UUID-First Entity Model with Type-Scoped NIDs and Fast BitSet Operations

Overview

This pattern provides a highly efficient way to manage, reference, and process large numbers of entities—where each entity is globally identified by a UUID, but also receives a type-specific, sequential numeric ID (nid) for compact storage and fast set-based queries.

You get: - UUIDs for stable, order-independent, globally unique reference. - Per-type sequential integer IDs (nids) for each entity, assigned when its type is known. - Array-based and BitSet data structures for ultra-fast querying, iteration, and set operations by type.

Workflow

Encounter Phase

- On encountering a UUID (entity reference or placeholder), add to a global registry/map:

```
Map<UUID, EntityStub>
```

- When the type of entity is discovered, assign it the next available sequence number ("nid") local to its type:

```
Map<EntityType, AtomicInteger> typeSequenceCounters
Map<EntityType, Map<UUID, Integer>> typeToUuidToNid
```

- Store the entity in a type-scoped array or indexed list (e.g., `concepts[nid]`, `descriptions[nid]`).

Reference Linking

- Cross-references between entities always use UUIDs for stability and order independence.

Fast Set queries with BitSet

- Maintain, for each type, a BitSet indexed by nid.
- Example: `BitSet activeConcepts` where the i-th bit signifies if concepts[i] is "active".
- To select/flag all type-X entities meeting a given criterion:

```
BitSet resultSet = new BitSet();
for (int i = 0; i < concepts.length; i++) {
  if (concepts[i].meetsCriterion()) {
    resultSet.set(i);
  }
}
```

- Set logic (`and`, `or`, `not`, etc.) is extremely fast and compact.

Data Structure Summary

Purpose	Structure	Pros
Global unique reference	UUID	No collisions, no ingest order dependency
Fast local (type-scoped) ref	typeNid (int)	Fast indexing and low-memory representation
Cross-type lookup	Map<UUID, EntityStub/Entity>	UUID always valid key for any lookup
Per-type storage	ArrayList<Entity>[]/Entity[][]	Optimal for sequential and indexed access
Set-based operations	BitSet per type	Ideal for queries, filters, and set math

Benefits

- **Global uniqueness:** Universally track and reference all entities with UUIDs, regardless of definition order or load sequence.
- **Efficient processing:** Arrays and BitSets make large-scale filtering/processing of entities (e.g., for LLM prompts, rule engines) extremely fast.
- **Minimal storage overhead:** BitSets and indexed arrays are vastly more compact and cache-friendly than hash sets or lists for large entity populations.
- **Type-based analysis:** Instantly handle "all concepts", "all relationships", or other entity-type–scoped logic.
- **Scalable:** Suitable for millions of entities distributed across various types.

Example Workflow (Pseudocode)

```
1. For every UUID encountered, add to a global registry (as a stub if details
lacking).
2. When entity details and type are known:
   - Assign the next available typeNid for its type.
```

```
   - Store entity at entities[type][typeNid].
3. Any reference to entities uses UUID as the key.
4. For queries (e.g., "all active concepts"), build/update a BitSet of typeNids.
5. Use BitSet operations for queries, filters, and set manipulations.
```

Applicability

This approach is ideal for: - Knowledge graphs - Terminology and ontology platforms - High-performance semantic data stores - LLM/RAG knowledge bases needing both global identity and fast, type-scoped, set-based analysis

In summary

Use UUIDs for global, stable reference. On type discovery, assign a type-specific nid for efficient storage and BitSet-based set algebra. This yields a flexible, high-performance knowledge infrastructure—combining the universality of UUIDs with the speed and compactness of sequential indexing. = Key Structure Documentation Author Name v1.0, 2024-07-24

Overview

This document describes the binary key structure used to represent components in RocksDB. Each key encodes multiple fields, allowing efficient lookup and storage of different component types and their associated information. = Managing Public IDs (UUID Arrays) as Flexible External Identifiers

Problem Statement

In some knowledge systems, an entity must be externally identifiable by one or more UUIDs—that is, it has a "public id" consisting of an array (or set) of UUIDs. Any match against any UUID in the public id is a valid match for that entity. This strategy supports: * Merging independently-authored identical entities (e.g., two people create the same concept with different UUIDs) * Legacy or alternate identifiers * Extensible identifier history

Integration Into the Efficient Entity Management Paradigm

Entity Modeling

- Each entity is assigned a `PublicId`, which is an **array or set of one or more UUIDs**.
- The `PublicId` is considered matched if **any** of its UUIDs match a query UUID.

```
class Entity {
    PublicId publicId; // encapsulates List<UUID>
```

```
    // other fields ...
}
```

Lookup and Registry

1. **UUID → nid/entity mapping**
    - Maintain a map from every known UUID to the entity's internal nid.
    - Each UUID in a PublicId array for an entity points to the **same nid**.
    - This enables O(1) resolution from any UUID to the correct entity.

```
Map<UUID, Integer> uuidToNid;          // All UUIDs known in the system map to their entity's nid
Map<Integer, Entity> nidToEntity;      // nid-based fast entity access
```

1. **Entity creation and merging**
    - When a new entity is created, assign it a fresh PublicId with one UUID.
    - To merge, **append** new UUID(s) to an entity's PublicId and update uuidToNid for each new UUID assigned.
    - On merge (e.g., two nids found to be the same entity), consolidate all UUIDs into one PublicId and update all mappings to the chosen nid.

Fast Matching (O(1) UUID Resolution)

- To find the entity for a given UUID:
    - Check `uuidToNid`; retrieve the matching nid; lookup entity in `nidToEntity`.
- To check if a PublicId matches:
    - For each UUID in the PublicId, check if `uuidToNid` maps to the desired nid/entity.

Adding New UUIDs

- Add new UUIDs to an entity's PublicId **after creation**:
    - Add to the PublicId UUID array/set for the entity.
    - Insert new mapping(s) in `uuidToNid` pointing to the same nid.
    - This allows the entity to be matched by any of its historical/alternate UUIDs.

Example Table

| Action | What Happens | |---|---| | New entity | Assigns PublicId with one UUID, maps uuidToNid, stores entity | | Query by UUID | Check uuidToNid, then retrieve entity via nidToEntity | | Add alternate UUID(s) | Update PublicId, add each to uuidToNid mapping to same entity/nid | | Merge entities | Combine PublicIds, update

all relevant uuidToNid entries, pick canonical nid |

Benefits

- **Flexible identity:** Entities are discoverable via any valid/external UUID (including post-facto merges or collaborations).
- **No order constraints:** Alternate authors, legacy imports, or identifier insensitivity is natively supported.
- **Fast lookup:** Always O(1) via uuidToNid for any UUID, regardless of how many are in a PublicId.
- **Efficient maintenance:** No need to walk all entities to match; all UUIDs directly map to the right entity.

Summary

Use a `PublicId` (UUID array or set) as the stable, external identifier for an entity. Maintain a global mapping from **every known UUID** to the entity's nid/internal id. Whenever a public id must be matched, simply look up each UUID in the uuid-to-nid map. This approach enables extensible, many-to-one, and collaborative identity without lookup or storage overhead.

RocksDB Columns

Table 1. IKE data columns

Column Name		
Key	Value	Description
UUID to Nid Column		
UUID	int	Maps a UUID to its correspinding nid. Multiple keys may be mapped to the same nid.
Stamp suffix nid to byte[] Column		
int+	byte[]	The first 32 bits of the key is the entity nid, the second 32 bits is the stamp nid for each version of the

UUID → nid (only needed for import…)
nid → [patternsequence][sequence]
[patternsequence][sequence] → component (uuids, type, referenced component, pattern) byte[]
[patternsequence][sequence][stamp sequence] → component version byte[]

Have a two pass process (using a patch/fixup column for entities that need to be backpatched):
1. Load entities into a UUID-based binary column UUID → byte[] For each insert, see if all references exist, if so, convert it immediately, and remove from the column.

All entity types are now known.

1. Convert remaining entities into a [patternsequence][sequence]→ byte[] column

PatternScopedSequence… Entities will have a pattern, and a patternScopedSequence

NOTE	This strategy will require that each import unit is referentially complete, although the referenced entity does not have to be committed. It can be in an uncommitted state.

Key Structure Format

Each key consists of the following parts, in order:

Field Name	Size (bytes)	Description
Sequential Identifier	8	Unique, monotonically increasing identifier for the component (e.g., primary key, long).
Type Byte	1	Indicates the type/category of the component. - Example values: - `0x01`: Concept - `0x02`: Semantic - `0x03`: Pattern - `0x04`: Version - (expand as needed)
Component-specified Integer	4	Additional information, whose meaning depends on the component type. For `Version` components, this is a reference to the STAMP of that version.

Field Details

Type Byte

The type byte determines the interpretation of the trailing integer and any further encoding. Possible types:

Value	Meaning	Integer/Additional Encoding
0x01	Concept	Unused/reserved
0x02	Semantic	Semantic ID
0x03	Pattern	Pattern ID
0x04	Version	STAMP ID (see below)

Version Components and STAMP

When the type byte = `0x04` (Version), the last integer field represents a STAMP, which encodes version metadata:

Field	Type	Description
Status	int	Code for the lifecycle state

Field	Type	Description
Time	long	Timestamp of the version event
Author	int	Author/User identifier
Module	int	Module or subsystem reference
Path	int	Path/context in which the version exists

NOTE The actual STAMP may be encoded elsewhere and referenced by this ID.

Example Key Encodings

Example 1: Key for a Version Component

```
+---------------------------+-----------+-----------------+
| Sequential Identifier (8b)| Type 0x04 | STAMP ID (4b)   |
+---------------------------+-----------+-----------------+
| 0x00 00 00 00 00 00 00 01 | 0x04      | 0x00 00 00 05   |
+---------------------------+-----------+-----------------+
```

Example 2: Key for a Semantic Component

```
+---------------------------+-----------+-----------------+
| Sequential Identifier (8b)| Type 0x02 | Semantic ID (4b)|
+---------------------------+-----------+-----------------+
| 0x00 00 00 00 00 00 00 0A | 0x02      | 0x00 00 00 03   |
+---------------------------+-----------+-----------------+
```

Extensibility

Future component types or additional fields can be added by allocating new values in the type byte, and/or extending the key structure with further fields.

Notes

- All integer fields are stored in big-endian order to preserve lexicographic sort order.
- Binary encoding enables fast prefix scans and range queries in RocksDB.
- Interpretation of the additional integer field **must** be consistent for each component type.