

# Motivation

At the moment Apache Ignite 3 has the data layout, which was built around the autonomous table partitions. It means that every table has its own independent data distribution, which is calculated for it by the affinity function `aff(dataNodes, partitions, replicas)`. Where:

- `dataNodes` is the list of dataNodes, which are available for table distribution in the current table zone.
- `partitions` is the number of table partitions.
- `replicas` is the number of partition replicas.

This way was suitable till present, but now we need to rethink this approach, because we need to take into account **collocation** feature. It means that we want to have a mechanism to mark some tables as a group of colocated tables.

But before we can implement the collocation feature itself, we need to do some preparations with the general UX. This document describes them and the next will describe the changes in our architecture, which will support these UX changes.

## Current state

Current node storage configurations looks like:

```
Unset
rocksDb:
  flushDelayMillis: 1000
  regions:
    lruRegion:
      cache: lru
      size: 256
    clockRegion:
      cache: clock
      size: 512

aipersist:
  checkpoint:
    checkpointDelayMillis: 100
  regions:
```

```
segmentedRegion:
  replacementMode: SEGMENTED_LRU
clockRegion:
  replacementMode: CLOCK
```

The zone creation looks like this:

```
Unset
create zone z1 engine 'aipersist' with
  partitions=1,
  replicas=3,
  data_nodes_filter='${?(@.region == "US" && @.type == "OLTP")}',
  dataregion = 'segmentedRegion';
```

At the same time table creation looks like this:

```
Unset
create table t1 with primary_zone='z1';
```

## Problem statement

Current state has the following drawbacks, which isn't connected with collocation feature directly, but for the consistent UX we need fix them as a prerequisite:

- Region-based naming can be suitable for one engine and not suitable for others, for example aipersist has regions, but RocksDb has not. [\[1\]](#)
- An engine and dataregion must be a part of table declaration, not the zone one. [\[2\]](#), [\[3\]](#)
- A primary\_zone is a bad naming, because the Apache Ignite 3 has the only one zone at all. [\[3\]](#)
- We need to have a guarantee that the table with chosen storage configuration can be deployed on **any node from the zone**. [\[2\]](#)
- There is an unnecessary limitation, that we can have multiple regions of one storage instance, but can't have multiple storage instances. [\[1\]](#)

From the data collocation perspective:

- We need to have an ability to collocate the tables with different storage engines and configurations.

## Decision

### ***Unify the storage references***

To simplify the UX around the storage configurations and appropriate references to them from the table descriptions etc. we need to introduce some kind of unified storage aliases, which are suitable for the different storage engines.

After some discussions we found out that we can use *storage profile* abstraction for this purpose. *storage profile* is a pair of (storage engine type, data space inside the storage instance). With this abstraction the configuration above will look like:

```
Unset
storages:
  engines:
    aipersist:
      checkpoint:
        checkpointDelayMillis: 100
    rocksDb:
      flushDelayMillis: 1000
  profiles:
    lru_rocks:
      engine: rocksDb
      cache: lru
      size: 256

    clock_rocks:
      engine: rocksDb
      cache: clock
      size: 512

    segmented_aipersist:
      engine: aipersist
      replacementMode: SEGMENTED_LRU

    clock_aipersist:
```

```
engine: aipersist
replacementMode: CLOCK
```

## Zone declaration

From the zone declaration user must receive the maximum information, if the zone nodes are compatible with table requirements. We already have a flexible list of node attributes for this purpose. But it seems that storage profiles can't be just a part of the already existing filters. The thing is not every filter query is compatible with the intention "every zone node supports all storage profiles". For example, we can write the filter like this:

Unset

--*WRONG APPROACH*

```
create zone z1 data_nodes_filter='${?(@.storageProfile ==
"lru_rocks" || @.storageProfile == "clock_aipersist")}]';
```

So, with this type of filter, when we create the table with `storageProfile=lru_rocks` we can't be sure that this table can be deployed on any zone node. This breaks the main concern: **table with chosen storage configuration can be deployed on any node from the zone**.

To fix this issue, we need to have the separate simple list of storage profiles, which any zone node must support:

Unset

```
create zone z1 with storage_profiles="lru_rocks,clock_aipersist";
```

Other storage attributes, like engine and dataregion must be removed from the zone declaration.

## Table declaration

Now we have all ingredients to fix the table declaration:

- Storage profiles must replace the separate engine and dataregion attributes.

Unset

```
create table t1 with storage_profile="lru_rocks" using zone="z1";
```

Moreover, because we lifted the available node storages configuration to the zone declaration, we can fail immediately if the user will try to create the table with non-existence (from the zone point of view) storage profile with obvious message:

Unset

```
create zone z1 with storage_profiles="lru_rocks,clock_aipersist";
```

```
create table t1 with storage_profile="segmented_aipersist" using  
zone="z1"; -- fast fail with the detailed message like  
-- "There is no available storage_profile="segmented_aipersist" in the  
zone=z1, only [lru_rocks,clock_aipersist]"
```

## Examples

Nodes configurations:

Unset

```
# Node1  
storages:  
engines:  
  aipersist:  
    checkpoint:  
      checkpointDelayMillis: 100  
profiles:  
  lru_rocks:  
    engine: rocksDb  
    cache: lru  
    size: 256  
    flushDelayMillis: 1000
```

```
clock_aipersist:
  engine: aipersist
  replacementMode: CLOCK

# Node2
storages:
  engines:
    aipersist:
      checkpoint:
        checkpointDelayMillis: 100
  profiles:
    lru_rocks:
      engine: rocksDb
      cache: lru
      size: 256
      flushDelayMillis: 1000

# Node3
storages:
  engines:
    aipersist:
      checkpoint:
        checkpointDelayMillis: 100
  profiles:
    lru_rocks:
      engine: rocksDb
      cache: lru
      size: 256
      flushDelayMillis: 1000

clock_aipersist:
  engine: aipersist
  replacementMode: CLOCK
```

Zones:

Unset

```
create zone z1 with storage_profiles="lru_rocks,clock_aipersist"; --  
(Node1, Node3)  
create zone z2 with storage_profiles="lru_rocks"; -- (Node1, Node2,  
Node3)  
create zone z3 with storage_profiles="clock_aipersist"; -- (Node1,  
Node3)
```

## Known issues

This approach is based on the manual description of supported profiles on zone creation.

Unset

```
create zone z1 with storage_profiles="lru_rocks";
```

But what about the storage profiles update? Obviously we need to implement the altering like:

Unset

```
alter zone z1 with storage_profiles="lru_rocks,clock_aipersist";
```

But at this moment the zone will lose all nodes, which does not support clock\_aipersist profile. As a result we can ruin the normal operation of the whole zone. So, the alter zone operation must be executed with great care.

## Alternative approaches

### Avoid storage profiles declaration on the zone side

In general another approach is available for sure. For example we can skip the storage profiles in the zone description and try to infer it at runtime.

It means that when we create table:

Unset

```
create table t1 with storage_profile="lru_rocks" using zone="z1";
```

- The zone z1 is validating about the fact, that all nodes support the lru\_rocks profile
- If table created successfully: it means, that this zone will reject the nodes without this profile in future

But this behavior is difficult to understand from the user point of view:

- The current zone data nodes is the dynamic property and on the moment of table creation the user can catch random errors according to the list of current alive/joined nodes
- And visa versa: the zone data nodes will depend on the current zone tables (actually the storage profiles of these tables)
- Instead of fast rejection of table creation, which is based on the simple table and zone comparison - we will need to implement the more complicated runtime check + table creation in the atomic manner.