


# KSL-020: Kessel Inventory Resource Schema for v1beta2

## About

Summary	Defines the resource schema and persistence logic for Kessel Inventory in alignment with the v1beta2 API.
Date	May 20, 2025
Authors	Sneha Gunta
Status	WIP
Review Deadline	 Calendar event <create calendar event for reviewing ADR>
Supersedes	

*Note: As you draft your thoughts, try to be as impartial as possible in the delivery of information as to not influence reviewers, allowing for an objective analysis of information, options and decisions to consider. For reviewers, you should be ok with pointing out subjectivity as feedback.*

## What

This DDR aims to deliver a clear and stable definition of the database schema backing the v1beta2 API, along with algorithms to support the ReportResource and DeleteResource operations. The scope includes:

- Defining the relational data model for Resource and Representation entities, using names and concepts consistent with the public API to ensure traceability and developer alignment.
- Designing the ReportResource and DeleteResource logic to be robust against race conditions, particularly around read-modify-write cycles and version conflicts in concurrent environments.

- Enabling immutable, versioned storage of facts reported about resources, allowing historical data to be retained and queried over time.
- Modeling logical deletion through tombstoning rather than physical deletion, ensuring that Delete events are non-destructive and auditable.
- Supporting resource generations, allowing identifiers to be reused safely by reporters while maintaining continuity and historical separation of data.

This design provides the foundation for future evolvability, auditability, and consistency guarantees in resource reporting workflows.

## Why

We have evolved our API to espouse a sustainable architecture with the v1beta2 API. The API semantics are documented [here](#) and specific changes in terms of contract and terminology is documented [here](#).

The current table structure in Kessel Inventory is aligned with the earlier v1beta1 API and was originally designed as part of a proof-of-concept for ACM. As such, it does not meet the requirements introduced by the v1beta2 API and imposes several structural and operational limitations.

Key motivations for this redesign include:

- Terminology alignment: The database schema currently uses terms that diverge from those used in the API, requiring developers to mentally translate between the two. Aligning the schema with v1beta2 vocabulary eliminates this cognitive overhead and improves maintainability.
- Simplifying historical tracking: The existing system retains history in a separate table, requiring multiple updates per resource write – including deleting from the resource table and inserting into a resource\_history table. This introduces unnecessary complexity and multiple failure points. Versioning within a single table offers a more robust, append-only model that is easier to reason about and test.
- Non-destructive deletion: The current approach physically deletes resources, making it impossible to query previously reported data. The proposed model introduces tombstones to mark logical deletion, preserving the ability to audit or rehydrate deleted resources.
- De-scope relationship modeling: v1beta2 moves all relationship modeling into SpiceDB. This proposal removes relationship and relationship history tables from Inventory to

prevent split-brain scenarios, reducing duplication and ambiguity. Future denormalization into Inventory for read optimization can be layered on later if needed.

- Avoiding expensive migrations later: Delaying this change would require migrating all resource data stored in production. Performing this work now – before GA and before multi-reporter write volume increases – avoids complex and potentially risky data migrations in the future.

This proposal provides a forward-compatible foundation for the system, aligned with the architectural intent of v1beta2 and built to support auditability, consistency, and future extensibility.

## Solution

### Usecases

#### ReportResource

The current algorithm is documented [here](#) for reference.

#### Proposed Algorithm

#### Assumptions

- A Listener will be running in the background and will have its own connection to postgres and a subscription can be created to it, with a txId, whenever we want to
- Schema can optionally be validated before passing the data on to the application service method

1. Begin Serializable Transaction
2. Find Resource By RepresentationReferenceId, where RepresentationReferenceId is (LocalResourceId, ResourceType, ReporterInstanceId, ReporterType) and correlationIds
  - a. You should get back 2 rows if Resource and Reporter Representation exists, one for the latest common representation and one for the latest reporter representation associated with the RepresentationReferenceId (This can change after correlation is implemented. We may get back a single row for

common\_representation only or all correlated references, TBD until correlation is implemented)

- b. Note: This algorithm currently does not consider correlation, it will need to be updated when we implement correction, but it won't be post M5*
3. Validate schema to ensure that we have our aggregates in a state that is valid within the rules of the domain
  - a. This needs to be done here because there is a need to validate the Resource aggregate within the context of existing data available for that Resource.
  - b. As an example, let us consider a scenario where workspaceId is required and is defined as such in the json schema in common\_representation.json. Also, common\_representation schema is common for all reporters. But it is completely valid for some reporters to not know workspaceId (or any other fields in common\_representation for that matter) and this should not prevent them from reporting their reporter\_representations for a Resource id the Resource aggregate already satisfies the condition that workspaceId is required (In this case workspaceId was already reported by a different reporter that does have the capability to report workspaceId). This validation cannot be done based on input alone and will need to take the existing Resource state into account.
  - c. [A visual representation of this validation across layers](#)
4. Generate txId
5. If Resource Does Not Exist → Perform Create
  - a. Generate new UUID
  - b. Insert Resource (ID, resource\_type)
  - c. Insert Representation References
    - i. Reporter Representation Reference
    - ii. Common Representation Reference → reporter\_type is inventory
    - iii. Each of the representation reference rows has a foreign key to the Resource table, indicating that they both are representations of a Resource with the given ID
    - iv. tombstone is set to false by default, representation\_generation and representation\_version are set to an initial value 1. This should match whatever is the latest/highest value in the representation tables corresponding to the rows
  - d. Insert Reporter Representation
  - e. Insert Common Representation

6. If Resource Exists → Perform Update
  - a. Get commonVersion from CommonRepresentation Reference, get reporterVersion from Reporter Representation Reference
  - b. If input request contains only reporter representation, increment reporterVersion
    - i. increment reporterVersion
    - ii. Insert Reporter Representation with latest version in the Reporter Representation table
    - iii. Update version column in Representation References table for the reporter representation row
  - c. If input request contains only common representation,
    - i. increment commonVersion
    - ii. Insert Common Representation with latest version
    - iii. Update version column in Representation References table for the common representation row
  - d. If input contains both,
    - i. update both versions
    - ii. Insert Common Representation with latest version
    - iii. Insert Reporter Representation with latest version in the Reporter Representation table
    - iv. Update version column in Representation References table for the common representation row
    - v. Update version column in Representation References table for the reporter representation row
7. Write to the Outbox
  - a. Write tuples replication event
  - b. Write Resource replication event
  - c. Pass txId so
8. If read-after-write is enabled, subscribe to the Listener using txId
9. Commit Serializable Transaction and retry if it fails
10. If read-after-write is *not* enabled
  - a. Return response
11. If read-after-write is enabled and consumer notifications are on :

- a. Wait for postgres notify from consumer
- b. Return response.

## Lifecycle Diagram

### Report Resource Step-by-Step Diagram

A Q&A has also been added [here](#) for reference

## DeleteResource

1. Begin Serializable Transaction
2. Find Resource By RepresentationReferenceId, where RepresentationReferenceId is (LocalResourceId, ResourceType, ReporterInstanceId, ReporterType)
  - a. You should get back 2 rows if Resource and Reporter Representation exists, one for the latest common representation and one for the latest reporter representation associated with the RepresentationReferenceId (This can change after correlation is implemented. We may get back a single row for common\_representation only or all correlated references, TBD until correlation is implemented)
3. Commit Serializable Transaction

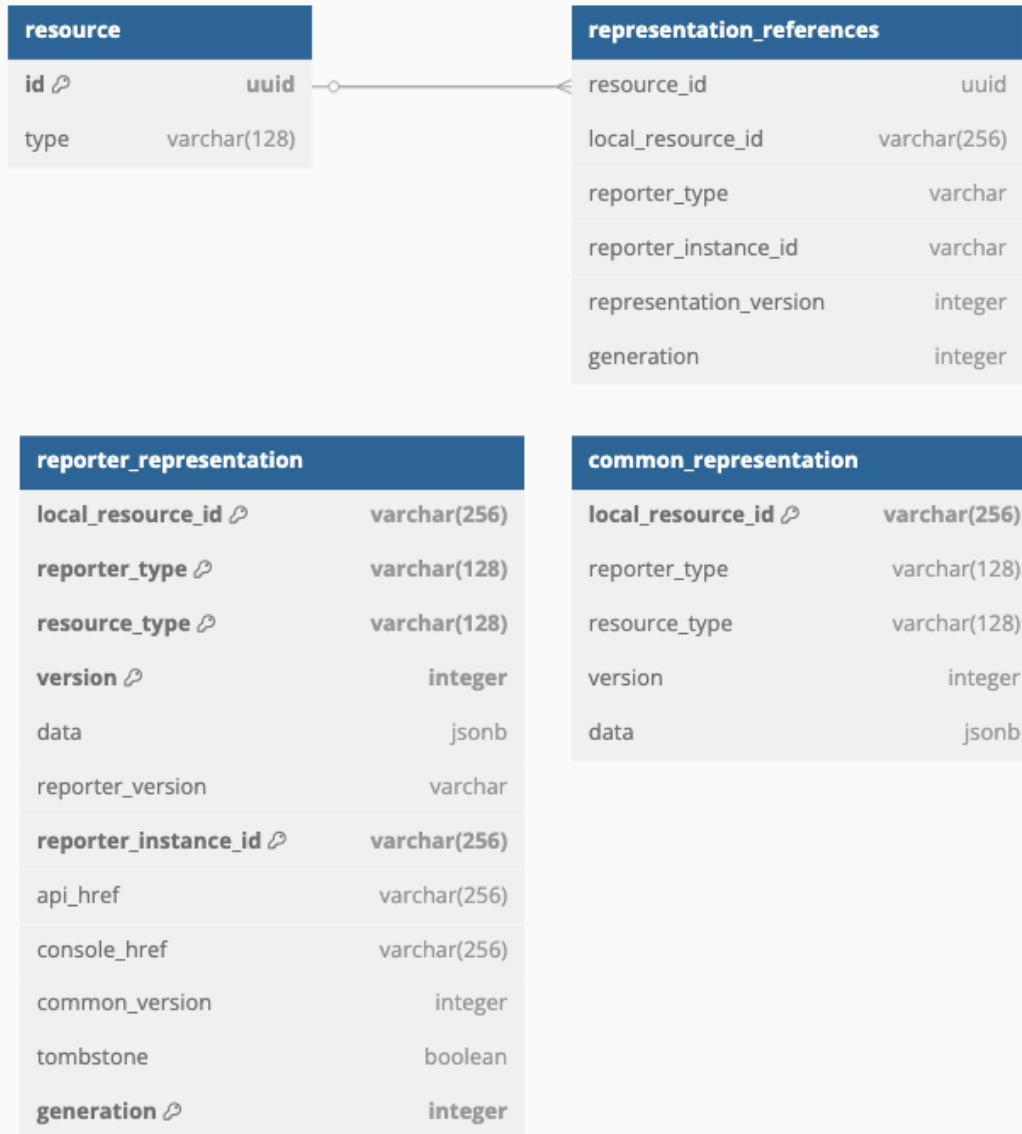
## Table Structure

In this section we will get into the specifics of how the tables will be structured, what fields each table will have and why, what are the unique identifiers for each table and what are the unique and non-unique indexes we will create for each table. We will also talk about the foreign key constraints between tables to maintain data integrity.

Below is an ER diagram showing that we will have 4 tables,

resource, representation\_references, reporter\_representation, common\_representation

## ER Diagram



**resource**

A list of all Resources, we will store an ID and type. Resources are not specific to any reporter, but represent an asset that will be reported to Kessel Inventory. A resource can have multiple reporter\_representations, each encapsulating a reporter's perspective of the Resource.

Is not an aggregate by itself. Does not store any versions.

## Fields

### *Id*

Identifies a Resource uniquely in Kessel Inventory. This is also the Primary Key for this table and is therefore already indexed.

It is of type v7 UUID and will be generated by the application code.

Can be colloquially referred to as the Inventory Resource Id or Common Representation Id or simply Resource Id.

### *type*

This represents the canonical type of the *Resource* (e.g., "k8s\_cluster", "host", "integration"). It must be a previously agreed-upon value between the *Reporter* and Kessel Inventory. It must be consistent across all *Reporter Representations* of a given Type reported by a given *Reporter*.

It is of type varchar(256).

Can be colloquially referred to as the *Inventory Resource type* or simply *Resource type*.

### *ktn*

This field will store the value of the consistency token that we get back from Spicedb when we replicate any Relationships to Spicedb via the outbox.

This will be updated by the embedded kafka consumer in Kessel Inventory.

It is of type varchar(1024) [as recommended by authzed](#).

## Other notes

There are no other unique identifiers or indexes required, in addition to the Id field.

## **representation\_references**

This table serves to denormalize the fields required for initial Resource lookup.



- It is a join table between the *resource* table and the *representation* table(s), so every row is a precomputed join between the *resource* table and either the *reporter\_representation* or *common\_representation*.
- It stores the latest version of a *reporter\_representation* or *common\_representation*, a precomputed value, so we can save the cost of calculating it on the initial Resource lookup.
- It stores other versioning data, such as tombstone and generation, denormalized from the *reporter\_representation* table, to save the cost of calculating it during the initial Resource lookup.

Each row is effectively a precomputed join falling into one of the below categories

1. Join between resource Id and fields to refer to a *reporter\_representation* uniquely (*local\_resource\_id*, *reporter\_instance\_id*, *reporter\_type*, *resource\_type*), hence a *reporter\_representation\_reference*, latest values for *representation\_version* i.e version from *reporter\_representation*, tombstone and generation.
2. Join between resource Id and fields to refer to a *common\_representation* uniquely (*local\_resource\_id*), hence a common *representation\_reference* and latest value for *representation\_version* i.e version from *common\_representation* table. Tombstone and generation are not applicable to *common\_representation*.

#### Fields

Since this is a join table, there are no fields that are specific to this table itself.

#### Indexes

##### *Unique*

To identify a row uniquely in the *reporter\_representation* table, we need to define a composite index comprising of the following fields

*resource\_id*, *reporter\_type*, *resource\_type*, *reporter\_instance\_id*, *representation\_version*, *generation*

##### *Non Unique*

The first step in the [proposed algorithm](#) requires a lookup on *representation\_references*, as part of the read-modify-write cycle to apply updates from *reporter\_representation* and *common\_representation* to an existing Resource if applicable or create a new Resource if

nothing exists. This means this lookup has to be really fast especially as the data set size increases.

## **reporter\_representation**

This table stores all versions of representations of a resource, reported by all the registered reporters for that resource as long as it is managed by at least one reporter.

All rows in reporter\_representation are versioned and immutable.

A create or an update reported by a reporter to the reporter\_representation will result in a new row in the table.

A delete reported by a reporter to a reporter\_representation will also result in inserting a new row in the reporter\_representation table, with a tombstone value false, as long it is not the last remaining Reporter to manage the Resource. If it is the last remaining Reporter, it will result in a deletion of all the Reporter Representations for the Resource, in addition to deleting all Representation References and Common Representations.

There are no usecases that we know of currently that will require mutability for an existing reporter\_representation.

## **Fields**

*local\_resource\_id*

The identifier used by the Reporter for the Resource in their namespace. The prefix local is an indicator that the resource identifier is local to the reporter namespace and is unique only within that.

Kessel Inventory does not control the sequence used for local\_resource\_id generation, so it is valid for different reporters to sometimes report the same local\_resource\_id for the same or different resource type. Hence, it cannot be considered unique in the Kessel Inventory namespace.

Can be colloquially referred to as Local Resource Id.

It is of type varchar(256).

### *reporter\_type*

This represents the type of the *Reporter* (e.g., "hbi", "acm", "acs", "notifications").

It must be a previously agreed-upon value between the \*Reporter\* and Kessel Inventory. It must be consistent across all *Reporter Representations* reported by a given *Reporter*.

Can be colloquially referred to as the Reporter Type.

It is of type varchar(128).

### *resource\_type*

This represents the canonical type of the *Resource* (e.g., "k8s\_cluster", "host", "integration"). It must be a previously agreed-upon value between the *Reporter* and Kessel Inventory. It must be consistent across all *Reporter Representations* of a given Type reported by a given *Reporter*.

Can be colloquially referred to as the Inventory Resource type or simple Resource type.

It is of type varchar(128).

### *reporter\_instance\_id*

This represents the Identifier for a specific instance of the *Reporter*. This is used to distinguish between multiple instances of the same Reporter and may not be applicable to all Reporters.

For example, in on premise scenarios, service providers may sometimes run multiple instances of a given reporter type and each instance may use the same sequence for local\_resource\_id, leading to situations where essentially 2 different resources share the local\_resource\_id, reporter\_type and resource\_type. To be able to differentiate these resources from one another and store them separately in Kessel Inventory, we will use the reporter\_instance\_id to further qualify the Resource.

In the scenarios where a Reporter does not have multiple instances, we will assign a default value to the reporter\_instance\_id. (TBD, more clarification on the details)

Can be colloquially referred to as the Inventory Reporter Instance Id.

This does not require any prior coordination with Kessel Inventory.

It is of type varchar(256).

#### *version*

This indicates the version of a reporter\_representation for a given resource. It is an Integer sequence, managed and incremented by the application and is fundamental to the concept of retaining history while implementing immutability for Reporter Representations.

Can be colloquially referred to as the Reporter Representation Version.

This field is purely for internal version management and history *within* a generation of Resources and does not mean anything to Reporters.

It is of type Integer.

#### *generation*

This indicates the generation of a reporter\_representation for a given resource. It is an Integer sequence, managed and incremented by the application.

This field is purely for internal version management and history *across* generations of Resources and does not mean anything to Reporters.

This field is different from the version field with respect to the tombstone field, described below. If a Reporter reported a \*delete\* event for a reporter\_representation, we will set the tombstone value to true and consider the Resource unmanaged by that Reporter.

But this does not prevent the Reporter from transitioning the Resource from unmanaged to managed again, and this transition happens via reusing the local\_resource\_id with all the other identifying fields i.e resource\_type, reporter\_type and reporter\_instance\_id remain the same.

In this scenario, we need to be able to store all versions of the unmanaged rows while also storing and allowing management of the resurrected Resource, while taking advantage of the unique index for data integrity. The generation field allows us to do this by further qualifying a reporter\_representation.

Can be colloquially referred to as the Reporter Representation Generation.

It is of type Integer.

*data*

This field stores all the data that a reporter wants to report to Kessel Inventory, for other reporters to integrate with.

Can be colloquially referred to as the *Reporter Representation Data*.

It is of type, jsonb.

We currently have no indexes defined on the attributes defined in this json string, since the data stored in this field does not have much meaning for Kessel Inventory, besides publishing the data in events.

Future considerations

For use cases where reporter specific relationships may need to be replicated to spicedb and also searched by, we may define some indexes and use (TBD postgres jsonb indexing)

*reporter\_version*

*api\_href*

*console\_href*

*common\_version*

This indicates the version of the common\_representation that was reported by a reporter, alongside the current reporter\_representation.

For example, if a reporter reports a reporter\_representation and common\_representation in the same API request, then this field will be populated with the value of the version from the common\_representation table.

If a reporter only reports a reporter\_representation and no common\_representation in the API, then the value for this field is null.

It is of type Integer.

*tombstone*

This is a boolean field and is used to indicate if a particular row is a tombstone for a reporter representation. This happens if a Reporter reports a delete event but the Resource is still

managed by other Reporters, we will insert a new row in the reporter\_representation table with a value = true for tombstone.

## **Indexes**

### ***Unique***

To identify a row uniquely in the reporter\_representation table, we need to define a composite index comprising of the following fields

local\_resource\_id, reporter\_type, resource\_type, reporter\_instance\_id, version, generation

### ***Non Unique***

Since we are not doing any lookups on this table, we are not going to define any other indexes for optimizing searches.

## **common\_representation**

### **Fields**

#### *id*

The identifier used by Kessel Inventory to identify a Resource uniquely. This is the ID field of type UUID, from the Resource table. This is also the primary key for the common\_representation table.

It is a v7 UUID and will be generated by the application code.

Can be colloquially referred to as the Inventory Resource Id or Common Representation Id or simply Resource Id.

#### *version*

This indicates the version of a common\_representation for a given resource. It is an Integer sequence, managed and incremented by the application and is fundamental to the concept of retaining history while implementing immutability for Common Representations.

Can be colloquially referred to as the Common Representation Version.

This field is purely for internal version management and does not mean anything to Reporters.

It is of type, Integer.

### *data*

This field stores all the common data that a reporter wants to report to Kessel Inventory, for other reporters to integrate with. This may contain some attributes that constitute relationships that will need to be replicated to Spicedb for relationship management.

Can be colloquially referred to as the \*Common Representation Data.\*

It is of type, jsonb.

We currently have no indexes defined on the attributes defined in this json string, since we do not have any usecases for doing lookups on the attributes in this json data.

We will store workspace information in this field currently and will replicate this to Spicedb.

### *reported\_by*

This will store the type of the reporter that reported the create or update event to the common\_representation for this resource type.

### **Other notes**

There are no other unique identifiers or indexes required, in addition to the Id field.

## **Notes about data types and sizes**

### **varchar**

We chose varchar(n) for all string types. [Postgres recommends](#) not using varchar(n) unless you have an application level requirement and it may be slower than the other types in some cases. But semantically it does make sense to impose some limits on size of the String values for long term manageability.

Feature	text	varchar	varchar(n)
Length constraint	✗ None	✗ None	✓ Yes (n characters max)
Padding	✗ None	✗ None	✗ None
Internal storage	varlena	varlena	varlena
TOAST support	✓ Yes	✓ Yes	✓ Yes

Performance difference	🚫 None	🚫 None	🚫 None (unless cast frequently)
Schema evolution	✅ Easy	✅ Easy	⚠️ Harder if n must change
Good for long strings	✅ Yes	✅ Yes	❌ Risk of truncation
Postgres-native preference	✅ Preferred	✅ OK	⚠️ Only if strict limits needed
Cross-DB portability	⚠️ Medium (not universal)	✅ Good	✅ Good
Enforces max length	❌ No (needs a CHECK)	❌ No (needs a CHECK)	✅ Yes (automatic validation)

## UUID v7

We talked about this in [a slack thread](#) a while ago. tl;dr These have the very nice property of being uuids (e.g. work with postgres native uuid type) but also monotonically increasing sequences which layout efficiently in b-trees

## Integer

We are currently using Integer type for version, generation and common\_version. An Integer is a 4-byte signed integer ranging from -2,147,483,648 to 2,147,483,647. In our model, we do not anticipate the number of versions for a given reporter\_representation or common\_version to go beyond that number, making this an appropriate choice for our schema.

## jsonb

We chose jsonb for storing the reporter\_representation and common\_representation for the following reasons

- More efficient option in terms of storage,
- It also provides facilities for indexing, which we will need to take advantage of in future usecases. We did some early research to test this, documented [here](#).
- It provides abilities to test containment and existence.
- It also does automatic deduplication within the stored json value, i.e the last key wins.

More details about jsonb [from official postgres docs](#).



# Performance Benchmarking

Why now and not later?

We need to migrate millions of Resources (hosts) to Kessel Inventory as part of onboarding HBI (happening in 3Q, 2025) and we need to ensure that Kessel Inventory is able to support processing multiple Resources without compromising any correctness constraints and also at a reasonable performance.

What is considered a reasonable performance?

The current goal is to be able to process 10 million hosts within 5 to 6 hours. This will allow us to remigrate repeatedly if necessary without stretching into days.

What does processing a Resource mean?

Processing includes running every reported Resource through the above algorithm and inserting to the database, within a serializable transaction. This means the numbers also include the cost of using a serializable transaction.

Does the current numbers include the outbox step or resource validation step?

No, the current numbers do not include the outbox step or the resource validation step. The current tests are only to test the performance characteristics of the various schemas, everything else being equal. Since the writes to the outbox or validation of the aggregate will have the same performance cost in both the options, it does not change the relative performance between the schemas.

We can easily expand the performance testing project to do more e2e performance testing. See [here](#) about the evolution of this project.

How is the input data set generated?

For an uncorrupted evaluation of performance, our input dataset needs to reflect realistic workloads.

To accomplish that we applied the following principles

- The implementation being tested was as close to the real production implementation as possible. [This test](#) encapsulates the actual algorithm that we will be implementing in inventory-api. [This test](#) encapsulates the alternative option. The model has all the indexes that we will use in production, for both uniqueness and query optimization. (Excluding correlation). This is important to testing performance since Indexes, while improving performance for reads, do make writes slower, hence they need to be added with careful consideration.
- We used a pareto distribution for data. The Pareto distribution is a power-law probability distribution often used to *model situations where a small number of causes lead to a large portion of the effects*—the classic “80/20 rule” is its most famous application. It is available out of the box with go and implemented [here](#). This allows us to simulate a real world example, like when we migrate all hosts from HBI into Kessel Inventory
- The input data provided is also as close as possible to the data that we would get in the real world. This means that I used real world examples of reporter\_representation and common\_representation used by hosts and acm. We also [saved the generated dataset](#) to use the same input across the candidate schemas being evaluated.
- We also used a neat modulo trick to get the distribution to cover all possible categories of input that we expect
  - hbi is the reporter, only reporter\_representation is reported
  - hbi is the reporter, only common\_representation is reported
  - hbi is reporter, both reporter\_representation and common\_representation are reported
  - acm is the reporter, only reporter\_representation is reported
  - acm is the reporter, only common\_representation is reported
  - acm is reporter, both reporter\_representation and common\_representation are reported

An additional dimension here, since some lds will be repeated, due to the pareto distribution, it will cover both create and update cases, with updates being exercised more frequently than creates.

- We ran multiple runs for the same data set, cleaning all state, dropping and recreating the DB between runs, to avoid corrupting the performance numbers.

What are the candidate schemas?

We originally considered 4 different schemas, our parameters for comparison being denormalized vs normalized references and storing the representations in 2 vs 3 tables, resulting in the following 4 options (also captured in [micro](#))

- Option1 -> denormalized reference table + 2 representation tables
- Option2 -> normalized reference table + 2 representation tables
- Option3 -> denormalized reference table + 3 representation tables
- Option4 -> normalized reference table + 3 representation tables

We quickly descope option3 and option4, since the advantage of using 3 representation tables was mainly to avoid duplicating fields between common\_representation and reporter\_representation. This can be easily achieved via modeling the schema to remove duplication in the go structs. Also, there wasn't much actually duplicated between the tables.

So we ended up comparing 2 options. Both the final schema can be found in the [kessel-benchmarking project](#)

- Option1 -> [denormalized reference table + 2 representation tables](#)
- Option2 -> [normalized reference table + 2 representation tables](#)

Show me the numbers

1 run

events/metrics	p50		p90		p99		Total time	
	Option1	Option2	Option1	Option2	Option1	Option2	Option1	Option2

1	7.020084ms	9.659834ms	7.020084ms	9.659834ms	7.020084ms	9.659834ms	7.262667ms	9.669334ms
10	619.417µs	539µs	7.522417ms	7.268375ms	7.522417ms	7.268375ms	13.995041ms	17.205834ms
100	399.5µs	423.917µs	674.084µs	645.584µs	7.692959ms	9.144666ms	55.528125ms	66.658334m
1000	227.083µs	319.708µs	410.625µs	466.667µs	937.875µs	1.28375ms	325.361459ms	370.051125m
10k	231.792µs	683.417µs	310µs	1.332416ms	522.625µs	1.752333ms	2.553289125s	7.464870125s
100k	265.708µs	330.792µs	379.834µs	1.086333ms	539.625µs	2.131084ms	26.30793975s	54.052057625s
500k	314.75µs	899.334µs	714.875µs	1.990125ms	1.409166ms	2.635875ms	2m3.233327167s	8m31.793605291s
1 million	304.834µs	1.056083ms	559.584µs	2.0275ms	1.3505ms	2.604167ms	6m18.068624375s	18m42.197465666s
10 million								

10 runs

Full results are captured in [this doc](#). I am only capturing avg total time in the below table, for easy comparison. If you want to look at other aggregated numbers like 50, 90 and 99 percentiles and see which option does better, review the linked doc.

events/metrics	Avg Total time	
	Option1	Option2
1	3.7ms	
10	9.6ms	
100	44.9ms	
1000	269.4ms	
10k	2566.4	
100k	34576.8	
500k	218682.5	
1 million		
10 million		

I want to see for myself

[This is the project](#) I used for running the tests.

Adding a README and some instructions to run the tests and changing configuration etc is TODO.

What can we do next with this?

The next step in the evolution of the benchmarking application to call the RPCs in Kessel Inventory to allow for benchmarking the complete lifecycle across multiple APIs

We could also generate a new input dataset that is more 1:1 to HBI i.e all new hosts, you just need to change the skew values [here](#) and generate a new dataset and update the input file [here](#) and run the test to get new numbers.

## Data Retention Policy

## Observability

## Technical Details/Risks/Concerns

## Alternatives Considered

Indexes for representation\_references

Numbers with using a regular Index

1 run

events/metrics	Total time
----------------	------------

1	7.029834ms
10	14.907833ms
100	59.019042ms
1000	296.053834ms
10k	2.565045459s
100k	29.00991925s
500k	3m27.538810458s
1 million	6m18.068624375s
10 million	

# References

## How validation of schema works at different layers

