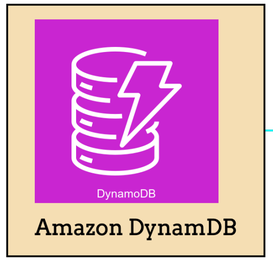
Amazon DynamoDB

Table of Contents



Amazon DynamDB

What is Amazon DynamoDB?

1. 🚀 Fast and predictable performance

⚡ Rapid, consistent performance

🗃️ Efficient data access, retrieval

2. 🌍 Seamless scalability

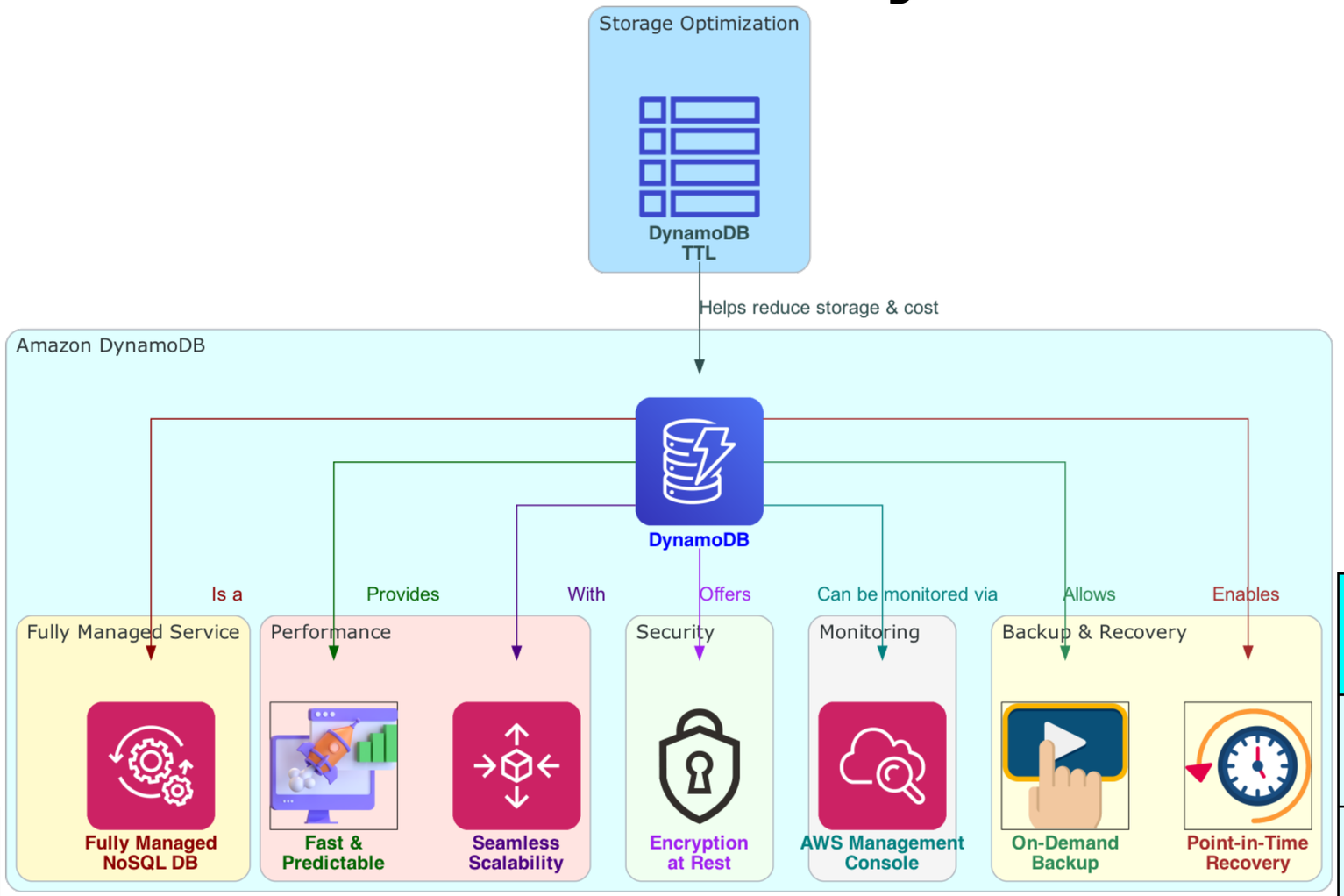
📏 Scale tables up or down easily

🚫 No downtime or performance issues

3. 🔄 Offload administrative burdens

🔧 No hardware provisioning, setup, config

🔁 No replication, patching, scaling tasks



4. 🔒 Encryption at rest

🔐 Built-in encryption for sensitive data

🔒 Reduces data protection complexity

5. 📊 Flexible throughput capacity

🎚️ Adjust capacity as needed

🌊 Handles varying traffic levels

6. 📅 On-demand backup and point-in-time recovery

💾 Create full backups of tables

⏰ Restore to any point in last 35 days

🔐 Protects against accidental changes

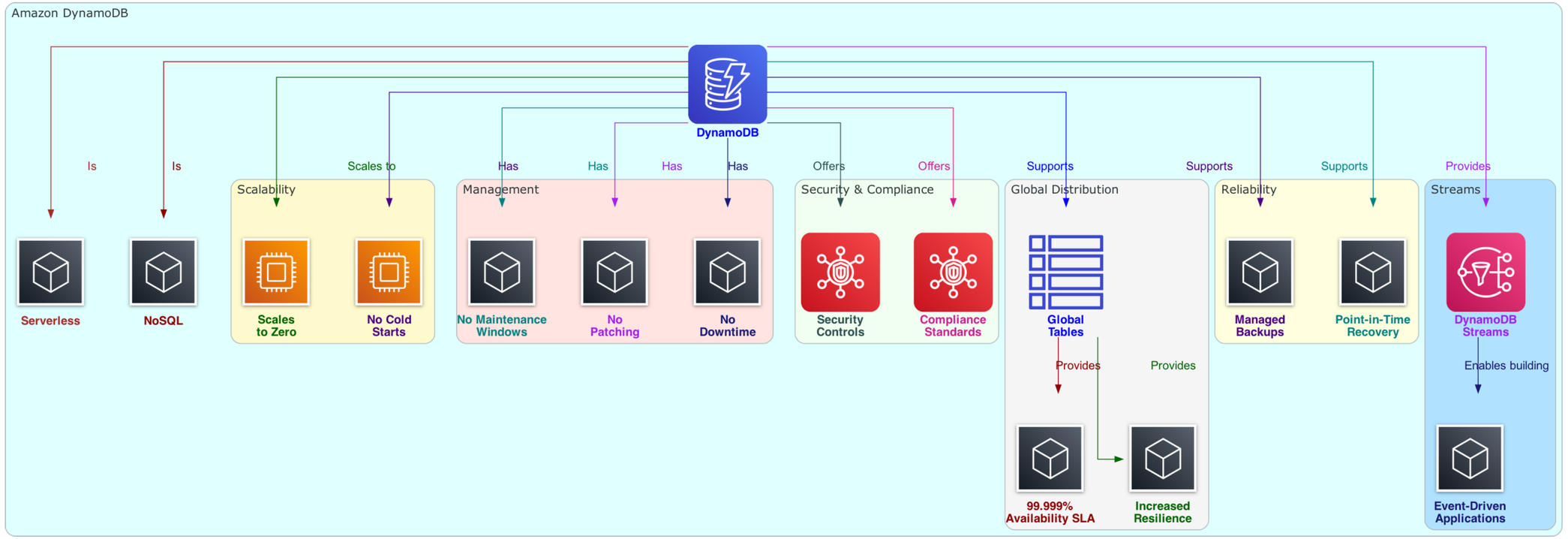
7. 🗑️ Automatic deletion of expired items

⏳ Set expiration timestamps on items

🧹 Auto-deletes irrelevant items

💸 Saves storage space and reduces costs

Why DynamoDB?



1. 🚀 Serverless and scalable

- 🗄️ Fully managed NoSQL database
- ⚖️ Automatic scaling for any traffic level
- 🆕 Ideal for modern, high-performance apps

2. 💰 Pay-per-use pricing

- 💸 Pay only for resources consumed
- 🤷‍♂️ No infrastructure provisioning or management
- 📉 Scales to zero when not in use
- $ Minimizes costs

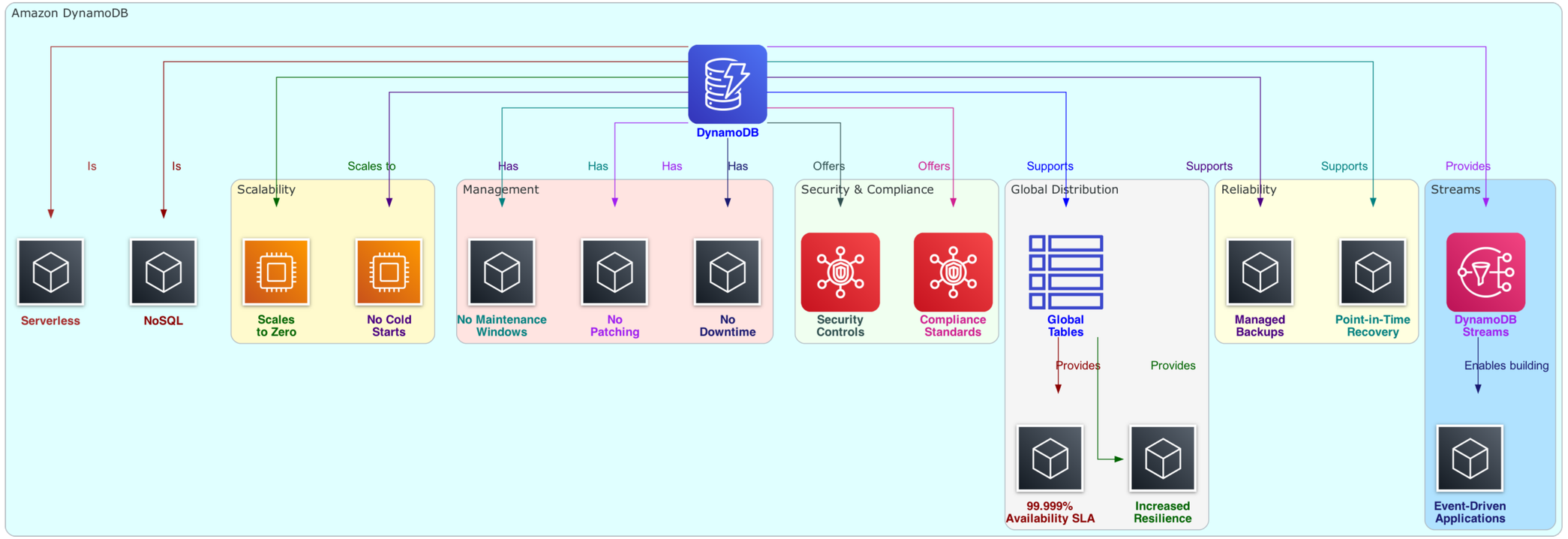
3. ❄️ No cold starts or maintenance

- 🚫 No version upgrades needed
- 🔧 No maintenance windows
- 🩹 No patching required
- ⏰ No downtime
- 🟢 App remains available and responsive

4. 🔒 Robust security and compliance

- 🛡️ Wide range of security controls
- 📜 Compliance certifications
- 🔐 Protects data
- ✅ Meets regulatory requirements

Why DynamoDB?



5. 🌐 Global tables with 99.999% availability
🗺️ For globally distributed apps
🌍 Multi-Region, multi-active database
💯 99.999% availability SLA
🦾 Increased resilience

6. 🔄 Managed backups and point-in-time recovery
🗄️ Enhanced reliability
💾 Managed backups
⏳ Point-in-time recovery
🔒 Data protection and recoverability

7. 🌊 DynamoDB streams for event-driven apps
🏗️ Build serverless, event-driven apps
⚡ React to database changes in real-time
💪 Enables powerful and efficient architectures

Core components of Amazon DynamoDB

1. 🗂️ Tables, items, attributes

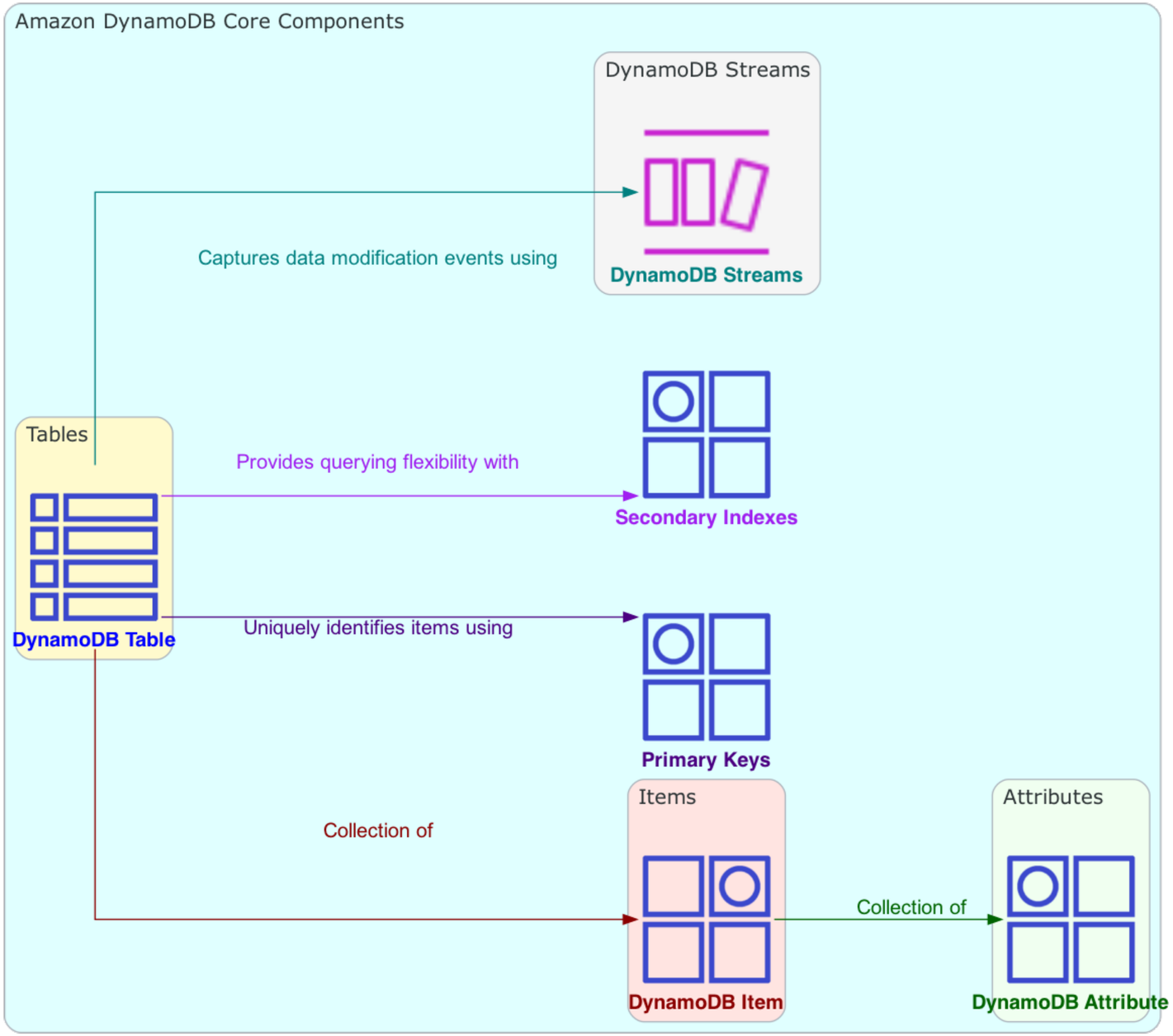
📊 Core components to work with

2. 🗂️ Tables

🗃️ Collection of items

3. 🗂️ Items

🗂️ Collection of attributes



4. 🔑 Primary keys

🆔 Uniquely identify items in table

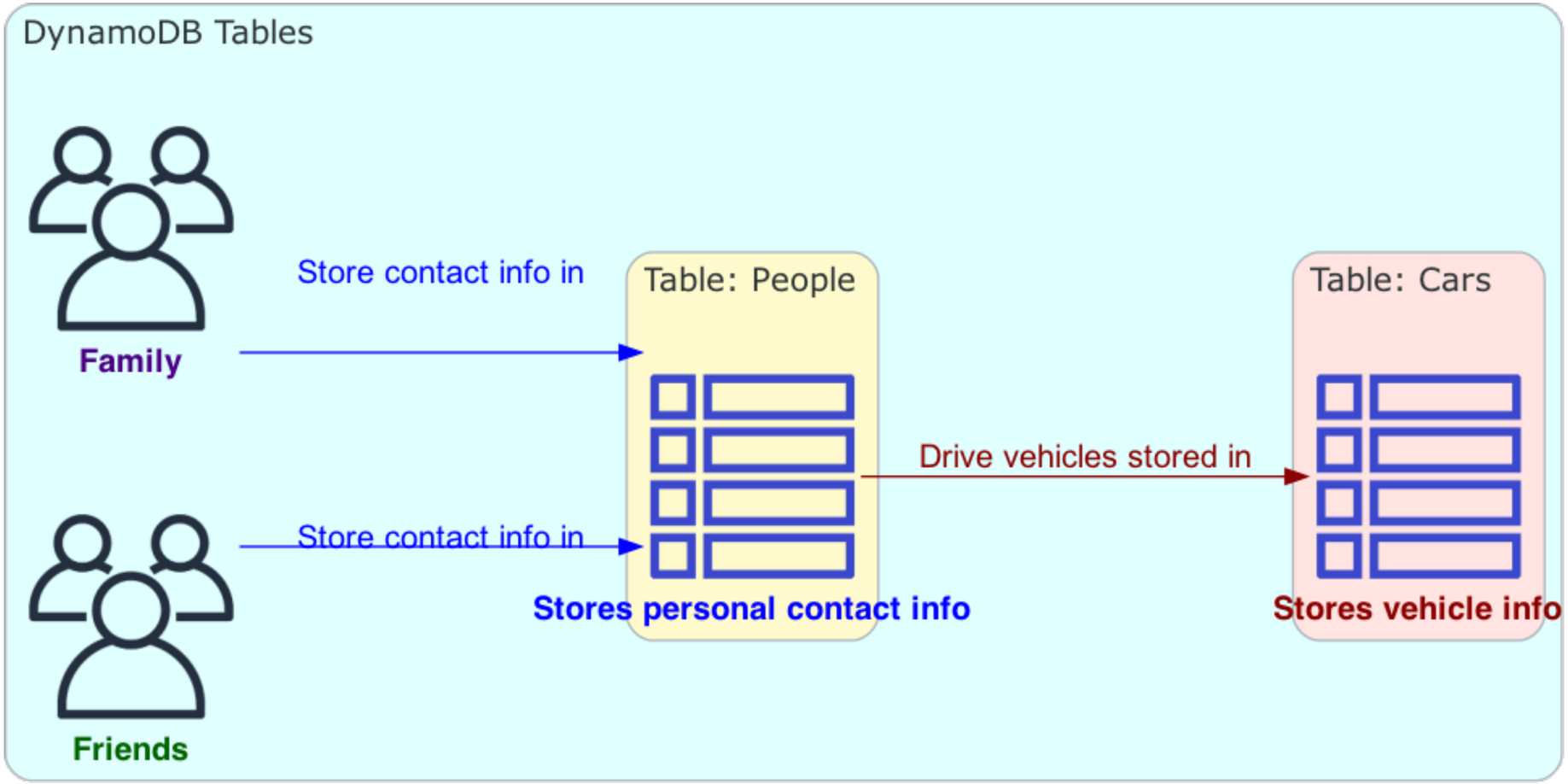
5. 🔍 Secondary indexes

🔎 Provide more querying flexibility

6. 🌊 DynamoDB Streams

🎣 Capture data modification events

Tables



1. 📊 Similar to other databases

🗄️ Stores data in tables

2. 🗂️ Table

🗃️ Collection of data

3. 👥 People table

📇 Stores personal contact info

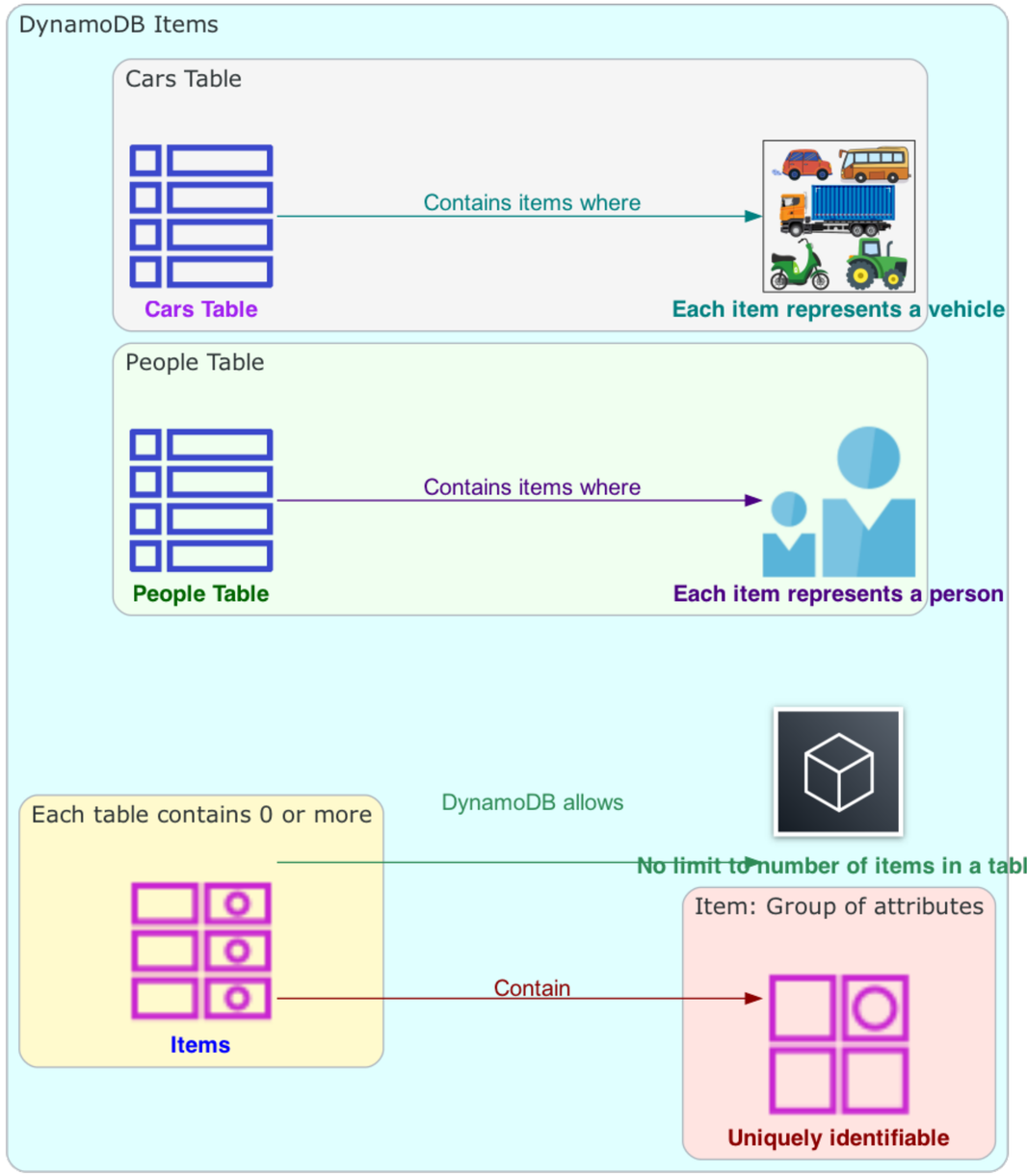
👫 Friends, family, others

4. 🚗 Cars table

🚙 Stores vehicle information

👬 Vehicles people drive

Items



1. 📄 Each table contains 0 or more

⌨️ Items

2. ⌨️ Item

🗂️ Group of attributes

🆔 Uniquely identifiable

3. 👥 People table

👴 Each item represents a person

4. 🚗 Cars table

🚙 Each item represents one vehicle

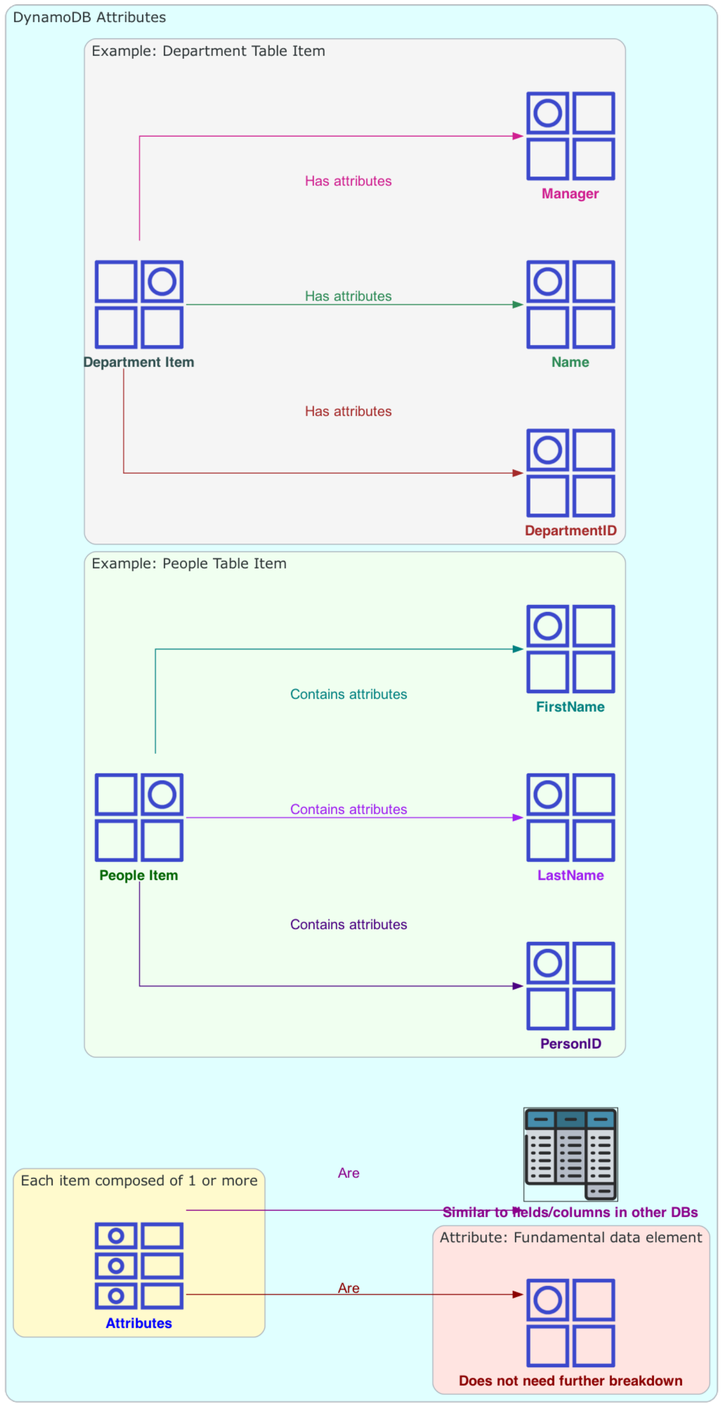
5. 📊 Similar to other databases

📈 Rows, records, tuples

6. 🗄️ No limit

🧱 To number of items in a table

Attributes



1. 📦 Each item composed of 1 or more

🗃️ Attributes

2. 🗃️ Attribute

🧩 Fundamental data element

🔍 Does not need further breakdown

3. 👥 Example: People table item

🆔 PersonID

👫 LastName, FirstName, etc.

4. 🏢 Example: Department table item

🔢 DepartmentID

📛 Name

👔 Manager, etc.

5. 📈 Similar to other databases

🗄️ Fields or columns

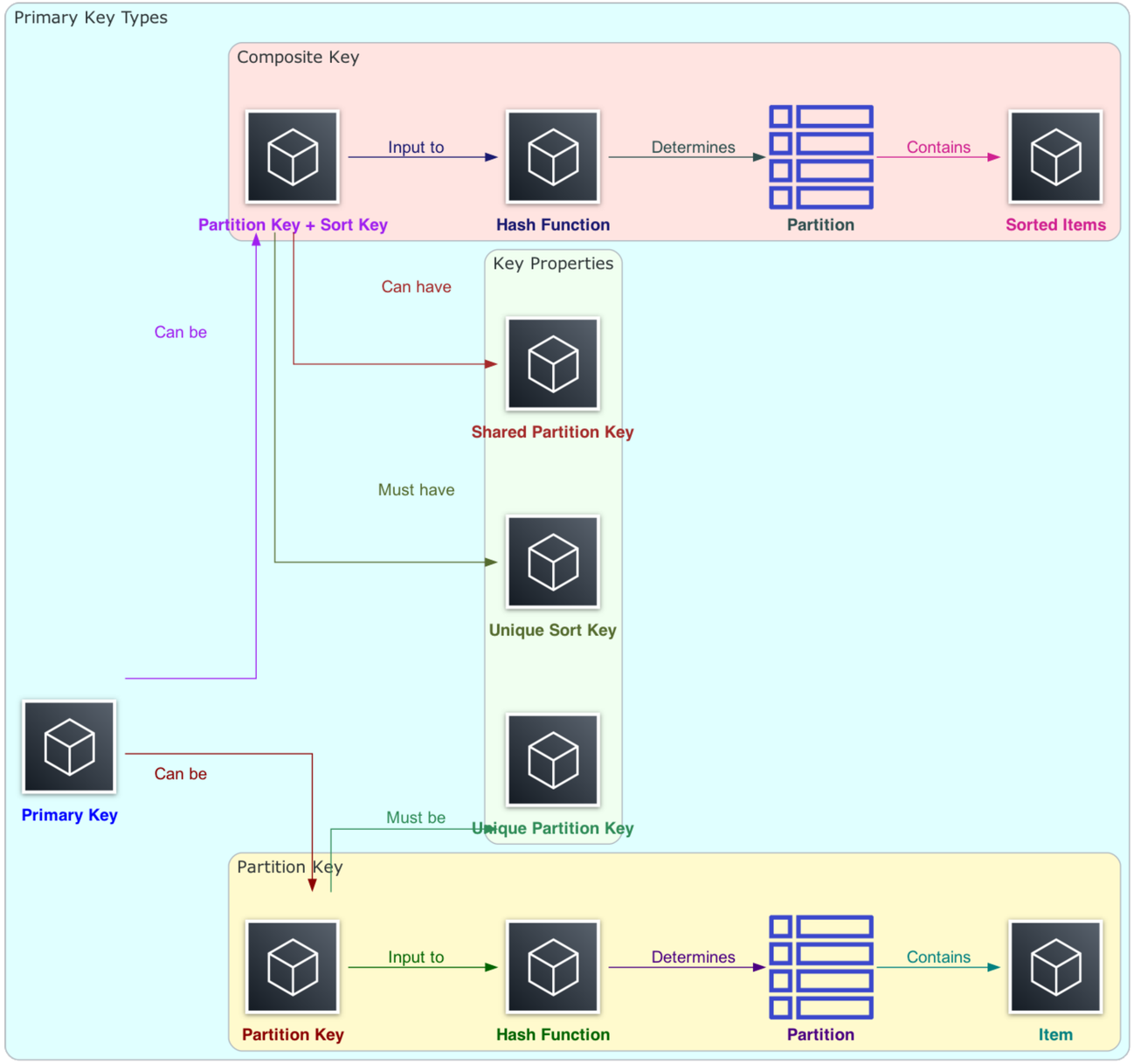
People


```
{
    "PersonID": 101,
    "LastName": "Smith",
    "FirstName": "Fred",
    "Phone": "555-4321"
}

{
    "PersonID": 102,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}

{
    "PersonID": 103,
    "LastName": "Stephens",
    "FirstName": "Howard",
    "Address": {
                "Street": "123 Main",
                "City": "London",
                "PostalCode": "ER3 5K8"
    },
    "FavoriteColor": "Blue"
}
```

Primary Key



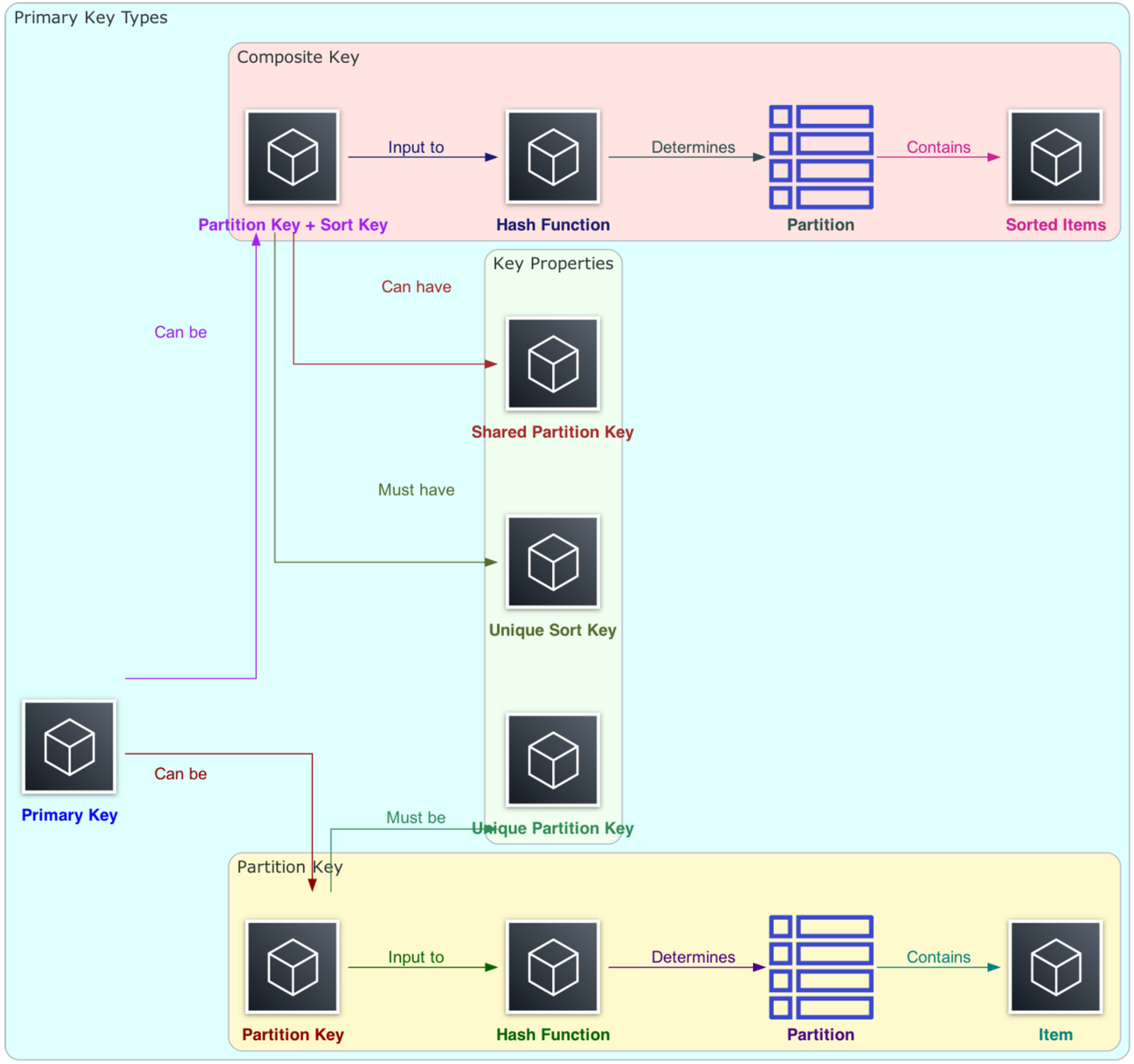
1. 📝 Table Creation

✍️ Specify table name

🗝️ Specify primary key

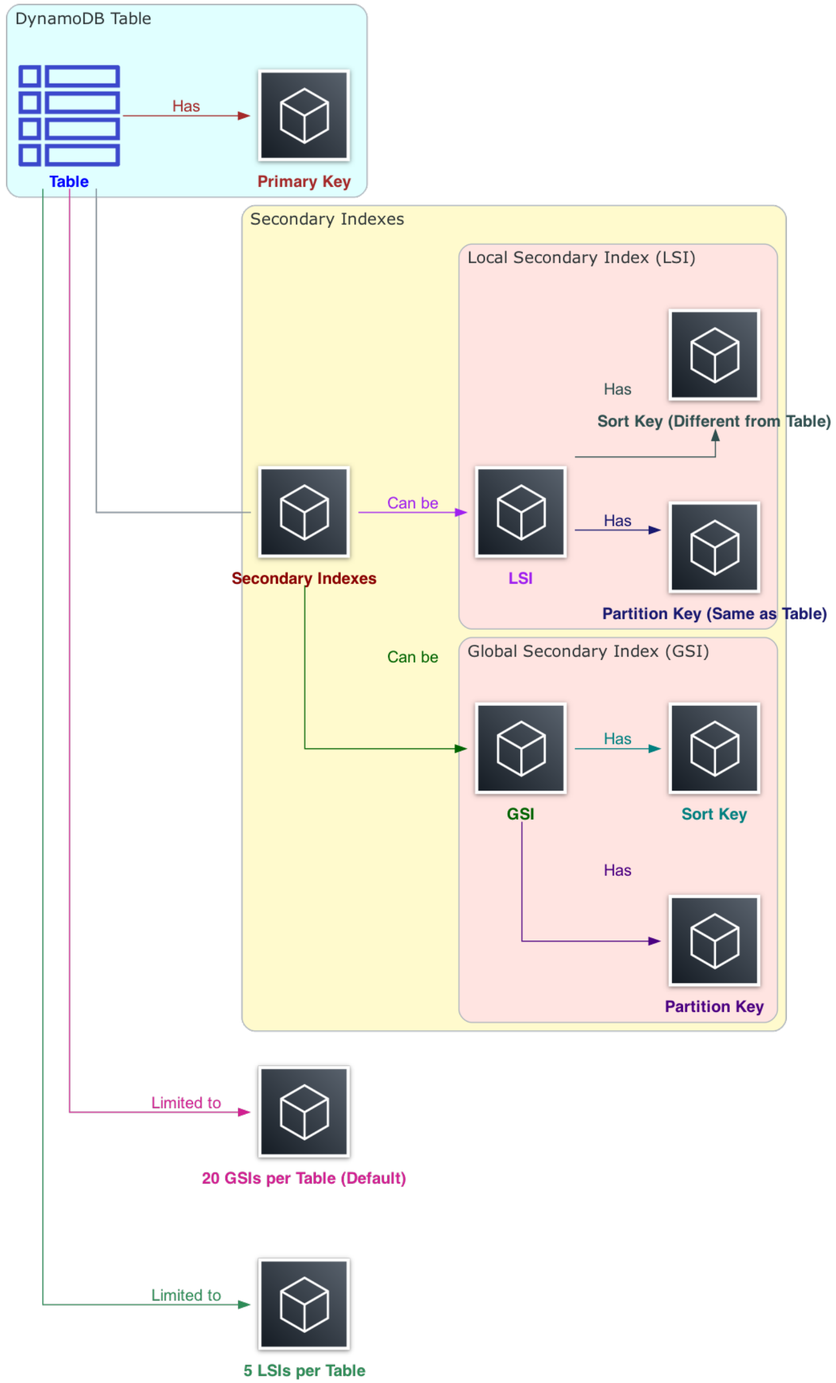
🆔 Uniquely identifies items

Primary Key



2. 🗂️ Two Types of Primary Keys	
🧩 Partition Key	🔐 Simple primary key
	🍯 One attribute
	🔢 Input to hash function
	🪨 Determines storage partition
	❗ No duplicate partition keys
🗃️ Composite Key	🗝️🔢 Partition key + Sort key
	🔢 Partition key input to hash function
	🪨 Determines storage partition
	🗄️ Items with same partition key stored together
	🔠 Sorted by sort key value
	✅ Allows duplicate partition keys
	❗ Must have unique sort keys

Secondary Indexes



1. 🔍 Query data using alternate keys
🗝️ Keys other than primary key
🌈 Flexibility in accessing data

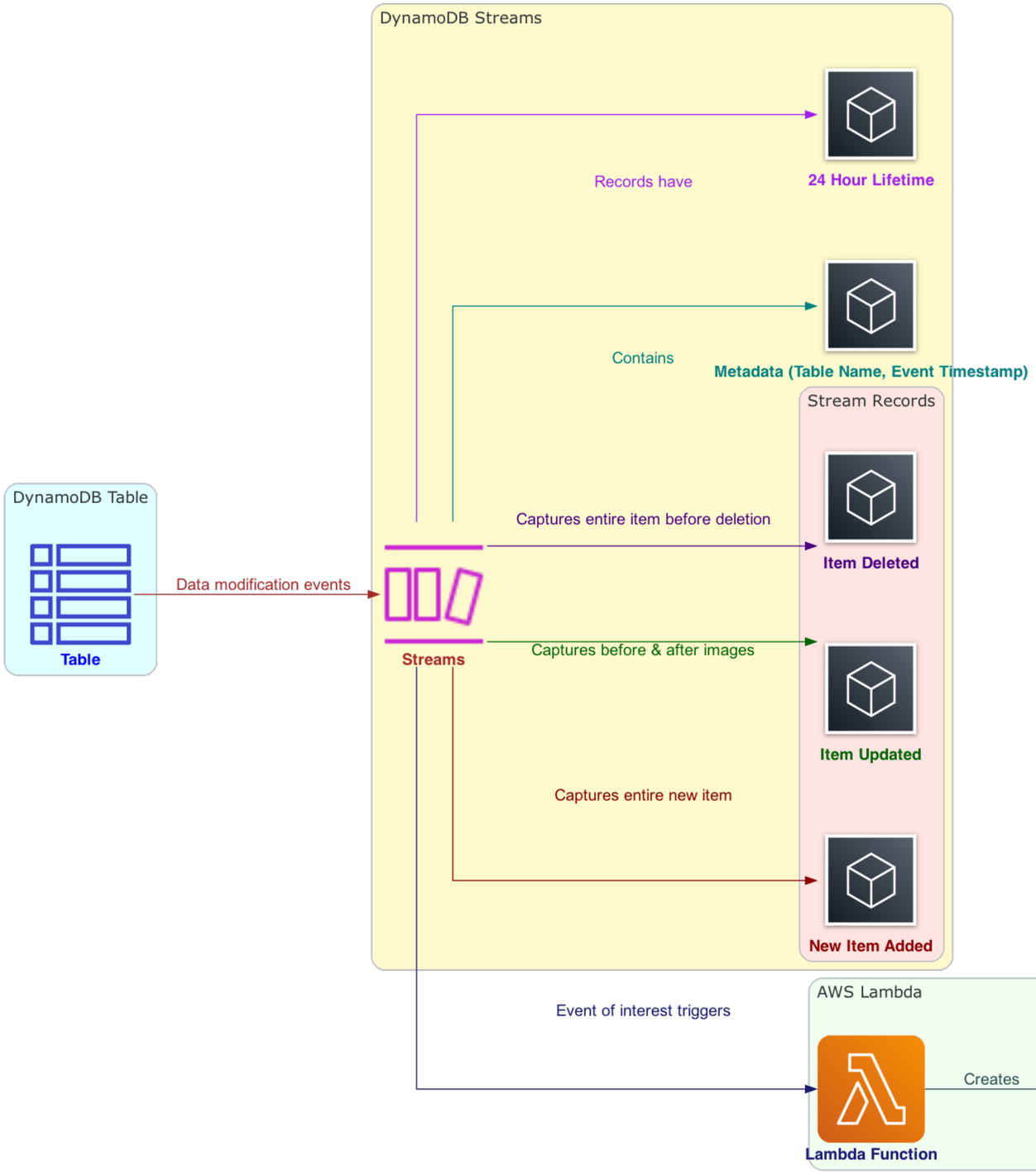
2. 🌍 Global Secondary Index (GSI)
🔑 Different partition and sort keys
🌎 Queries across entire table

3. 🏠 Local Secondary Index (LSI)
🗝️ Same partition key as base table
🔑 Different sort key
🏚️ Query options within partition

4. 🌐 GSIs span the entire table
🗺️ Cover entire table
🚀 Efficient data retrieval

5. 🔢 Default quota
📈 20 GSIs per table
📊 5 LSIs per table
🧰 Ample indexing options

DynamoDB Streams



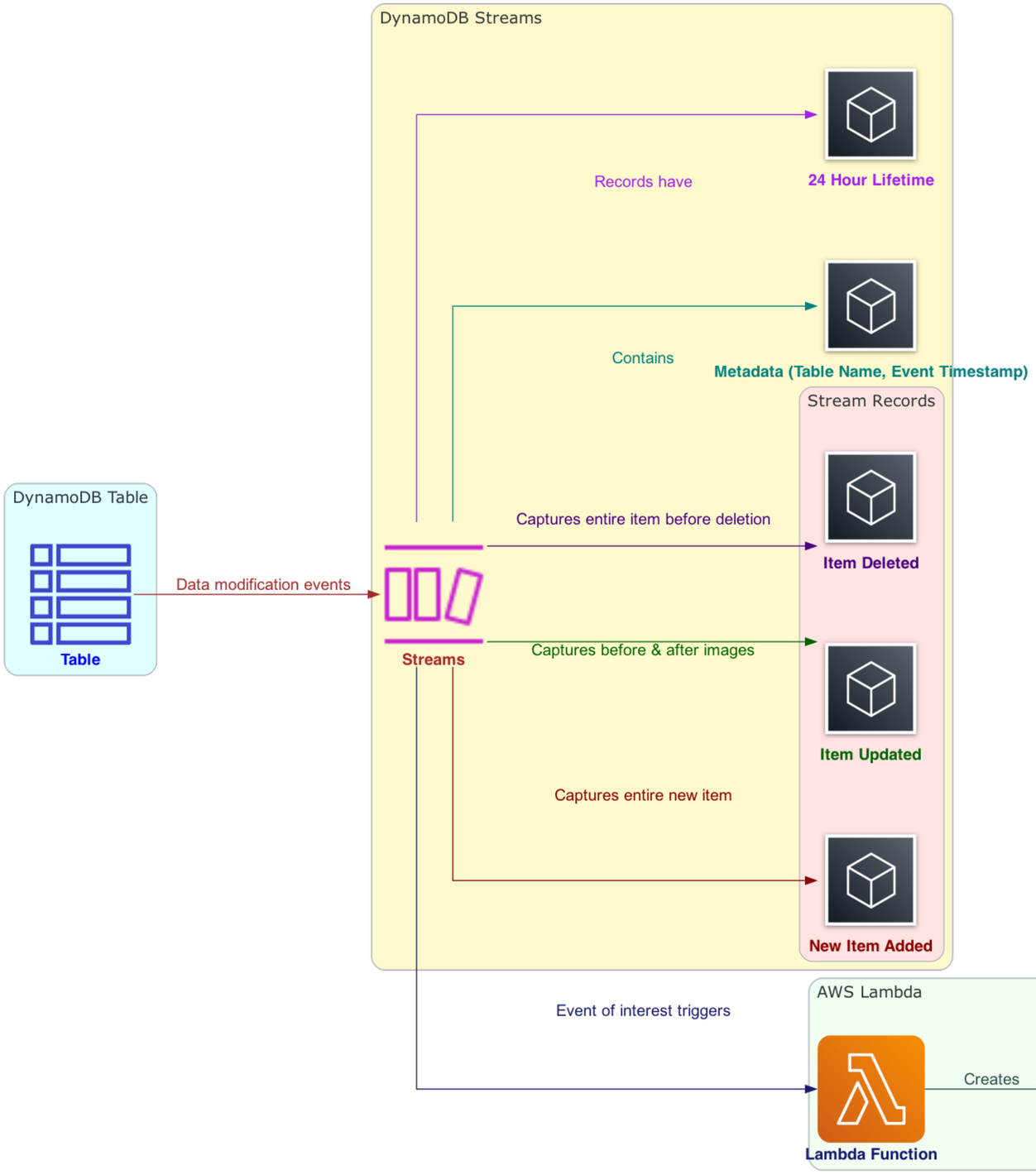
1. 📝 Captures data modification events
➕ Addition of items
🔄 Update of items
➖ Deletion of items

2. ⏰ Near-real time, ordered event stream
🕰️ Timely data capture
📋 Accurate event order

3. 🆕 New item: Captures entire item image
📸 Complete snapshot of new data
🏷️ Includes all attributes

4. 🔄 Updated item: Captures 'before' and 'after' images
🔍 Easy tracking of changes
📊 Modified attributes captured

DynamoDB Streams



❌ Deleted item:

5. Captures pre-deletion item image

💾 Preserves record of removed data

📸 Snapshot before deletion

📊 Includes table name,

6. event timestamp, and metadata

📋 Valuable context for events

⏰ Event timestamp included

7. ⏳ Stream records have a 24-hour lifetime

🔄 Automatically removed after 24 hours

🔍 Focuses on recent events

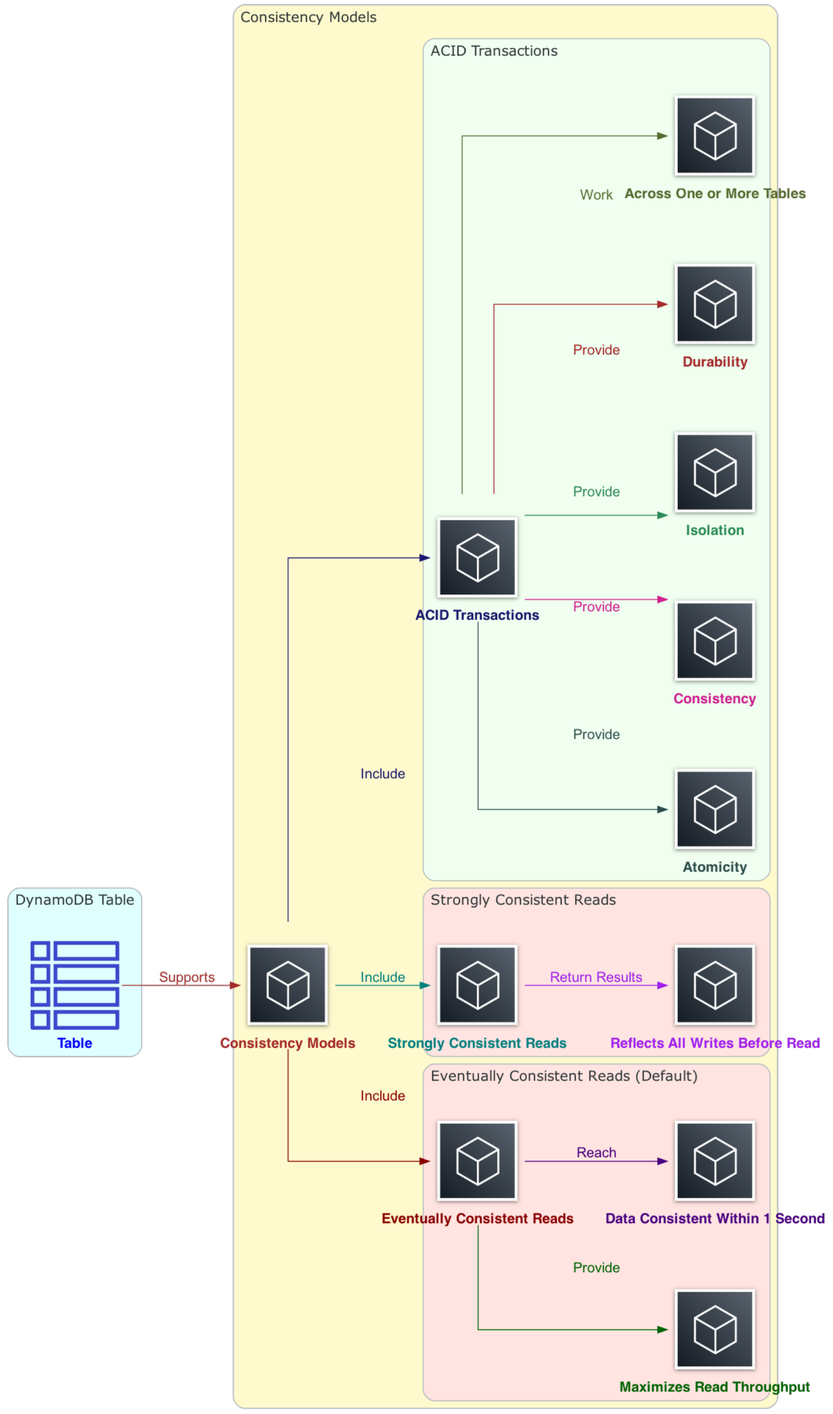
🎣 Trigger AWS Lambda

8. functions on events of interest

⚡ Event-driven architectures

🕰️ Real-time data processing

Consistency Model

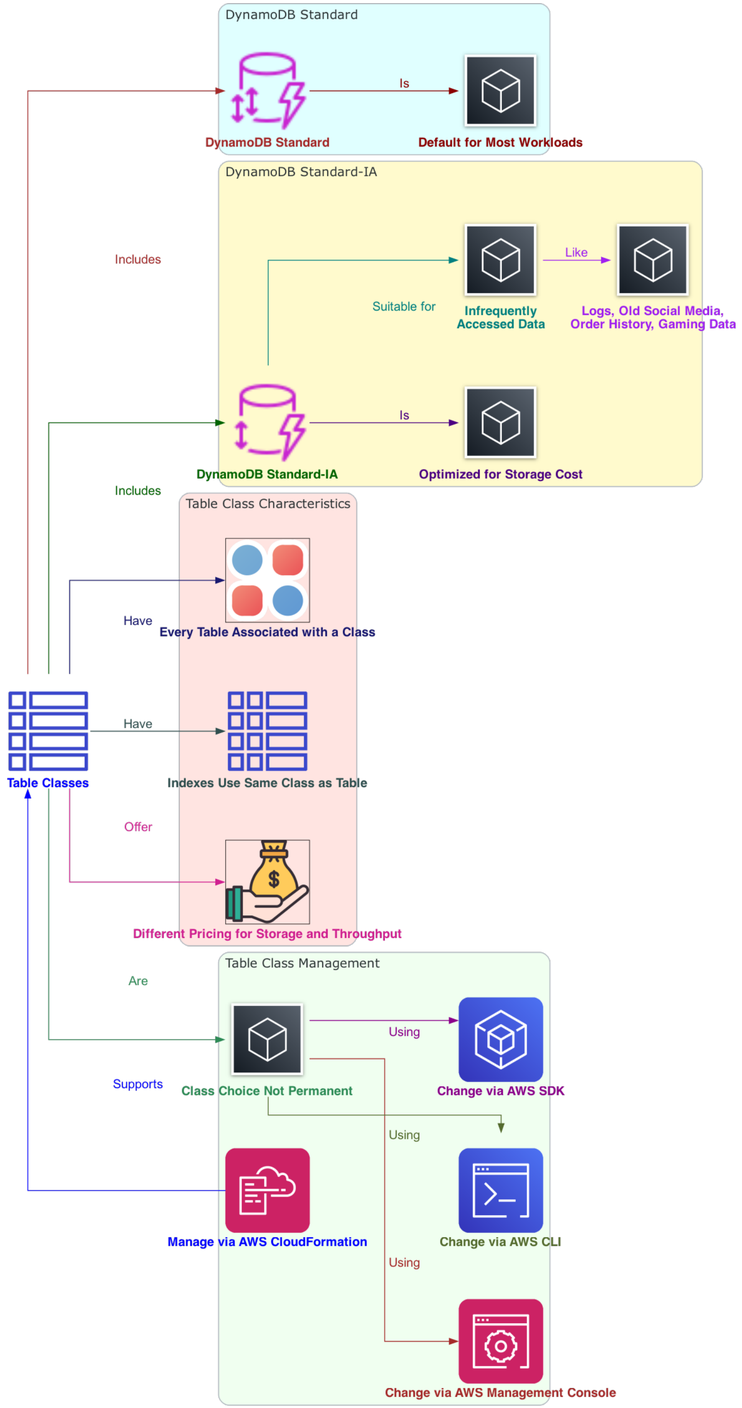


1. ⏱️ Eventually Consistent Reads (Default)
📈 Maximizes read throughput
⚠️ Might not reflect recent writes
⏳ Consistent within 1 second
🔄 Repeat read for updated data

2. 💪 Strongly Consistent Reads
🎛️ Flexibility and control
📝 Returns result reflecting all writes before read

3. 🔒 ACID Transactions
🅰️ Atomicity
🔄 Consistency
🔐 Isolation
💾 Durability
📊 Across one or more tables
🏢 Within single AWS account and region
🔗 Coordinated inserts, deletes, updates
💼 Part of single logical business operation

Table Classes



1. 📊 Two table classes for cost optimization

🎯 Choose suitable option for workload

2. 🔵 DynamoDB Standard

🔧 Default table class

👍 Recommended for most workloads

⚖️ Balance between cost and performance

3. 🟢 DynamoDB Standard-IA

💾 Optimized for storage-dominant cost

📂 Infrequently accessed data

📜 Logs, social media, orders, gaming data

4. 🔗 Every table associated with a class

🔧 Default: DynamoDB Standard

📊 Indexes use same class as table

5. 💰 Different pricing for storage and read/write requests

💸 Select cost-effective class based on usage

6. 🔄 Table class choice is not permanent

🔧 Change as requirements evolve

💸 Flexibility in optimizing costs

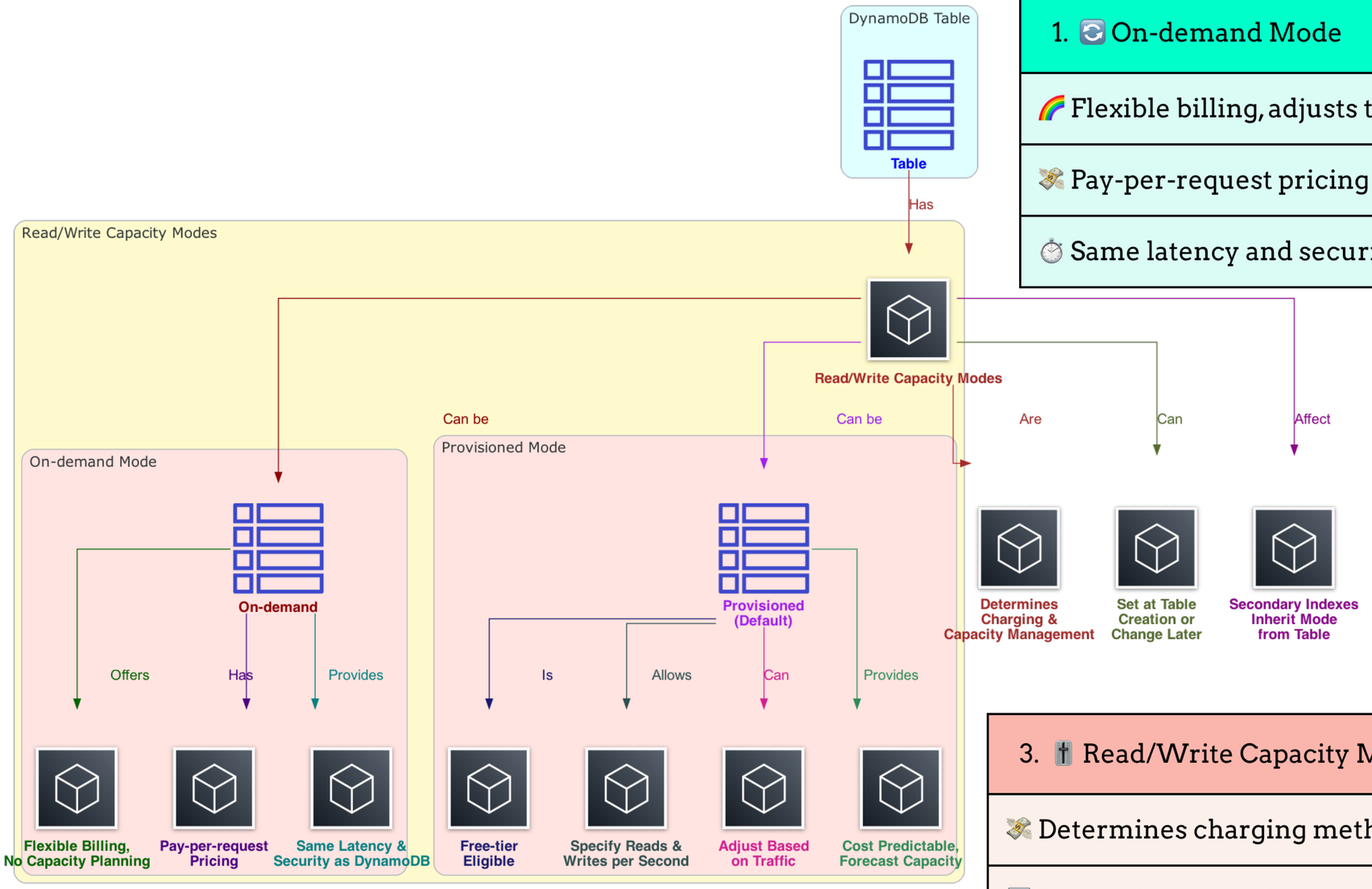
7. 🛠️ Manage table class via various methods

🖥️ AWS Management Console

⌨️ AWS CLI or SDK

☁️ AWS CloudFormation

Amazon DynamoDB Read/Write Capacity Modes



1. 🔄 On-demand Mode

- 🌈 Flexible billing, adjusts to traffic
- 💸 Pay-per-request pricing
- ⏱️ Same latency and security as DynamoDB

2. 📊 Provisioned Mode

- 🆓 Default, free-tier eligible
- 🔢 Specify reads and writes per second
- ⚖️ Adjust capacity based on traffic
- 💰 Cost predictable, forecast capacity

3. 🚦 Read/Write Capacity Mode Significance

- 💸 Determines charging method
- 🗄️ Determines capacity management
- ⏰ Set at table creation or change later

4. 📊 Secondary Indexes Relation

- 👨‍👩‍👧‍👦 Inherit mode from primary table

On-demand Mode

1. 🚀 Thousands of requests per second

📈 High volume of requests

🌊 Handled seamlessly

2. 🧠 No need for capacity planning

🔧 Automatically adjusts capacity

📊 Based on demand

3. 💸 Pay only for what you use

💰 Flexible billing option

🔍 Users pay for actual usage

4. ⚡ Single-digit millisecond latency

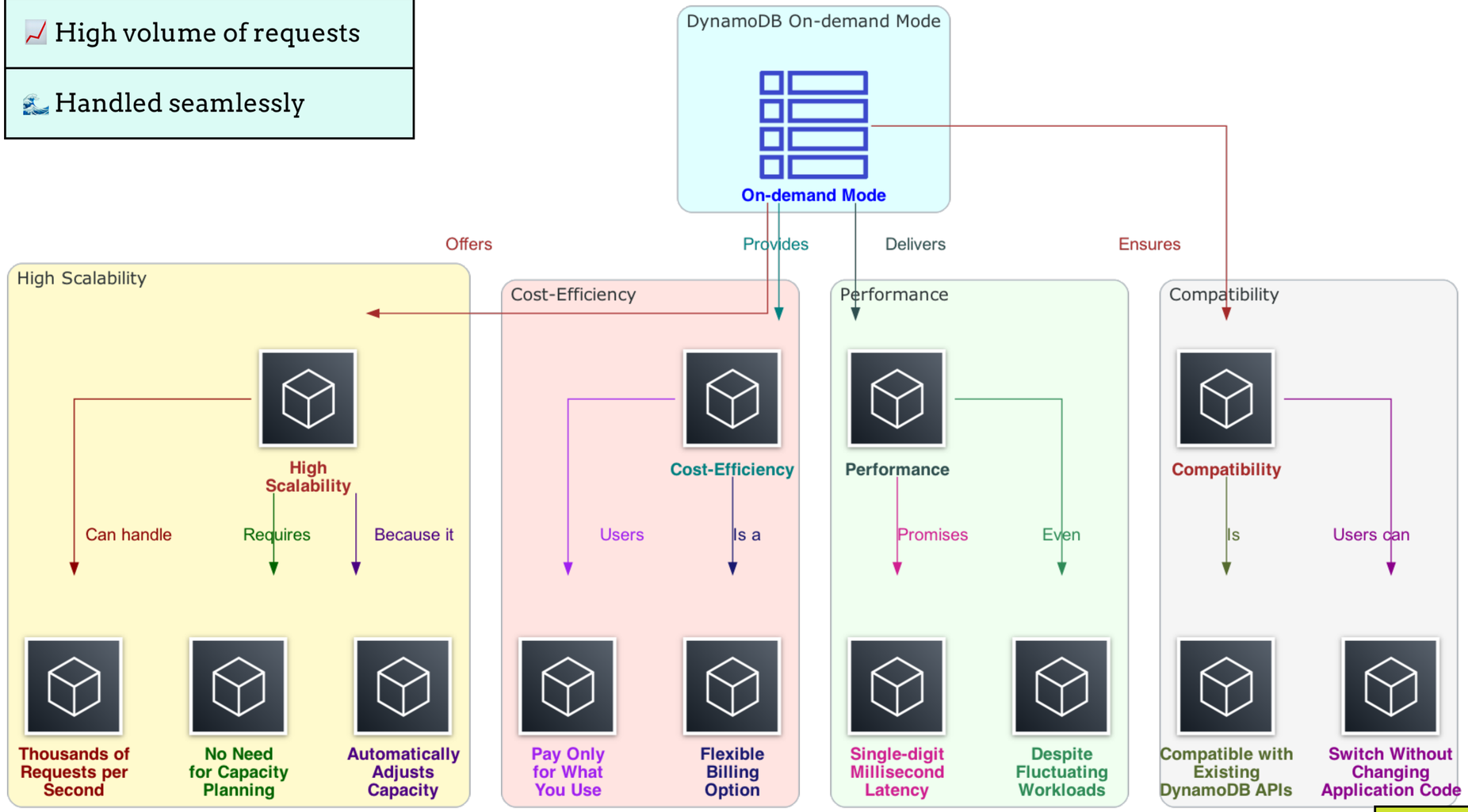
⏰ Despite fluctuating workloads

🚀 DynamoDB promises low latency

5. 🔗 Compatible with existing DynamoDB APIs

🔄 Switch to on-demand mode

💻 Without changing application code

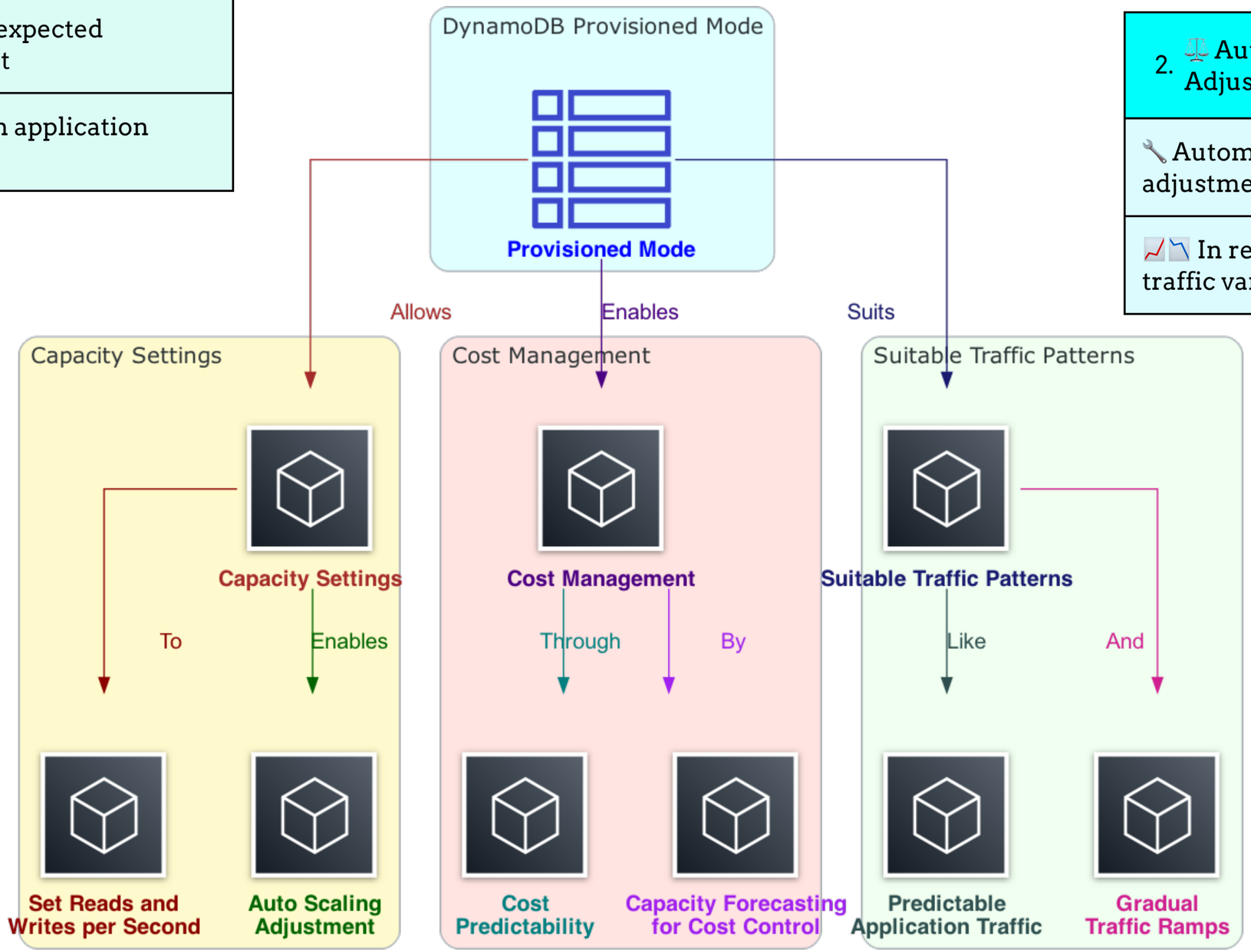


Provisioned Mode

1. 🔢 Set Reads and Writes per Second

📈 Specify expected throughput

🎯 Based on application needs



2. ⚖️ Auto Scaling Adjustment

🔧 Automatic capacity adjustments

📈📉 In response to traffic variations

3. 💰 Cost Predictability

🔮 Forecasting capacity needs

💸 Better cost prediction

4. 📊 Predictable application traffic

🚦 Beneficial for consistent traffic patterns

5. 📈 Gradual traffic ramps

🌱 Suitable for gradual traffic increase

6. 💸 Capacity forecasting for cost control

📅 Anticipating capacity requirements

💰 Better cost management

DynamoDB API Overview

1. 🌐 Control Plane Operations

- 🗄️ Manages tables, indexes, streams, objects

2. 💬 PartiQL

- 📜 SQL-compatible query language
- 🗳️ Executes statements, batch operations

3. ➕ Creating Data Methods

- 📝 PutItem
- 📦 BatchWriteItem

4. 📖 Reading Data Methods

- 🔍 GetItem
- 📩 BatchGetItem
- 🔎 Query
- 🕵️ Scan

5. 🔄 Updating Data Methods

- 🖋️ UpdateItem

6. ➖ Deleting Data Methods

- ❌ DeleteItem
- 📩 BatchWriteItem

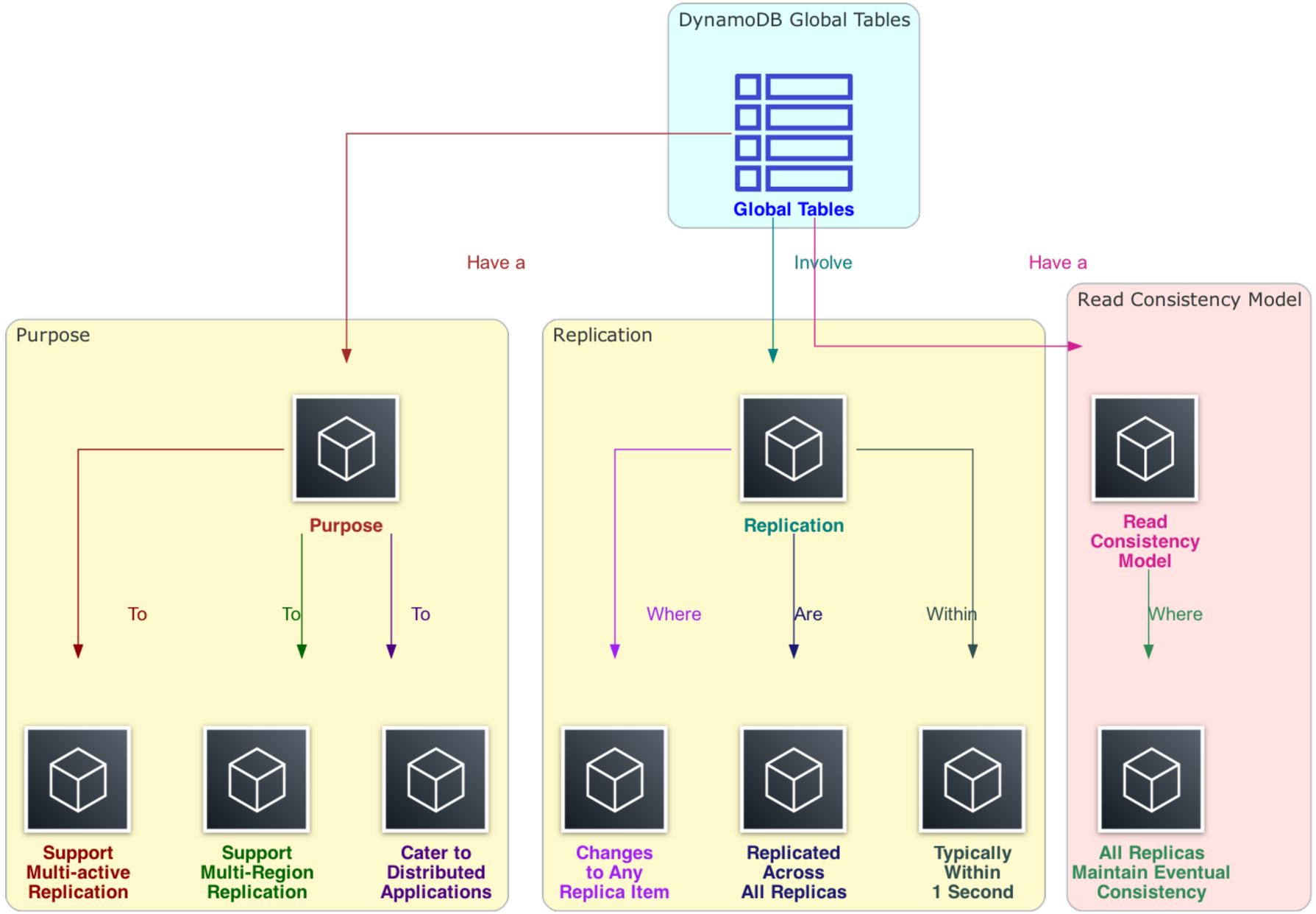
7. 🌊 DynamoDB Streams Operations

- 📜 ListStreams
- 🔍 DescribeStream

8. 💼 Transactions Overview

- 🔒 Provide ACID properties
- ✅ Maintain data correctness
- 🔀 PartiQL or CRUD APIs

Global Tables



1. 🎯 Purpose of Global Tables

- 🔄 Support multi-active replication
- 🌍 Support multi-Region replication
- 🌐 Cater to distributed applications

2. 🔄 Replication

- 🔀 Changes to any replica item
- 📊 Replicated across all replicas
- ⏱️ Typically within a second

3. 📖 Read Consistency Model

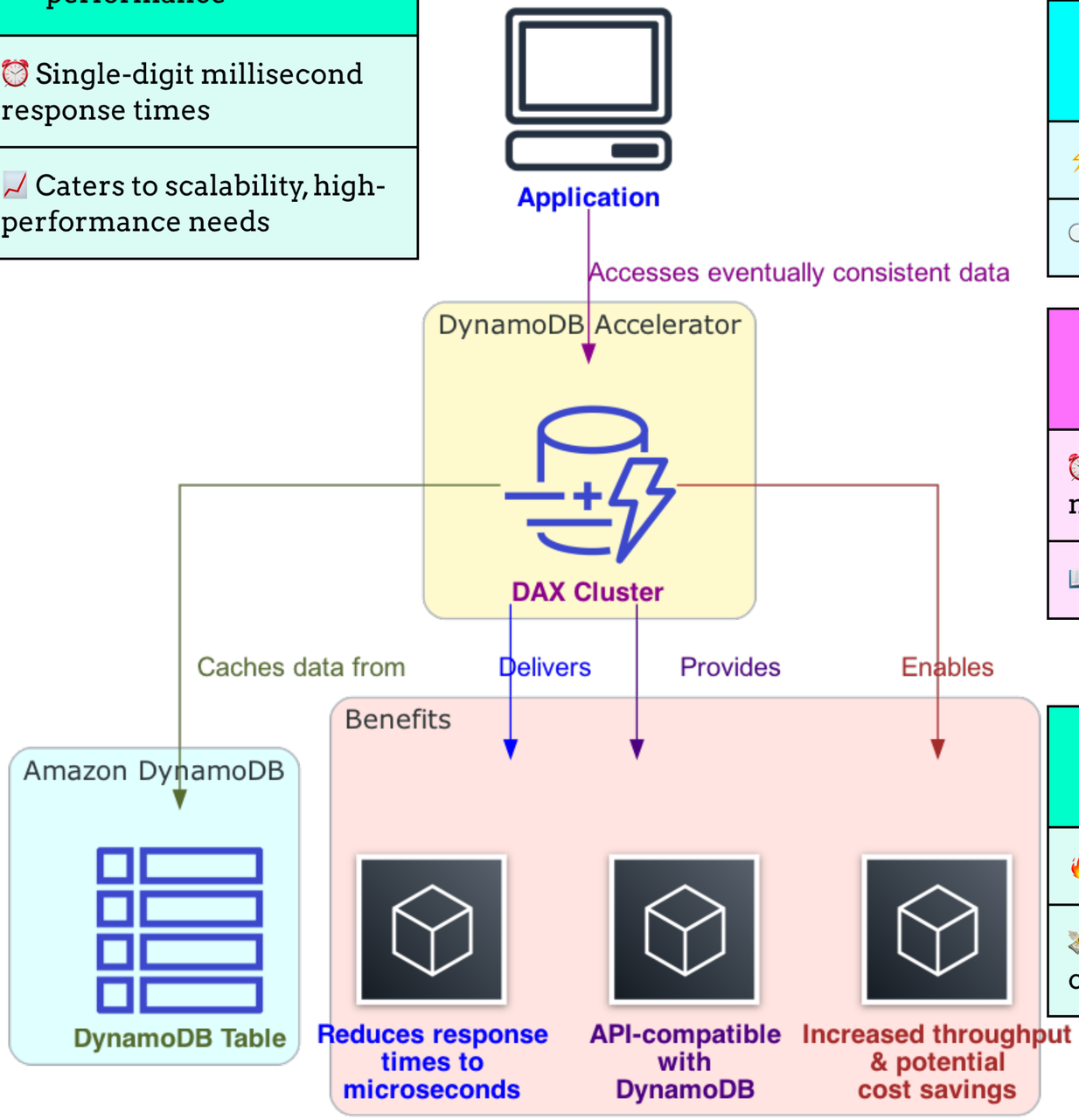
- 🔄 All replicas maintain eventual consistency

DynamoDB DAX

1. 🚀 DynamoDB: Designed for scale and performance

⏰ Single-digit millisecond response times

📈 Caters to scalability, high-performance needs



2. ⏱️ DAX: Microsecond response times for eventually consistent data

⚡ Delivers microsecond latency

🔍 For use cases requiring faster response times

3. 🎭 DAX: DynamoDB-compatible caching service

🔄 Fully compatible with DynamoDB

💨 Enables fast in-memory performance

4. 💾 In-memory cache: Reduces response times by an order of magnitude

⏰ From single-digit milliseconds to microseconds

📖 For eventually consistent read workloads

5. 🧩 Managed service: API-compatible with DynamoDB

🔧 Minimal functional changes to integrate

🔄 Reduces operational complexity

6. 📈 Increased throughput and cost savings for read-heavy workloads

🔥 Provides higher throughput

💸 Reduces need to overprovision read capacity units

7. 🔑 Beneficial for applications with repeated reads for individual keys

🔄 Frequently read same individual keys

⚡ Optimizes performance for repeated reads

Create Table

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5
```

Common Operations

```
# Write item to a table
aws dynamodb put-item --table-name Music --item file://item.json

#Read item from a table
aws dynamodb get-item --table-name Music --item file://item.json

# Delete item from a table
aws dynamodb delete-item --table-name Music --key file://key.json

# Query a table
aws dynamodb query --table-name Music \
--key-condition-expression "ArtistName=:Artist and SongName=:Songtitle"

# Delete a table
aws dynamodb delete-table --table-name Music

# List table names
aws dynamodb list-tables
```

1

2

3

4

5

6

Thanks for Watching