Choosing a partitioning strategy | Multitenant | Scalable

- Requirement is to create partition on the transactional db to be able to support multi tenant architecture.
- Ability to fetch at least 5-7 days of data with less read throughput and less TAT.
- Given this telemetry , we should identify suitable partition keys in terms of efficient read/write throughput:

```
{
    "siteClassId": "US-5260$$$Fryer",
    "deviceId": "Giles-7163712301905",
    "metadata": {
        "messageId": "944123296270382202166526906952195024073085289336 3",
        "messageType": "device_readings",
        "messageVersion": "1.3",
        "messageTimestamp": "2021-11-16T19:19:59.658Z",
        "manufacturerName": "Giles",
        "manufacturerDeviceModel": "EOF-BIB/LT/24/24",
        "manufacturerSerialNumber": "7163712301905",
        "manufacturerFamily": null,
        "customerGuid": "2019-002997",
        "customerName": "Left Fryer Vat",
        "customerDeviceClass": "Fryer"
    },
    "headers": {
        "tenant-id": "43f4b965-aa85-405a-43d7-08d958688822",
        "source-type": "EdgeModule",
        "site-id": "US-5260",
        "edge-device-id": "US-store-05260-01",
        "edge-module-id": "telemetryreceiver-01"
    },
    "type": "Telemetry",
    "data": {
        "nod": "1",
        "timer1Direction": "0",
        "controlMode1": "3",
        "controlMode2": "0",
        "dataItemType": "0",
        "boardTemp": "78.8",
        "cookCycleNumber": "0",
        "timer1Seconds": "420",
        "addLevelDiff": "0",
        "heatContactorCounter1": "1",
        "inAlarm": "1",
        "heatContactorCounter2": "0",
        "net": "1",
        "realTimeSeconds": "213877354",
        "cookCyclesSinceFilter": "0",
        "cooktimeMinutes": "0",
        "heatingElementsOnMinutes": "11",
        "timer2Direction": "0",
        "idleTimeMinutes": "8",
        "addLevelProbeTemp": "0.0",
        "epochTimeSeconds": "217019999",
        "heatingElementTemp": "164.4",
        "uptimeMinutes": "20",
        "heatingElementsIdleMinutes": "0",
        "oilProbeTemp": "118.9",
        "timer2Seconds": "420",
        "heatingElementSetPoint": "350",
        "basketElevatorCycles2": "1",
        "powerCycleID": "213876154",
        "heatingElementActiveMinutes": "11",
        "basketElevatorCycles1": "1",
        "rightMenuItemString": "MANUAL",
        "leftMenuNumber": "0",
        "rightMenuNumber": "51",
        "leftMenuItemString": "MANUAL",
        "telemetryTimestamp": "2021-11-16T19:19:59Z",
        "localTimestamp": "2021-11-16T13:19:59-06:00"
    },
    "id": "abe3547f-f2f2-4b19-b363-4af294f00cc4",
    "_rid": "x6MMALDeYzwBAAAAAAAAA==",
    "_self": "dbs/x6MMAA==/colls/x6MMALDeYzw=/docs/x6MMALDeYzwBAAAAAAAAAA==/",
    "_etag": "\"0f0039e9-0000-0700-0000-619404600000\"",
    "_attachments": "attachments/",
    "_ts": 1637090400
}
```

For **all** containers, your partition key should:

- Be a property that has a value which does not change.
- Should only contain `String` values - or numbers should ideally be converted into a `String` e.g GUID.
- Have a high cardinality. In other words, the property should have a wide range of possible values.
- Spread request unit (RU) consumption and data storage **evenly** across all logical partitions. This ensures even RU consumption and storage distribution across your physical partitions.

  Throughput constraint
  Note that the maximum *RUPS per physical partition* is 10,000 units.

  Storage constraint
  Each individual *physical partition* can store up to 50GB data.

  What is the expected throughput (request unit per sec) from each site/tenant?
  Frequency for request throughput is every 30 sec.

Partition key candidates:

A good choice of partition key is a key that occurs frequently in your filters while querying.

- tenant-id
- siteId
- deviceId

We will understand considering each of the above keys :

`/tenant-id` :

Current maximum throughput of prod *deli-telemetry* container is 4000 RU/s.
Microsoft guarantees that the all the documents with the same PartitionKey are stored in the same Logical Partition and ultimately in a single physical machine.

Initial thought was to keep telemetry of a tenant in one single machine so that even if we need to scaleout it would still have data of at least one **tenant** in one single machine.That means if later we query the data within the same partition (e.g. telemetry data for one tenant), the query runs over a dataset in a single physical machine and therefore, it does not have to travel over the network and collect data from different data storages! That brings the **maximum speed** to us.
Let's say partition key is **/tenant-id** and a tenant has number of sites more than 4000, then it would demand scale out.

Storing all the documents with the same partitionKey in a single machine, brings up a size limit! Maximum size of a logical partition should not exceed 20 GB.
Considering above , **/tenant-id** is not a suitable partition key choice.

`/siteId` :
If partition key is narrowed down to siteId , let's understand that the storage constraint is bound to violate in near-future. That makes it not a suitable choice.

`/deviceId` :
/deviceId is a potential candidate considering device telemetry received every 30 sec will be evenly distributed.

But when storage is full , scale out will require cross-partition queries for a tenant.

Isn't deviceId too narrowed down? It might make the throughput per logical partition underutilized.
Each device generates telemetry every 30 secs. It will never even reach the minimum 10 RU/s criteria (refer this)

It would be good if we can couple */deviceId* with some other keys and form **synthetic partition keys**.

We can create synthetic partition keys by concatenating multiple property values into a single artificial `partitionKey` property.
Azure Cosmos DB now supports large partition keys with length up to **2 KB**.

While creation Microsoft recommends using a :

partition key with a random suffix
Another possible strategy to distribute the workload more evenly is to append a random number at the end of the partition key value. When you distribute items in this way, you can perform parallel write operations across partitions.
For example we can concatenate **`deviceId`** and **`date`**

`deviceId` will have random values and while querying we can use **`date`** part for fetching last few days data.
Because you randomize the partition key, the write operations on the container on each day are spread evenly across multiple partitions. This method results in better parallelism and overall higher throughput.

partition key with pre-calculated suffixes

The random suffix strategy can greatly improve write throughput, but it's difficult to read a specific item. You don't know the suffix value that was used when you wrote the item. To make it easier to read individual items, use the pre-calculated suffixes strategy. Instead of using a random number to distribute the items among the partitions, use a number that is calculated based on something that you want to query.

In our case it can be `**deviceId**` can be a random value known to us.

Let's see few potential synthetic partition keys below:

`tenantId` + `siteId` + `deviceId` is a good potential candidate but over the time a **device** at a **site** may generate data which exceeds storage constraint. Hierarchical partition keys will help overcome this.

Hierarchical partition keys

Azure Cosmos DB distributes your data across logical and physical partitions based on your partition key to enable horizontal scaling.
With hierarchical partition keys, or subpartitioning, you can now configure up to a three level hierarchy for your partition keys to further optimize data distribution and enable higher scale.
If you use **synthetic keys** today or have scenarios where partition keys can exceed 20 GB of data, subpartitioning can help.

With this feature, logical partition key prefixes can exceed *20 GB* and *10,000 RU/s*, and queries by prefix are efficiently routed to the subset of partitions with the data.

In our multi-tenant scenario where we store telemetry for `**device**` at `**sites**` under each `**tenant**`.
In a real world scenario, some tenants can grow large with thousands of users, while the many other tenants are smaller with a few users. Partitioning by `**/tenantId**` may lead to exceeding Azure Cosmos DB's *20GB* storage limit on a single logical partition, while partitioning by `*/siteId*` may exceed storage (eventually) and queries will be tenant cross-partition.

If we are hierarchically using `*/tenantId*` + `*siteId*` + `*deviceId*` there will be sub partitions created.
When a physical partition exceeds 50GB of storage, Azure Cosmos DB will automatically split the physical partition so that roughly half of the data will be on one physical partition, and half on the other. Effectively, subpartitioning means that a single **tenantId** can exceed *20GB* of data, and it's possible for a **tenantId** data to span multiple physical partitions.

While querying we specify any of below combinations and query will be efficiently routed to only the subset of physical partitions that contains the relevant data:
`*tenantId*`
`*tenantId*` + `*siteId*`
`*tenantId*` + `*siteId*` + `*deviceId*`

Specifying the full or prefix subpartitioned partition key path effectively avoids a full **fan-out query**.

For example, if the container had 1000 physical partitions, but a particular **tenantId** was only on five of them, the query would only be routed to the much smaller number of relevant physical partitions.
This will reduce significantly reduce the querying time.

---

1)Determine the partitioning strategy in line with multi tenancy design and scaling requirements.

`/tenantId` + `siteId` + `deviceId` as horizontal partition scheme is a good choice.

---

2)Evaluate the query execution time with the decided partitioning strategy.

---

3) Evaluate all the filters that might be included in the Cosmos query

Filtering using partition keys such as *tenantId*, *siteId*, *deviceId* as mentioned here.
Additionally we can leverage **point queries.**
Point queries demands choosing your custom id (group of filters concatenated together to form a unique id).
We can concatenate `*date*`+`*random bits*` as **Id** (maximum length of Id is 1023 bytes).
`*date*`+`*random bits*` will satisfy the unique constraint on **Id**.

We can use like/contain operator against *date* while `point querying` the **Id**.

References :

- Azure Synapse link : https://learn.microsoft.com/en-us/azure/cosmos-db/synapse-link
- Custom partitioning in Azure Synapse Link : https://learn.microsoft.com/en-us/azure/cosmos-db/custom-partitioning-analytical-store
- Common Azure Cosmos DB use cases : https://learn.microsoft.com/en-us/azure/cosmos-db/use-cases
- Multitenancy and Azure Cosmos DB : https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/service/cosmos-db
- Create a synthetic partition key : https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/synthetic-partition-keys#use-a-partition-key-with-a-random-suffix
- Manage indexing policies in Azure Cosmos DB : https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/how-to-manage-indexing-policy?tabs=dotnetv3%2Cpythonv3