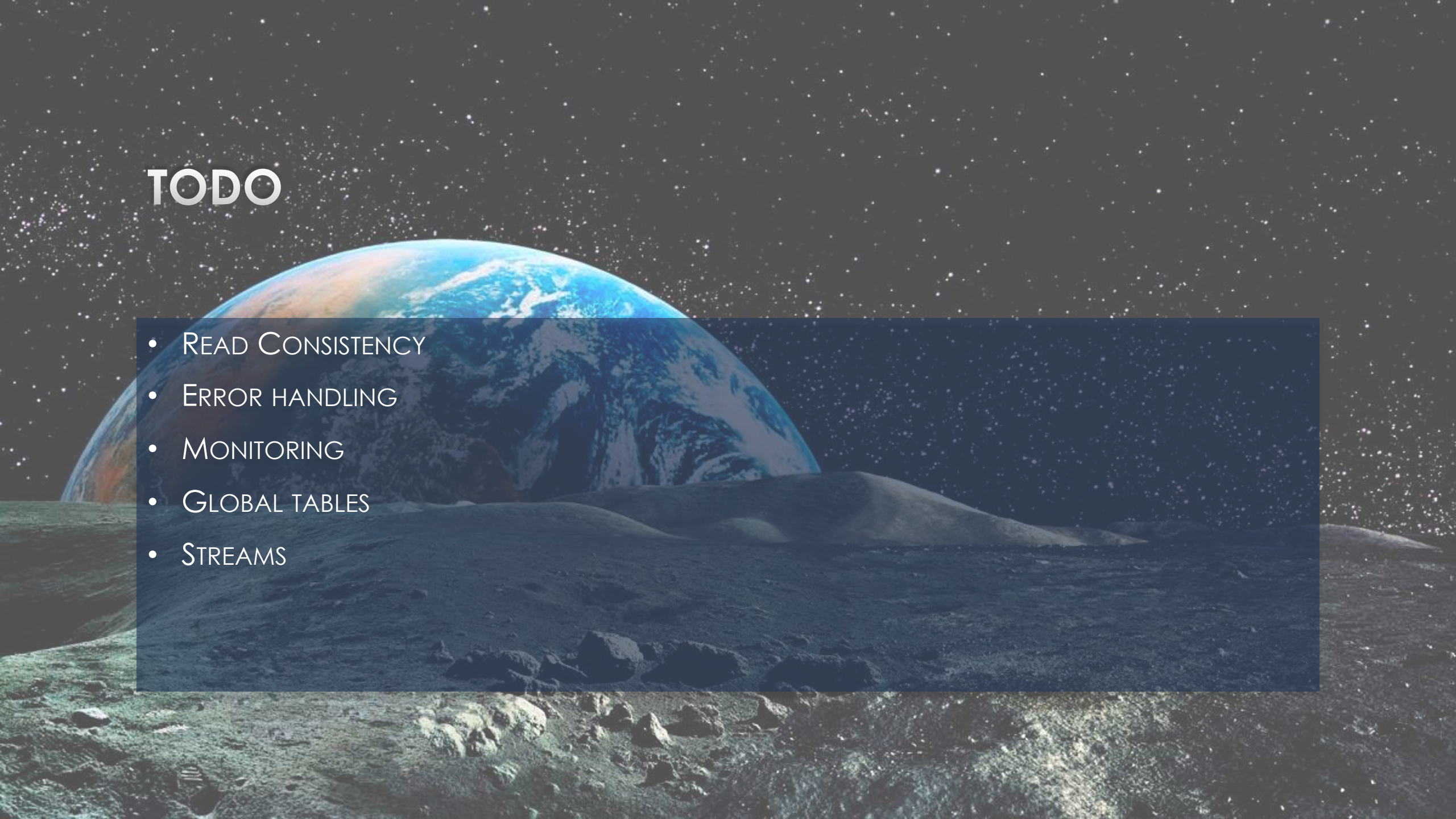
5.
BATCH OPERATIONS


Get item 1
Get item 2
Write item 3
Delete item 4
Get item 5
up to 100 read operations (max of 16MB data)
up to 25 write operations (max of 16MB data, PUT/DELETE but not UPDATE)
Batch operation
DynamoDB
you cannot put and delete the same item in the same BatchWriteItem request


[{...}, {...}, {...}, ...]
Batch operation
DynamoDB
[{...}, {...}]

USEFUL LINKS

- Official Documentation
  https://docs.aws.amazon.com/dynamodb/index.html
- Amazon Labs
  https://amazon-dynamodb-labs.com
- DynamoDB Guide
  https://www.dynamodbguide.com/
- Alex DeBrie
  https://www.alexdebrie.com/
- Dynobase
  https://dynobase.dev/dynamodb-tutorials



TODO

- Read Consistency
- Error handling
- Monitoring
- Global tables
- Streams

PRICING

On-demand capacity mode

- Create new tables with unknown workloads
- Have unpredictable application traffic
- Prefer the ease of paying for only what you use

Provisioned capacity mode

- Have predictable application traffic
- Run applications whose traffic is consistent or ramps gradually
- Can forecast capacity requirements to control costs

PRICING

Free Tier

25 Gb of Storage

25 provisioned Write Capacity Units (WCU)

25 provisioned Read Capacity Units (RCU)

Enough to handle up to 200M requests per month.

1 KB ~ 1024 B

1 RCU <= 4 KB per second

1 WCU <= 1 KB per second

Transactional write requests require 2 WCUs

5 RCU = 20 KB per second

Payload 7 KB ~ 2 requests per second

On-demand capacity mode

Read/Write Capacity Units (RCU/WCU)

RCU up to 4 KB read request per second

WCU up to 1 KB read request per second

Provisioned capacity mode

Read/Write Request Units (RRU/WRU)

RCU up to 4 KB read request **NOT** per second

WCU up to 1 KB read request **NOT** per second

DynamoDB does not consume capacity units for:

- Creating snapshot backup of the table and restoring it (billed per GB/month)
- Having Continuous backups for point in time recovery enabled (billed per GB/month)
- Streaming changes using DynamoDB Streams (billed on stream read request units, separate from table RCU/RRU)
- Transferring the data to another region (billed per GB)

TRANSACTIONS - PRICING X2

Transactional read/write requests require 2 RCUs/WCUs:

Read request of 8 KB = 2 RCU

Transactional read request of 8 KB = 4 RCU

Write request of 3 KB = 3 WCU

Transactional write request of 3 KB = 6 WCU

BEST PRACTICES

- Use queries instead of scans
- Keep item size small:
  - If possible - compress larger attribute values
  - Store objects larger than 400KB in S3 and use pointers (S3 Object ID) in DynamoDB

Amazon DynamoDB is not ideal for the following situations:

- Traditional RDS apps.
- Joins and/or complex transactions.
- BLOB data.
- Large data with low I/O rate.

IS DYNAMODB RIGHT FOR YOUR USE CASE?

- Have had scalability problems with other traditional database systems
- High-performance reads and writes are easy to manage with DynamoDB
- Are deploying a mission-critical application that must be highly available at all times without manual intervention
- Require a high level of data durability, regardless of your backup-and-restore strategy
- Have insufficient data for forecasting peaks and valleys in required database performance