Cabinet V1 Implementation

Ikigai Cabinet is a globally-distributed, high-performance, horizontally scalable data store capable of performing a set of core operations designated as building blocks for any modern application.

The approach taken with this product is heavily inspired by the modern service-mesh platform design, where multiple self-contained layers can work in unison to build a sophisticated end-products, without a single point of failure.

This datastore implements learnings from existing classic relational databases (such as PostgreSQL), simple NoSQL document stores (MongoDB) and alternative representation data stores (Graph, Neo4J) and attempts to create a simple design which can be extended by multiple layers within the application code.

Cabinet does not intent to ever replace a traditional monolithic database and it will never attempt to achieve parity or compatibility with the functionalities provided by these systems. You can view the Cabinet as more of a highly-distributed, fast file system with additional abilities, while all the responsibilities of maintaining indexes, meta, edges and ensuring updates are always done in a single transaction will fall down to the application and associated developers.

Moving the complexity away from the database layer ensures a very simple product that is easy to maintain and poses very few constraints on the application development process. This does however slightly increase the engineering workload as they are fully responsible for the data stored: Ikigai Cabinet makes very few assumptions on what it right or wrong and does not perform high-level constraint, foreign key or schema integrity validations on its own.

Core elements:

- Nodes (document stores)
- Edges (QUAD stores for relations, similar to a Graph database)
- Indexes (views with searchable arbitrary properties)
- Meta (meta values/property list)
- Counts (distributed real-time atomical counts)
- Sequential (sequential number generation in a distributed cluster)

Composite Elements:

- Transactions (complex ReadCheck and Mutation requests processed in an ACID environment)

DB Design

Node: /n/{NODE_ID} = {protobuf}
Edge: /e/{SUBJECT_ID}/{PREDICATE}/{TARGET_ID} = {protobuf}
Index: i/n/{INDEX_ID}/{VALUE}/{NODE_ID} = null
Meta: /m/{n/e/i}/{ID}/0xY = {binaryVal}
Count: /c/{n/e/i}/CountName/{ID}/0xY/{0x0 - OxF} = atomic<INT>
Sequences: /s/{SEQ_NAME}/{SEQ_ID} = {NODE_ID}

Nodes
Applies to: None, Core data structure

Node IDs are always server-side generated - Cabinet does not accept arbitrary IDs and it is fully responsible for managing its own identification structure. As of V1, the IDs are sequential, generated using a customized version of KSUID[1]. Within a transaction, all Node IDs can be also referenced to as “iTMP:{UUID}”, which will be replaced at execution with the correct ID.

Cabinet Edge IDs are composed of the following core elements, totalling 22 bytes (176 bits), proving considerably more entropy than classic GUIDs and ensuring sequential read and writes based on creation times:

Size (22 bytes)	Type	Description
16 bits/2 bytes	uint16	Property Type ID, as provided by the Cabinet Registry
32 bits/4 bytes	uint32	UTC Timestamp since EPOCH + 14e8 (2014-03-05)
128 bits/16 bytes	int128	Cryptographically-strong pseudo-random payload

Core Properties:

- 0x0: <int> version (object revision, used for upgrades)
- 0x1: <int> date_created (mSECONDS from VALENCE_EPOCH)
- 0x2: <int> date_updated (mSECONDS from VALENCE_EPOCH)
- 0x3: map<string:bytes> direct/embedded properties

Node are always stored as very simple documents with very few properties associated. Please utilize the embedded properties in a highly selective matter, as to reduce as much as possible the memory, network and processing penalties imposed by having to load, represent and handle these objects on a regular basis). Meta properties (see below) are completely out-of-band stores designed to contain any field your application requires, with on-demand loading.

/node/{NODE_ID} = protobuf{0x0, 0x1, 0x2, 0x3}

[1] K-Sortable Globally Unique IDs: https://github.com/segmentio/ksuid

Versions are stored alongside each object and you can query these versions using the built-in “NodeByVersion” Index. Versioning allows you to simultaneously allow objects from different product generations or cycles to co-exist and to enable efficient live updates in batches, without requiring any downtime. Reading the version during runtime allows a worker to adapt the behavior based on the associated object version, simultaneously handling different versions. Applications are fully responsible for properly tagging the right node tag id.

The Embedded Meta Properties, as defined in 0x4, is a user-supplied K/V map. Due to the design of this document-based approach for Nodes, please always keep in mind to store the absolute minimum amount of information within the Node definition itself and utilize the Meta core element of the datastore for any other contents.

Edges
Applies To: Nodes

Edges are simple relations between two nodes stored in the system, stored as QUADs. This can be represented as following:

/edge/{SUBJECT}/{PREDICATE}/{TARGET} = {EMBEDDED_META}

Following this design, each relation between two object must have two edges - an inbound and an outbound. For example, assuming a GALLERY (Node, G) that has IMAGES (Node, I), would be ideally represented as follows:

Predicate 0x1: ContainsImage
Predicate 0x2: ImageIsContainedIn

/edge/{G_ID}/0x1/{I_ID} = {EMBEDDED_META}
/edge/{I_ID}/0x2/{G_ID} = {EMBEDDED_META}

For efficiency reasons, {EMBEDDED_META} must be duplicated on both legs of the edge. To ensure that performance remains optimal, please only embed minimal content and critical information and defer any other large fields to a dedicated EdgeMeta instance which can be loaded on-demand.

Queries for the vast majority of applications can be quickly answered using this approach. Given the Gallery example, this results in:

Load all Images in Gallery 122:
LIST (ALL) /edge/122/0x1/*

Check the galleries where the image 1902 is contained:
LIST (ALL) /edge/1902/0x2/*

Assuming you want to apply special operations on a data set, such as ordering the images in the gallery based on a custom POSITION int32 field, you can take advantage of the {EMBEDDED_META} to load all the ImageIDs and properties and pre-process the final list in the application logic code.

Load all Images in Gallery 122:

- LIST (ALL or PROPERTIES) /edge/122/0x1/*
- Iterate through the results and handle the logic for *.property.position

Considering each one of your Gallery Images contains a separate field, description, which is an arbitrary text field, it would be prudent to not store it alongside the {EMBEDDED_META} fields as that can seriously increase the cost of processing on multiple levels of the application. Furthermore, a description is the perfect candidate for a on-demand field, which can be stored in a separate EdgeMeta entry, such as:

SET /meta/e/{G_ID}/{I_ID} = FIELD

Note: you do not need to have an inbound and an outbound leg (unless your application demands different meta properties on the two directions) as at the time you are reading this field you are completely aware of the IDs involved. Having this consideration in mind while developing your data design can substantially reduce the storage cost and requirements by removing unnecessary duplication.

Predicate Management (Using Registry)
Predicate Management is performed in a two-step process: the application developer defines his predicates, alongside an UUID for each of them which is used by the application for all referential tracking. Upon installation of a new data schema, the “net.ikigai.cds” service will automatically create a mapping file, attributing a small, sequential ID for each associated predicate ID and provide that back to the application for embedding in the final binary.

Predicate: ContainsImage
Development Mapping: 8EA5B156-027E-4196-85A7-54590216433A
Production Mapping: 0X1 (relevant to the application data store)

Indexes
Applies to Nodes

Indexes allows fields stores in either the Embedded Meta or Meta details of an object to be used for quick classification and retrieval. Indexes apply to a single field and automatically order the lists keys and values based on internal sorting algorithms.

In the gallery example, let’s create an Index on Galleries of type=Video

Index GalleryByType: 0x1

SET /index/n/0x1/video/{G_ID} = null

To load all Video Gallery IDs, simply use:

LIST (IDS) /index/n/0x1/video/*

To load the names of the galleries, you can use a Node or Meta lookup for each of the results.

Meta
Applies to Nodes, Edges

Meta fields are very powerful out-of-band data storage solution for any non-critical fields your Node, Edge or Index contains. This allows an efficient storage mechanism for any types of content while reducing the duplication and transfer/processing overhead of always delivering this information for all requests.

Given our gallery example, let’s store a description field for gallery 122:

Meta GalleryDescription: 0x1
GET /meta/n/122/0x1 = FIELD

Meta properties have the unique advantage that they apply for multiple types of data, meaning you can attach out-of-band properties for any EDGE or NODE and efficiently retrieve them in your application code.

Node: /meta/n/{NODE_ID}/{META_ID} = FIELD
Edge: /meta/e/{SOURCE}/{PREDICATE}/{TARGET}/{META_ID} = FIELD

Counts
Applies to: Nodes, Edges

Addressing the needs of very fast, real-time counters that minimize the pressure placed on databases, Ikigai Cabinet Counts deliver a key element required by a wide array of products.

Counts are designed to be operated in a transaction for an ideal operation. Simultaneously adding or removing Edges/Nodes and incrementing/decrementing the count allows for real-time tracking that does not become stale due to inconsistent caching.

Counts are designed as:

Count GallerySubscribers: 0x1
/count/n/0x1/{G_ID}/{0..16} = atomic<INT>

Depending on the tracked type, Counts are stored as:

- Nodes: /count/n/0x1/{G_ID}/{0..16} = atomic<INT>
- Edges: /count/e/0x1/{SOURCE}/{PREDICATE}/{TARGET}/{0..16} = atomic<INT>

Each count has 16 subkeys, which are used as a load-balanced way to perform updates. When reading, the database will stream-load all 16 records and perform a simple SUM operation. When performing an update (increment or decrement), it will randomly choose one of the keys and issue a highly optimized, async atomic update.

Transactions

Transactions are a wrapper around the existing capabilities of a single mutation action, which are designed to ensure ACID compliance in a highly-distributed environment. To assist developers in performing live data and integrity validations part of the request itself, Ikigai Cabinet exposes a series of read-check operators.

Taking advantage of our underlying database design, these read-check operations can provide a critical element in ensuring proper ACID compliance and resolving conflicts automatically. In any given transaction, our data store will automatically invalidate the whole transaction and try again if any of the fields read, written or deleted have been successfully mutated by another live transaction.

This level of consistency offers a high degree of assurance that any given transaction is capable of being executed at any point, completely removing a whole array of challenges involved with traditional eventual consistency designs.

Read Check Operators

The following operators are exposed through the ReadCheck Transaction API:

Operator	Symbol	Description	Notes
EXISTS	e	Existence Check	Value is ignored
EQUAL	=	Bytes comparison	any (Value | IRI)
NOT_EQUAL	!=	Bytes comparison	any (Value | IRI)
GREATER_THAN	GT	Numeric Comparison	Numeric (Value | IRI)
GREATER_THAN_EQUAL	GTE	Numeric Comparison	Numeric (Value | IRI)
LESS_THAN	LT	Numeric Comparison	Numeric (Value | IRI)
LESS_THAN_EQUAL	LTE	Numeric Comparison	Numeric (Value | IRI)
BETWEEN	<>	Numeric Comparison of mixed pairs	Numeric Pairs Value:IRI Value:Value

			IRI:Value IRI:IRI

These operators can be applied to the result of any GET request as exposed by the associated APIs for Nodes, Edges, Indexs, Meta or Counts. IRIs are accepted as the “source” and optionally destination, offloading the reading operations as part of a single server-initiated transaction.

Given the consistency offered by our underlying database technology, this can ensure that the whole transaction would be automatically rolled back and retried should ANY of the fields previously read within the transaction boundaries change during the commit phase. This alone provides an excellent ACID compliance guarantees.

Registry
The registry is responsible for installing new versions of sequential fields used across the cabinet system. The design of the registry allows multiple developers, across multiple different versions of the platform to work independently on these fields, without requiring any complex merge conflict resolutions.

This is achieved by first requiring all sequential fields (as defined below) to be documented in development versions by a UUID, while stored in the database as sequential IDs generated upon the installation of the new revision. To offload the task of handling, processing and transmitting 128-bit integer in the place of simple 16-bit ones, the Registry allows a cross-mapping of fields, independent of the machine or environment that runs the code.

Sequential Fields are defined as:
- Node Types
- Predicates
- Index Name
- Embedded Property
- Meta Property
- Counter Name

Sequential Series
The Sequential system in Cabinet allows the generation and reference of Sequential Numbers in a global environment, which are still guaranteed to be ACID compliant across a large database.

These sequential series are however prone to high retry rates or even bottlenecks during execution in a highly distributed systems, due to the underlying design. Therefore, it is highly recommended to abstain from using Sequential Numbers for any objective that requires high-performance generation at scale.

The primary use case for Sequential Numbers within a Cabinet installation is for the Registry. Utilizing highly optimized uint16 (16 bit/2 bytes) instead of UUID references (128 bit/16 bytes) for things like Edge Predicates or Node Types results in substantial storage, processing and memory use optimizations with no costs (as references are bundled at deploy time).

Sequential series are returned as uint32 numbers, yet depending on your application needs for this sequence series, applications can (at their own discretion), cast the values as uint16 or even uint8. Given the technology design, the maximum value for counters is 2^32. Should an application expect necessary values higher than that, a different architecture might be more suitable or utilizing multiple parallel sequences within different series.

Application Matrix

	N	E	I	M	C	T	S
E	Y	N	N	N	N	N	N
I	Y	N	N	N	N	N	N
M	Y	Y	N	N	N	N	N
C	Y	Y	N	N	N	N	N
T	Y	Y	Y	Y	Y	Y	N
S	Y	N	N	N	N	N	N

Element	Applies To
Node	N/A (Core Structure)
Edge	Nodes
Index	Nodes
Meta	Nodes, Edges
Counts	Nodes, Edges
Transactions > Transact	Mutation Operations + ReadCheck
Transactions > ReadCheck	IRI accessible data (read and check only)
Sequences	Nodes

Cabinet Error Messages
During the execution of any of the RCP methods, the following defined errors can be returned:

Code	Name	Reason
	Authentication (Access Class)	
0	AuthenticationFailed	Cabinet access denied
1	AccessDeniedIRI	IRI access not granted
2	AccessDeniedMutation	Mutation on object is not allowed
3	AccessDeniedMethod	RPC method is not allowed
	General Errors (General Class)	
10	ConnectionError	Connection to data store is unavailable
11	MalformedIRI	IRI Resource provided has an invalid markup
12	GeneralError	Unspecified Error (server issue)
50	ListNoPagination	Within *_list, pagination not specified
51	IllegalUpdate	Attempt to update immutable fields
	Node Errors (Node Class)	
100	NodeNotFound	Missing Node Record
101	NodeInvalidID	Provided Node ID is not valid
102	NodeInvalidType	Provided NodeType is not valid
	Edge Errors (Edge Class)	
150	EdgeNotFound	Missing Edge Record

151	EdgeInvalidSubject	Subject ID is not valid
152	EdgeInvalidTarget	Target ID is not valid
153	EdgeInvalidPredicate	Predicate ID is not valid
	Index Errors (Index Class)	
200	IndexNotFound	Missing Index Record
201	IndexInvalidID	Index ID is not valid
202	IndexInvalidValue	Value is not valid
203	IndexInvalidNode	Node ID is not Valid
	Meta Errors (Meta Class)	
250	MetaNotFound	Missing Meta Record
251	MetaInvalidObject	Invalid Node ID or Edge
252	MetaInvalidKey	Invalid Meta Property Key
	Registry Errors (Registry Class)	
350	FieldInvalidID	Field ID is invalid
351	FieldInvalidUUID	Development Field UUID is not valid
352	FieldInvalidType	Type ID is invalid
	Counter Errors (Counter Class)	
400	CounterInvalidIncrement	NaN on increment value

	Transaction Errors (Transaction Class)	
451	TransactionInvalidAction	One or more performed actions is not possible
452	TransactionSyntaxError	Mutation Error on one or more Operations
453	ReadCheckNaN	NaN on casting one of ReadCheck elements
454	TransactionRetriesExceeded	Too many retries (as per RetryPolicy)
	Sequential (Sequential Class)	
500	SequentialInvalidType	Type format invalid