# Analysis of Anti-patterns of Azure Tables Design

## What are examples of anti patterns in azure table design in the context of rowkeys and partition keys, also using a column to store NoSQL data such as json.

When designing tables in Azure Table Storage, it's important to avoid certain anti-patterns to ensure optimal performance, scalability, and cost-efficiency. Here are some key anti-patterns related to RowKey, PartitionKey, and the use of columns to store NoSQL data like JSON:

**1. Poorly Designed Partition Keys**

* **Single Partition Key for All Rows**: Using the same PartitionKey for all rows can lead to hotspots, limiting the scalability and performance of your application since all operations are directed to a single partition.
* **High Cardinality Partition Keys**: Using a unique PartitionKey for every row can prevent the benefits of partitioning, making queries inefficient since they can't leverage the partitioning scheme.

**2. Poorly Designed Row Keys**

* **Sequential RowKeys**: Using sequential RowKeys (like an auto-incrementing number or a timestamp) within a single partition can lead to hotspots because all new records are inserted into the same partition until it fills up.
* **Too Large RowKeys**: Very long RowKeys can increase storage costs and may impact the performance of operations.

**3. Storing Unstructured Data in Columns**

* **Storing Large JSON Documents**: Putting large JSON documents in a single column can make the storage inefficient and querying complex. It can also lead to high data transfer costs and increased latency.
* **Inconsistent Schema**: When JSON documents have inconsistent schemas, it can make it challenging to query the data effectively. Azure Table Storage is designed for structured, schema-less data, so maintaining consistency is important.

**Examples of Anti-Patterns**

**Example 1: Single Partition Key for All Rows**

A black background with white text

Description automatically generated

*Problem*: This causes all user profiles to be in one partition, leading to a bottleneck.

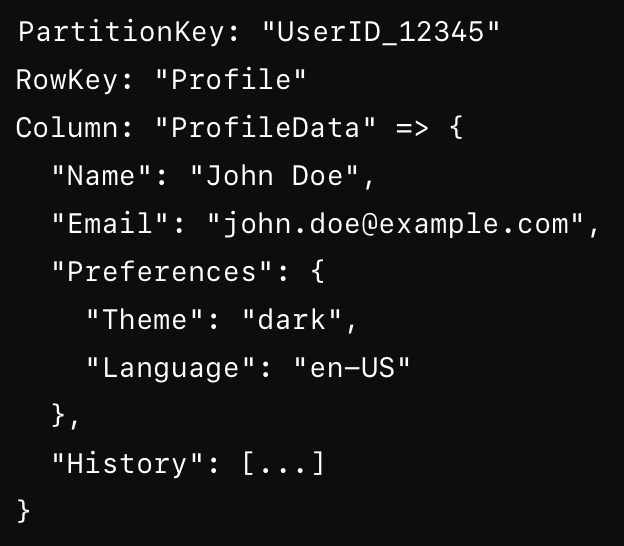
**Example 2: Sequential RowKeys**

A black background with white text

Description automatically generated

*Problem*: This creates a hotspot for insert operations as all new records go to the same partition.

**Example 3: Storing Large JSON in a Single Column**



*Problem*: Difficult to query specific fields within the JSON efficiently and can lead to high storage and retrieval costs.

## Best Practices

* **Design Partition Keys for Even Distribution**: Ensure that PartitionKeys distribute data evenly across partitions to avoid hotspots. For example, use a hash of a user ID or a combination of multiple fields.
* **Non-Sequential RowKeys**: Use GUIDs or other non-sequential identifiers to avoid hotspots within partitions.
* **Store Structured Data**: Instead of storing large JSON blobs, normalize the data into multiple columns or tables. If JSON is necessary, consider using Azure Cosmos DB, which is better suited for JSON data.

By avoiding these anti-patterns, you can ensure better performance, scalability, and cost-efficiency in your Azure Table Storage design.

## Scenario 1 - Storing JSON Request and Response Payloads

In a scenario where you need to store JSON request and response payloads for customers invoking a REST API, certain anti-patterns can negatively impact the performance and manageability of your storage solution. Here are the key anti-patterns and how to avoid them:

**Anti-Patterns in Storing JSON Request and Response Payloads**

**1. Storing Large JSON Blobs in a Single Column**

* **Anti-Pattern**: Storing the entire JSON request and response payload in a single column.
  + **Problem**: Large JSON documents can lead to inefficiencies in storage and make querying specific parts of the data cumbersome and slow. Additionally, it increases the cost of data retrieval and storage.

**2. Using Sequential Keys**

* **Anti-Pattern**: Using sequential PartitionKeys or RowKeys.
  + **Problem**: Sequential keys can create hotspots in your data, where all new data is written to the same partition, leading to performance bottlenecks.

**3. Overloading a Single Partition**

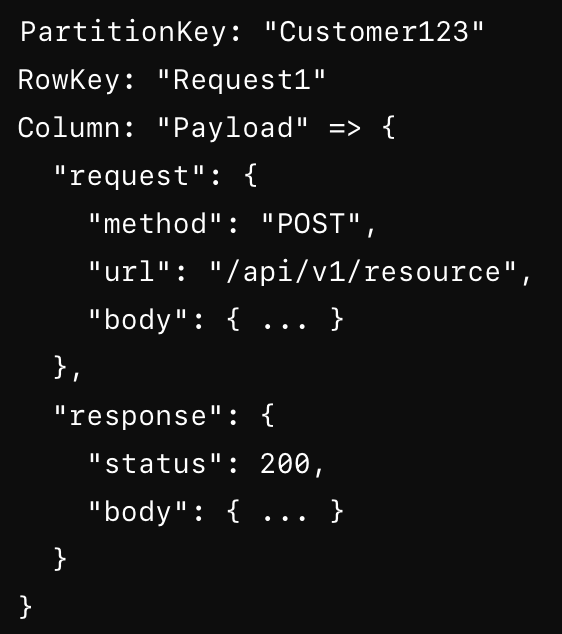
* **Anti-Pattern**: Using a single PartitionKey for all records.
  + **Problem**: This approach leads to all data being written to a single partition, which can cause scalability issues and affect query performance.

**4. Inconsistent Schemas**

* **Anti-Pattern**: Storing JSON documents with inconsistent schemas.
  + **Problem**: This can complicate querying and data processing, as the schema may vary between records, making it difficult to extract meaningful insights.

**Examples of Anti-Patterns**

**Example 1: Storing Large JSON Blobs in a Single Column**



*Problem*: Difficult to query specific parts of the request or response, leading to inefficiencies.

**Example 2: Using Sequential Keys**

*A black background with white text

Description automatically generated*

*Problem*: Sequential RowKeys create hotspots and limit scalability.

**Example 3: Overloading a Single Partition**

*A black background with white text

Description automatically generated*

*Problem*: All records end up in the same partition, causing performance bottlenecks.

**Best Practices to Avoid Anti-Patterns**

1. **Decompose JSON Payloads**: Store specific parts of the JSON payload in separate columns to make querying more efficient.
   * Example:

A computer screen with white text

Description automatically generated

1. **Use Non-Sequential Keys**: Use a combination of timestamp and a unique identifier (e.g., GUID) to ensure even distribution.
   * Example:

A black background with white text

Description automatically generated

1. **Distribute Data Evenly**: Design PartitionKeys to distribute data evenly across partitions.
   * Example:

A black background with white text

Description automatically generated

1. **Ensure Consistent Schemas**: Maintain a consistent schema for JSON payloads to facilitate querying and data processing.
2. **Use Appropriate Storage Services**: If the JSON payloads are large or require complex querying, consider using Azure Cosmos DB instead of Azure Table Storage. Cosmos DB is optimized for JSON and provides rich query capabilities.

By avoiding these anti-patterns and following best practices, you can ensure that your storage solution for JSON request and response payloads is efficient, scalable, and cost-effective.

## Scenario 2 – Design Patterns for Efficient table structure that stores JSON

Designing an efficient table structure for storing JSON request and response payloads in Azure Table Storage requires careful consideration of key attributes and data distribution. Here’s a good design pattern to ensure optimal performance, scalability, and manageability:

**Design Pattern for Table Structure**

**Key Components:**

1. **PartitionKey**: Designed to evenly distribute data across partitions.
2. **RowKey**: Uniquely identifies each record within a partition and avoids hotspots.
3. **Columns**: Specific fields from the JSON payload are extracted into separate columns for efficient querying.

**Step-by-Step Design:**

1. **Define the PartitionKey**:
   * Use a composite key that combines customer ID and a time component (e.g., month and year). This helps distribute data evenly across partitions and avoids hotspots.
   * Example: PartitionKey = "{CustomerID}\_{YYYYMM}"
2. **Define the RowKey**:
   * Use a combination of timestamp and a unique identifier (e.g., GUID) to ensure uniqueness and avoid sequential patterns.
   * Example: RowKey = "{Timestamp}\_{GUID}"
3. **Extract Key Fields**:
   * Extract relevant fields from the JSON payload into separate columns to facilitate efficient querying.
   * Example fields: RequestMethod, RequestUrl, ResponseStatus
4. **Store JSON Payloads**:
   * Store the original JSON payloads in separate columns if needed for complete records, but ensure they are manageable in size.
   * Example fields: RequestBody, ResponseBody

**Example Table Structure:**

**Columns:**

* **PartitionKey**: {CustomerID}\_{YYYYMM}
* **RowKey**: {Timestamp}\_{GUID}
* **RequestMethod**: POST
* **RequestUrl**: /api/v1/resource
* **ResponseStatus**: 200
* **RequestBody**: JSON string of the request body
* **ResponseBody**: JSON string of the response body
* **Timestamp**: Standard Azure Table Storage timestamp (automatically managed)

**Example Record:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PartitionKey | RowKey | Request Method | Request  Url | Response  Status | Request  Body | Response  Body | Timestamp |
| Customer123\_202308 | 20230805T153045Z\_abcd1234 | POST | /api/v1/resource | 200 | { "param": "value" } | { "result": "success" } | 2023-08-05T15:30:45Z |

**Benefits of This Design:**

1. **Even Data Distribution**: The combination of customer ID and a time component in the PartitionKey helps distribute the data evenly across partitions, avoiding hotspots and ensuring scalability.
2. **Unique and Efficient RowKeys**: Using a timestamp combined with a GUID in the RowKey ensures each record is unique and prevents sequential key issues.
3. **Efficient Querying**: Extracting key fields from the JSON payload into separate columns allows for efficient querying without needing to parse the entire JSON blob.
4. **Manageable JSON Storage**: Storing the JSON payloads in separate columns keeps the data manageable and allows for retrieval of complete records when necessary.
5. **Time-Based Queries**: This design facilitates time-based queries, making it easier to retrieve records for specific periods, which is common in many logging and auditing scenarios.

**Example Query Scenarios:**

1. **Retrieve All Requests for a Customer in a Specific Month**:
   * Query by PartitionKey:

PartitionKey = "Customer123\_202308"

1. **Retrieve Specific Request by Timestamp and GUID**:
   * Query by PartitionKey and RowKey:

PartitionKey = "Customer123\_202308"

AND RowKey = "20230805T153045Z\_abcd1234"

1. **Filter Requests by Method and Status**:
   * Query by PartitionKey with filters:

PartitionKey = "Customer123\_202308"

AND RequestMethod = "POST"

AND ResponseStatus = "200"

By following this design pattern, you can create a table structure in Azure Table Storage that is optimized for performance, scalability, and efficient querying.

## What are good books or cloud vendor resource links that can be used as reference?

Several good books and cloud vendor resources that can help you design and manage efficient table structures in Azure Table Storage, as well as understand broader concepts in cloud computing, are:

**Books**

1. **"Designing Data-Intensive Applications" by Martin Kleppmann**
   * This book provides a comprehensive guide to designing scalable and maintainable systems. It covers a wide range of topics, including data modeling, storage, and distributed systems.
   * [Amazon Link](https://www.amazon.com/Designing-Data-Intensive-Applications-Reliable-Maintainable/dp/1449373321)
2. **"Azure for Architects: Create secure, scalable, high-availability applications on the cloud" by Ritesh Modi**
   * This book provides in-depth insights into designing applications on Azure, including storage solutions and best practices.
   * [Amazon Link](https://www.amazon.com/Azure-Architects-secure-scalable-high-availability/dp/178899173X)
3. **"Cloud Architecture Patterns: Using Microsoft Azure" by Bill Wilder**
   * This book covers various cloud architecture patterns and how to implement them using Azure, including data storage patterns.
   * [Amazon Link](https://www.amazon.com/Cloud-Architecture-Patterns-Microsoft-Azure/dp/1449319777)

**Cloud Vendor Resources**

1. **Microsoft Azure Documentation**
   * The official documentation for Azure Table Storage provides comprehensive guides, best practices, and examples.
   * [Azure Table Storage Documentation](https://docs.microsoft.com/en-us/azure/storage/tables/table-storage-overview)
2. **Azure Architecture Center**
   * This resource provides architecture best practices, design patterns, and reference architectures for building solutions on Azure.
   * [Azure Architecture Center](https://docs.microsoft.com/en-us/azure/architecture/)
3. **Azure Storage Blog**
   * The official blog offers updates, tips, and best practices for using Azure Storage services, including Table Storage.
   * [Azure Storage Blog](https://techcommunity.microsoft.com/t5/azure-storage/bg-p/AzureStorage)
4. **Microsoft Learn**
   * Microsoft Learn offers free, interactive learning paths and modules for Azure, including data storage and management.
   * [Microsoft Learn - Azure](https://docs.microsoft.com/en-us/learn/azure/)
5. **Azure Patterns and Practices**
   * This resource provides a collection of patterns and practices for designing and implementing solutions on Azure.
   * [Azure Patterns and Practices](https://docs.microsoft.com/en-us/azure/architecture/patterns/)

**Additional Resources**

1. **"The Azure Handbook" by Reza Salehi**
   * This is a practical guide to Azure services and includes detailed explanations of various storage solutions.
   * [Amazon Link](https://www.amazon.com/Azure-Handbook-practical-Microsoft-solutions/dp/1801071096)
2. **"Pro Azure Governance and Security" by Peter De Tender, David Rendon, Samuel Erskine, and Alfredo Deza**
   * This book covers best practices for governance and security in Azure, which is crucial for managing storage solutions.
   * [Amazon Link](https://www.amazon.com/Pro-Azure-Governance-Security-Practices/dp/1484267492)

By leveraging these resources, you can gain a solid understanding of Azure Table Storage, best practices for cloud architecture, and efficient data management techniques.