IOWN GLOBAL FORUM™

# Data Hub Functional Architecture

Classification: APPROVED REFERENCE DOCUMENT

Confidentiality: PUBLIC

Version 1.0

1/27/2022

[IDH]

# Legal

# Contents

# List of Figures

# List of Tables

# 1. Introduction

The IOWN Data Hub or IDH for short is a data management & sharing infrastructure that enables fast and trusted data processing, usage, and exchange among multiple parties or locations. The Data Hub is realized as an application's functional node on top of the IOWN GF DCI [IOWN GF DCI] architecture and is intended to be commonly used by other application nodes. It maintains compatibility with existing applications while taking advantage of the advanced nature of the DCI and Open APN [IOWN GF Open APN].

This document defines IOWN Data Hub services so that service providers can develop the Data Hub as a service, and application providers can design their applications through multiple service providers on top of Data Hub services. The IOWN Data Hub supports APIs that are functionally compatible with those of well-adopted services, such as Apache Kafka/Pulsar, AWS S3, and SQL. To achieve trusted data exchange among multiple parties while maintaining data ownership, the Data Hub provides unified secure access not only to its internal storage but also to data that each organization has kept on-premises or in the cloud. As some use cases require message brokerage services and some require database/storage services, we will discuss what service types will be needed in chapter 2. We then define several service types and classes as described in chapter 3.

# 2. Required Service Types

As mentioned above, various use cases discussing in IOWN GF have diverse requirements for Data Hub [IOWN GF AI UC] [IOWN GF CPS UC]. We first organized these requirements into several service types. A service type is defined as a unit of functionality and behavior that should be implemented together in one or several IDH services. On the other hand, it may not be sufficient to provide individual service types to realize use cases. In other words, it is necessary to design a component that inherits and implements multiple service types. Such a component is defined as an IDH class and described in chapter 3. The implementation using the IOWN DCI corresponding to each class is described in chapter 5.

Based on the discussion in the Reference Implementation Model (RIM) document [IOWN GF RIM], which analyzes use cases and builds full-stack system models, we extracted data flows relevant to Data Hub and examined what service types would be required. The data flow details will be updated based on ongoing discussions to improve the RIM. Still, we analyzed the current data flow of the RIM because it contains enough clues to determine the minimum number of service types needed.

The table below shows the data flows and their properties related to Data Hub, based on “RIM Annex C. Detailed DPD for CPS Area Management Security”.

*Table 1. IDH related Data flows from RIM for “Area Management Security” use case*

| DATA FLOW | DATA TYPE | # OF DATA SOURCES | # OF DATA CONSUMERS | FORMAT | VOLUME (PER SOURCE) | OCCURRENCE (PER SOURCE) | PERSISTENCE NEEDED | COMMUNICATION SCHEME |
|---|---|---|---|---|---|---|---|---|
| **Inference results** | Labeled data (Text data) | 1,000 x N (N: number of monitored areas) | (logically) 1 | (TBD) | 3 KB x10 labeled objects | 15 / second | Y | TBD |
| **Surveillance video** | Video data | 1,000 x N | (logically) 1 | H.264 or Motion JPEG | 0.4MB / second (H.264)<br>125KB x 15 / second (Motion JPEG) | ← | Y | TBD |
| **LiDAR data** | Sensor data | 1,000 x N | (logically) 1 | Point Cloud | FFS MB x 20 | 1 / second | Y | TBD |
| **Notification messages** | Text data | 1,000 x N | (logically) 1 | (TBD) | 256KB | 15 / second | | TBD |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **User status** | Composite data | 1,000 x N | (logically) 1 | (TBD) | 1.5KB | 1 / minute | Y | TBD |
| **Voice messages** | Audio data | N | 100 x N | (TBD) | 200KB | 1 / day | | TBD |

We considered which service type of IDH handles each data flow in the table above. This description roughly follows the concept of data flow processing for current Internet applications. Still, we assume that it will not change much in the IOWN Data Hub any time soon, considering its compatibility with existing applications. The basic idea behind this mapping is as follows:

- Individual data identifiable by some kind of structural query should be put into the Distributed Relational Database, as long as the processing speed permits.
- Relatively large size data flows, such as video and image data, are handled by the Object Storage. It typically requires more throughput (bandwidth) than latency.
- Although the size of each record is relatively small, a vast number of data records are handled by the Key-Value Store (KVS). Certain time-series data or sensor data will be mapped to this service type.
- A message broker is used in the following cases.
    - The throughput (TPS) is so large that it cannot be accommodated by any other service type and must be stored in some queue.
    - The number of data providers and consumers is vast and changes dynamically.
    - Only the relaying of data is required, not the persistence of the data.

*Table 2. Mapping of data flows to IDH service types*

| DATA FLOW | PREFERRED IDH SERVICE TYPE |
|---|---|
| **Inference results** | Distributed Relational Database |
| **Surveillance video** | Object Storage |
| **LiDAR data** | Key-Value Store or Object Storage |
| **Notification messages** | Message broker |
| **User status** | Key-Value Store |
| **Voice messages** | Message broker |

# 3. IDH Service types and classes

## 3.1. Service types

A service type is defined as a unit of functionality and behavior that will be implemented by one or several actual IDH services. To make the IDH service available for external application nodes, functional and non-functional requirements and interface requirements such as API* are described here.

* This document contains only high-level API type information. The detailed API specification will be developed through various efforts following this document.

### 3.1.1. Distributed Relational Database

A distributed relational database is a service type for storing structured data. If the data is “well-formed,” meaning the individual data can be identifiable by some kind of structural query, and if the data read/write speed is sufficient, it would be best to use this service type so that various applications can store and use the data without any extra data conversion process outside the Data Hub. This service type is intended for OLTP in general databases and data analysis use cases such as OLAP in data warehouses.

Today’s single server, HA configuration, or typical Cloud-based databases can handle 20 ~ 50 K TPS for OLTP and 5 - 10x bigger data inserts per second for IoT-type benchmark traffic. Some distributed databases can provide better performance by having more servers and distributing the data based on some algorithm; however, cross-server data queries, especially joins, do not scale linearly. IOWN GF is trying to improve this situation.

If the performance requirements of the target application fall within this range, then it can be covered by this service type. On the other hand, if the use case requirements cannot be met in terms of performance, this service type cannot be used, so it is necessary to use other service types such as KVS or Message Broker to receive the data.

#### Supposed data flows

- Labeled data of detected objects (output of AI Inference)

#### Functional requirements

- APIs
    - JDBC, ODBC, that support SQL
    - REST API that accepts SQL-based data queries and responds in JSON format
- Other key requirements
    - Data schemas need to be flexible enough to accommodate long-term data lifecycles

#### Performance and other non-functional requirements

- Scalability of data writing and reading speed when the amount of data to be stored increases
    - Especially sufficient speed for writing data when the data generation interval is very short
- Strong consistency even in a distributed DB
- Flexibility to change consistency level according to performance requirements.

## 3.1.2. Key-Value Store (KVS)

KVS is a service type for storing an enormous number of data records, e.g., trillions, quadrillions, or more, although the size of each record is not so big, e.g., less than 100 KB. For example, time-series data and sensor data would be handled by this service type.

### Supposed data flows

- Sensor data (e.g., point cloud)
- User related data

### Functional requirements:

- APIs
    - REST APIs to put and update data, specifying the key and the value, or a list of key-value sets
    - REST APIs to get and delete data, specifying a single key, or a list of keys
    - REST APIs to query data, specifying query conditions against value data
        - Note: Value field could be JSON or any other wide-column type field, that consists of child, grandchild, or descendant key-value sets. Therefore, this API has to support queries based on these sub-keys.

### Performance and other non-functional requirements

- Consideration of data location during data accesses
    - Note: Data will be distributed across multiple KVS servers. So, any data access requests need to be forwarded to the node where the target data exists. If there are multiple copies of data, then the nearest node should be selected for performance.
- Latency (between data storage and retrieval)
- Depending on the performance and size requirements, we should be able to choose a data storage from different storage classes, such as
    - In-memory KVS without persistence,
    - In-memory KVS with persistence, and
    - Disk-based KVS.

## 3.1.3. Graph Store

This service type stores data modeled as a graph. Nodes and edges that connect nodes form a graph. Information of nodes is stored as properties of those elements. The stored data forms a graph of information with links between pieces of information. Stored data might follow a pre-defined schema named ontology. The ontology describes how the data is structured and allows the application developers to formulate a semantic query. A semantic query allows retrieving information based on associative and contextual scope.

### Supposed data flow

- Sensor data

- Metadata of video/audio/text data
- Notifications messages
- User related data
- Knowledge data of the environment

**Functional requirements**

- APIs
    - REST API that accepts queries based on SPARQL, PGQL, and other domain-specific languages (DSL), and responds in JSON format
    - JDBC, ODBC that support SPARQL, PGQL, and other DSL
- Other key requirements
    - Ontologies and metamodel needs to accommodate the semantic of the specific use case and across use case: metamodel, cross-domain ontology, domain-specific ontologies
    - A domain-specific language for Digital Twin comprising means for thing query, geographic query, and historical query
    - Synchronous data retrieval: query-response
    - Asynchronous data retrieval: subscribe-notification

**Performance and other non-functional requirements**

- Scalability of the system to accommodate large graph of data
- Mechanisms to avoid retrieval of a too big subgraph resulting in query timeout, e.g., pagination

## 3.1.4. Message Broker

Message Broker is intended to be used as a message brokering service or data streaming service for any of the following scenes:

1. The application needs to be distributed to a large number of clients.
2. Data contains a mixture of sizes, types, or formats.
3. Data needs to be formatted, converted, or aggregated in a complicated pipeline.

### Supposed data flows

- Notification messages
- Voice Messages
- A mixture of notification messages and camera images

### Functional requirements:

- APIs

- General message broker API (e.g., Apache Kafka compatible API)

### Performance and other non-functional requirements

- High throughput (amount of data can be relayed)
- Low data transfer latency
- Achieve both high throughput and low latency, regardless of data size, type, or format
- Ensuring data reachability

## 3.1.5. Object Storage

Object storage is a service type to store relatively large object data, such as video, image, log, and other types of semi-structured and unstructured data, at low cost. Typical use cases are to store big data for a more extended period of time and provide data-to-data analysis applications such as AI and data lakes to prepare for future data utilization.

### Supposed data flows

- Video data
- Sensor data (e.g., point cloud)

### Functional requirements:

- APIs
  - General Object Storage API (e.g., REST API that is compatible with AWS S3)
  - Expanded APIs for various applications: classic file access like CIFS/SMB/NFS, high throughput and efficient data upload/download, specifying query conditions, etc.
- Other key requirements
  - Should be capable of bi-directional conversion between object storage and file storage.
  - It should have advanced encryption capabilities to preserve data security.

### Performance and other non-functional requirements

- Throughput for data access
- Scalability (amount of data that can be stored)
- It should be flexible enough to be configured in a trade-off manner according to performance and cost requirements.

# 3.2. Service Classes

An IDH Service Class represents a template including a typical combination of APIs and features referred to when the service provider implements IDH services. To meet different business needs, there will be various service classes to be developed. This section discusses service classes by dividing them into basic ones that inherit only one service type and applied ones that inherit multiple service types and/or are extended to cover IOWN GF unique requirements, such as federation.

### 3.2.1. Basic Service Classes

Basic Service Classes inherit only one corresponding service type. Those include the following:

- Distributed Relational Database (Distributed RDB)
- Key-Value Store
- Graph Store
- Message Broker
- Object Storage

### 3.2.2. Applied Service Classes

Applied Service Classes are defined so that it inherits multiple service types and/or integrate multiple service classes, as described below:

#### Converged DB

To satisfy various IOWN GF use cases, different types of data need to be processed simultaneously. For instance, when considering a scenario, in which people's behavior inside a building is analyzed to provide appropriate digital supports for them, the interactions of people, people-to-people, or people-to-thing must be managed and processed together. Therefore, the data structure that expresses such things requires great flexibility. If different IDH services are used for different data structures, the real-time-ness required in the IOWN GF requirements will not be achieved, especially when considering the process of reading them in sequence. A converged DB provides the foundation for high-speed, simultaneous processing of various data to meet such needs. A converged DB inherits the following service types:

- Distributed Relational Database
- Key-Value Store
- Graph Store
- Message Broker

#### Context Broker

This service class supports context-aware communication between data providers and applications. Data is requested by specifying the context scope, where context identifies the focus of the information, such as a thing (e.g., a specific building), a type of thing, geographic scope, temporal scope, or a combination of these. The typical architecture of the context broker is a federation of many context providers that have access to data, or federation of context brokers, or a hybrid combination of providers and brokers. Upon a context request, the context broker provides the context managed locally and context managed by other context brokers and context providers. Thus, this system acts as a broker that provisions data to the requester. The data provisioning might happen in two ways: 1) a synchronous request is responded with a single data response, 2) an asynchronous request ignites the establishment towards the requester of a stream of information that matches the requested context. As part of the context, a piece of information might be related to other information forming an interrelated context. A context broker inherits the following service types, with expanded APIs and features to support federation and/or aggregation of services:

- Graph Store

- Message broker
- Distributed Relational Database (Distributed RDB)

## Virtual Data Lake (Federated Object Storage)

The concept of a data lake, which was born with the advent of Hadoop, is now being transformed into a solution based on object storages in the cloud. As the amount of data increases continuously, it will change to a distributed one based on multi-cloud, edge cloud, etc. To prepare for such expected future demands, Virtual Data Lakes provide a mechanism for bundling multiple geo-distributed object storage and making them appear as one object storage. Virtual Data Lake inherits the following service type, with expanded APIs and features to support federation and/or aggregation of services:

- Object Storage

## Virtual Data Lake House

IOWN GF anticipates a future in which multiple companies and organizations exchange big data openly and securely to solve social issues. For example, to stabilize the operation of the electric power grid in which renewable energy accounts for the majority of the supply, the supply and demand situation of electric power will need to be exchanged in real-time. Virtual Data Lake House will provide solutions to such needs. It provides secure and transparent access to various data sources belonging to different owners. It also orchestrates data processing to execute it as close to the data source as possible to optimize overall data processing, by considering the placement of data and the IDH Service. Virtual Data Lake House inherits the following service types, with expanded APIs and features to support federation and/or aggregation of services:

- Distributed Relational Database
- Key-Value Store
- Graph Store
- Message Broker
- Object Storage

# 4. Today's Implementation Models and Gap Analysis

Many relevant technologies corresponding to the Data Hub classes in the previous chapter already exist in the market. However, to realize IOWN GF use cases, there are many challenges still to be overcome. In this section, typical implementation models of each technology type are reviewed, and gaps are analyzed.

## 4.1. Distributed Relational Database (Distributed RDB)

### Existing Technologies Overview

An example of a distributed Relational DB (Distributed RDB) is configured to have all-active masters on top of the shared storage. In such an implementation model, any mater node can update and reference any data. Another example is to use Read-only Replica to increase read throughput. In this configuration, the replica node cannot update data but provide read-only access. Besides these, there are also implementation models based on shared-nothing architecture. In these models, data is divided and assigned to each node or always synchronized among the nodes.

### Today's Implementation Models

Today's typical implementation models are described in the figure below. Data, data access requests, and responses to data access requests are forwarded in consideration of the distributed environment, as shown in the figure.

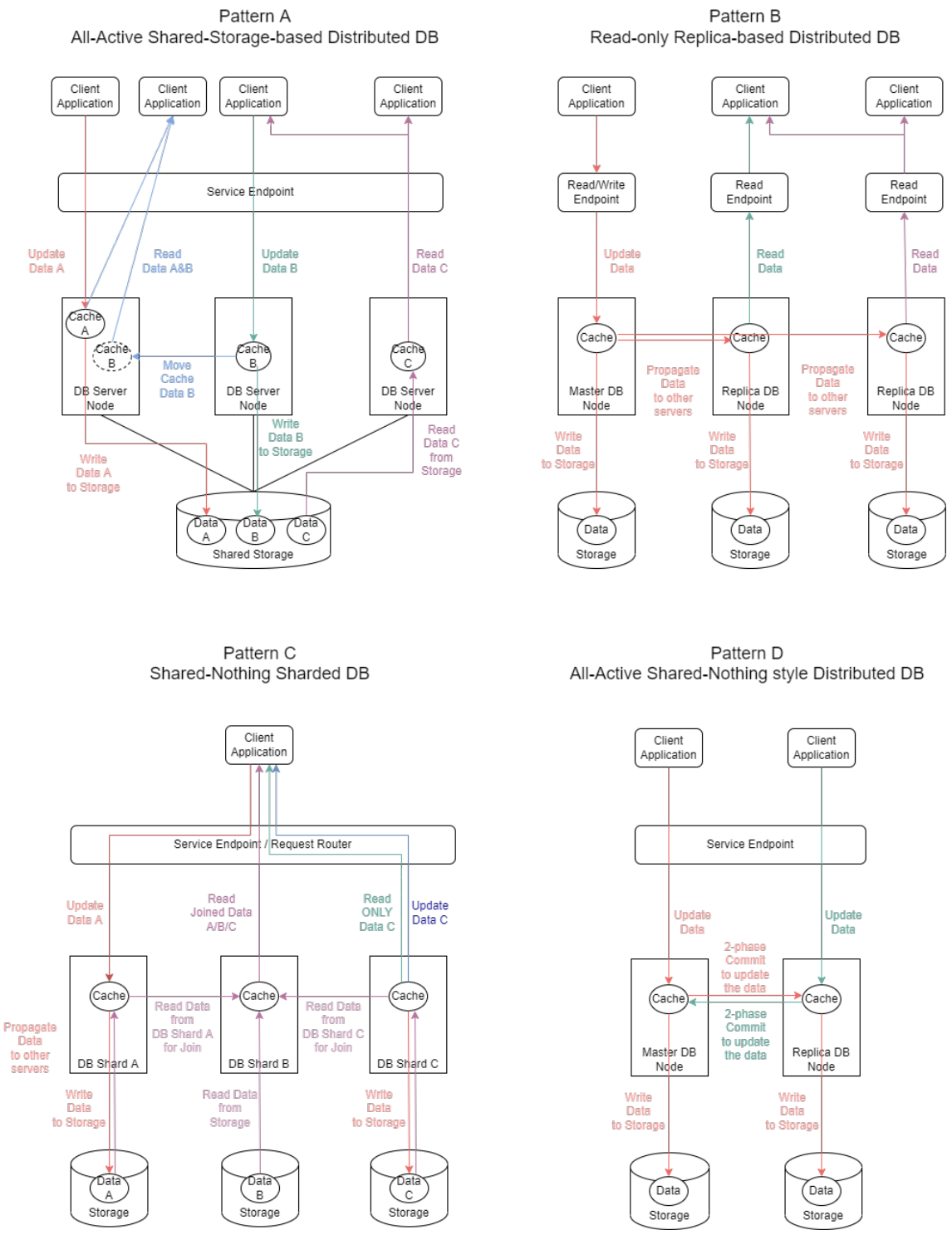



*Figure 1. Today's Implementation Models of the Distributed RDB*

## Gap Analysis for the DB Server Tier

A common consideration point across different implementation models is the existence of an Endpoint that connects the client and the Distributed RDB Service instances in the cloud. In a cloud service, a dedicated resource pool tends to be used to run the DB services for performance, and software-defined controls are embedded there to provision the

DB instance quickly for the tenant. Therefore, the roles of the DB Service endpoint are to connect the DB to the tenant's software-defined private network, block any unauthorized access to the DB, and provide load balancing and failover controls. Such a logical mechanism on top of the physical resource pool slows down the response time of the DB, which is typically more than a few milliseconds even for the single record query in typical cloud offerings.

For implementation models like All-Active Shared-Storage-based Distributed RDB, the performance will be degraded if a lot of cached data needs to be moved repeatedly. To avoid such a situation, the number of DB servers for the single application is typically 4 or 8. Thus it is usually very difficult to support as many as 1 million update transactions every second.

When assuming the digital era as described in IOWN GF use cases, the size of each data record will increase, and multiple clients will continually request access to data records. The result will be more frequent cached data movement resulting in network traffic that will continue to grow larger and larger.

To address this challenge, there will be several options in developing the new technologies. This might include new network technology that reduces network latency when connecting DB servers and storage nodes and increases the amount of bandwidth to move cached data more smoothly. Another option might be to avoid moving cached data across servers and forward the data access request, namely the SQL statement, to the server where the requested data exists. This is especially ideal when the data access is larger than the request itself or many "chatty" update transactions. By doing so, it will be possible to utilize more DB servers to scale out the performance and make DB servers span across adjacent data centers, i.e., availability zone in the public cloud case, for durability.

The master nodes that accept updates quickly become a bottleneck for implementation models like Read-only Replica-based Distributed RDB. This happens because the RDB usually is expected to guarantee data consistency, so only the master can update the data. If the data is updated in different servers in an optimistic way, it becomes difficult to ensure consistency since data update conflicts and/or wrong data updates by referencing outdated data will occur quite frequently. In addition, it is also difficult to guarantee the read consistency, e.g., the query requesting the latest data will be responded to older data by the RDB server, or dirty reads may happen as the data may be updated during the read transaction uncontrollably. To avoid such a situation, it is required to let the client ask the master node about committed transactions before it reads the data from one of the replicas and maintain the multi-version of data with a time stamp based on the most lagging local clock in the server cluster. Furthermore, if the replica server is placed in the adjacent data center as in the public cloud today, data synchronization becomes a further burden to performance. Therefore, the read performance scales very well with this implementation model to one hundred of thousands of TPS. Still, the write performance will be limited to tens of thousands of TPS, thus not enough for IOWN GF use cases.

To increase the write-operation throughput, a technique known as a sharded DB is also used. In the sharded DB, the whole data is split into several units and distributed across multiple DB servers (Shards). When updating the data, the DB shard that manages the target data is responsible for update operations. When reading the data, multiple DB Shards work together to process the query and respond the data back to the client. In this way, the write throughput can be linearly increased by adding more servers if the data update request is evenly distributed among the Shards. However, this pattern is not perfect. For instance, the data exchange between Shards will be frequent if the client requests a complex query that includes complex joins, sub-queries, etc. As a result, the query performance will be extremely slow in today's cloud environment. Also, if a server failure happens, specific data will not be accessible until the proper failover operation is conducted.

An All-Active technique can further increase the write performance and improve service continuity by eliminating any single point of failure. In this configuration pattern, data is always synchronized between 2 or several servers, and each server can conduct write operations with the support of the two-phase commit protocol. In this way, multiple CPU cores from multiple servers can work together to update the same set of data to increase write throughput. However, communications between/among servers will be very frequent in a "chatty" manner. The network will limit the communication performance, and CPU usage efficiency will decrease due to the waiting time for communications.

Lastly, it should be noted that these four patterns can be combined to solve a specific problem. For instance, Oracle RAC Sharding inherits patterns A and C, GCP Spanner inherits Pattern B and C, and Oracle TimesTen Scaleout inherits

patterns B, C, and D. In any case, the performance of these RDB implementations is significantly suppressed by the latency and bandwidth of network communications between DB servers, and between DB servers and data stores.

## Gap Analysis for the Storage Tier

Block storage is typically used as the data persistence layer for database-type workloads in today's cloud environment. To guarantee data persistence, the data is replicated locally three times across multiple storage servers. The storage servers handle the storage I/O requests sent from any clients and manage the volume and data blocks inside them while highly isolating each tenant data for security reasons. To manage the system configuration and the address of each data block, the storage server embeds the local DB, updates the information when the configuration is changed. To increase the throughput and reduce latency, the Block storage service is located in each Cloud availability zone, and sometimes the multi-tenant control plan functionality is offloaded to the Smart NIC. A typical implementation model of today's Block Storage system is shown below.

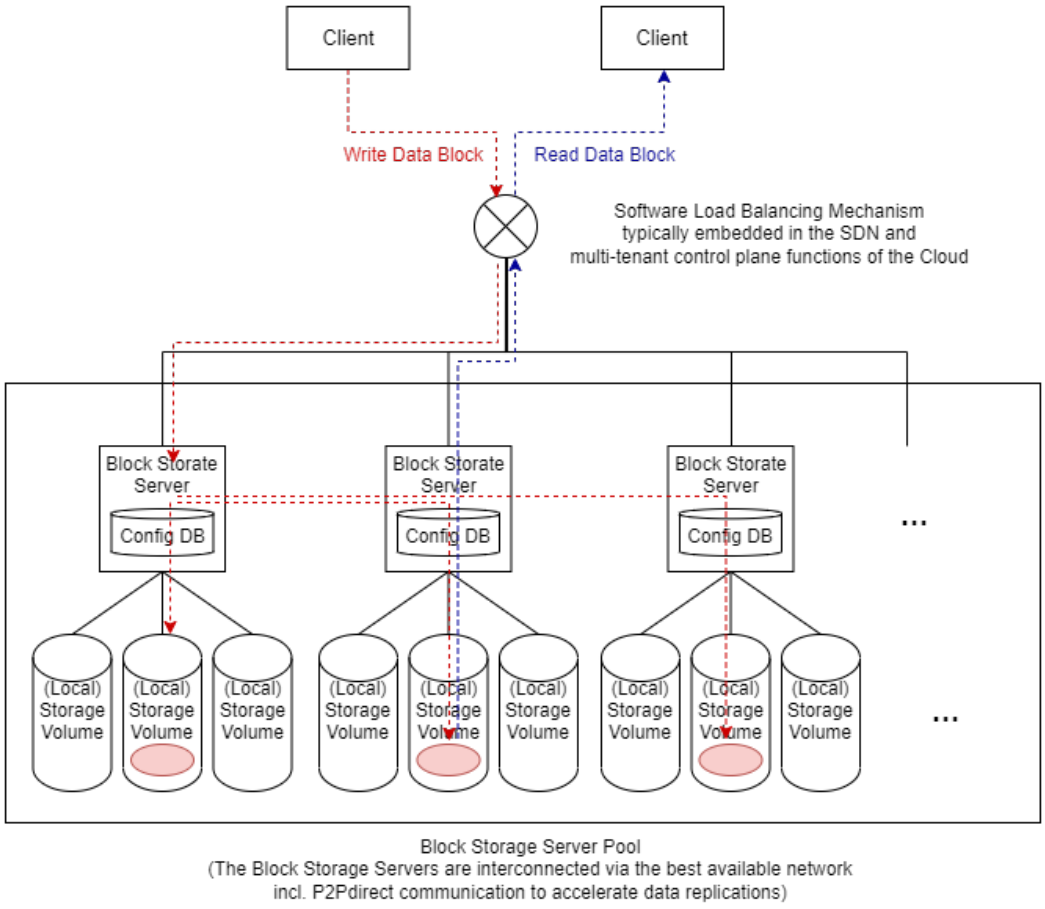



*Figure 2. Today's Implementation Model of the Block Storage*

The storage I/O capacity, access latency, and available bandwidth of the block storage are critical factors to build a high-performance database system. However, as seen in the above diagram, multiple points slow down the performance, as listed below:

- **Data Replication over limited network resources**
  To guarantee data persistence, the data is copied across multiple block storage servers. This data duplication slows down the write storage access even when the specialized resource pool is used for the Block Storage service because write operations must wait for all servers to finish writing the data to its local

storage. In addition, to avoid network congestion and resource contention, it generally applies tight throttling controls for storage access I/O and throughput.

- **The network connection between the block storage and its client (e.g., application server)**
  Due to the large resource pool in the Cloud environment, the network packet should go through multiple network switches; thus, storage access is slowed down.

- **Multi-Tenant Control**
  To build a highly scalable environment, cloud infrastructure typically adopts a mesh-style, clustered network, such as Spine-Leaf-Network architecture. This implies each time when the network packet is exchanged between two switches, the cloud control plane examines the network packet (header) and determines the physical network path to route the packet. This largely slows down the storage access latency.

Due to such restrictions, Block Storage cannot be used in the same way as Local Storage. For example, storage access latency is much higher, such as one millisecond or more (local storage could provide an order of tens of micro-second or less latency). Storage access IOPS is much lower, such as tens of thousands per 1 TB volume (1 TB PCIe SSD could provide more than hundreds of thousands of IOPS). This is the primary reason DB services in the cloud are much slower than the on-premise DB instances.

## 4.2. Key-Value Store (KVS)

### Existing Technologies Overview

There are various KVS systems already, but these can generally be described as a system with the following characteristics.

- The record is identified and accessed by the Primary Key
- Sharding Key and Distribution Algorithm controls how data is distributed across multiple servers
- Each record can hold various types of a value flexibly, e.g., a value can be JSON data, to include mode detailed data within it.

There is also a KVS system that has the following additional features:

- Secondary Indexes against the structured value, to be used for the data query, and joins
- Dynamic scale-out and scale-in to support workload fluctuations
- Sorting of data records within the server (shard), which accelerates a designated continuous/full scan of data

However, KVS is not good at the following points if compared to RDB.

- Strict (i.e., ACID properties) and/or Strong Consistency is difficult to support - some KVS supports BASE model only, and other KVS may support strong consistency within a shard but not do cross-shard consistency.
    - Note:
        - ACID means “atomicity, consistency, isolation, and durability,” a set of properties with which the RDB should be equipped.
        - BASE means “basically available, soft state, and eventual consistency.”

Strong consistency means some consistency level between ACID and BASE, e.g., providing ACID for the transactions that occur in a single shard and covers up to N records simultaneously.

- Complex joins against records stored across multiple shards is slow - this is due to 1) an inefficient secondary index mechanism, as you can imagine the creation of optimal indexes against the data of which data structure and statistical distribution are unknown in advance, and 2) overhead around parsing the "Value" field each time, and extracting data elements within it, etc.

## Today's Implementation Model

A typical implementation model of today's KVS is shown below.

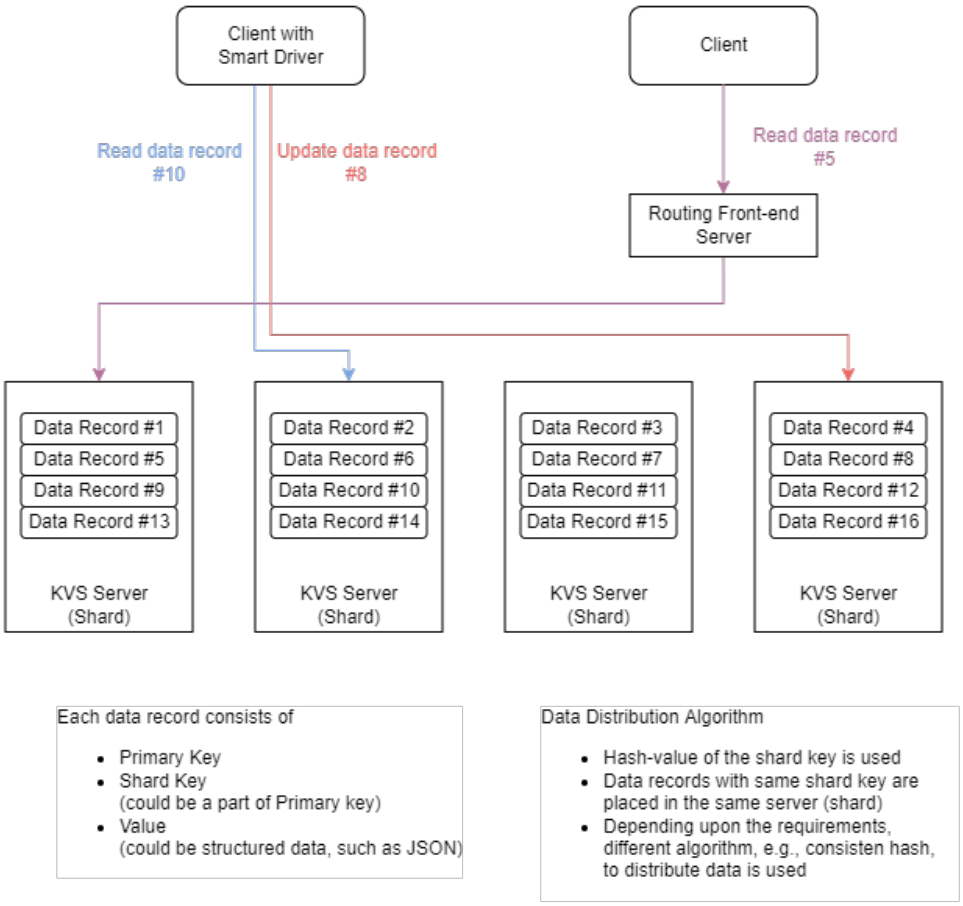



*Figure 3. Today's Implementation Models of the Key-Value Store*

For complex joins, there are several implementation models in use currently, as described below.

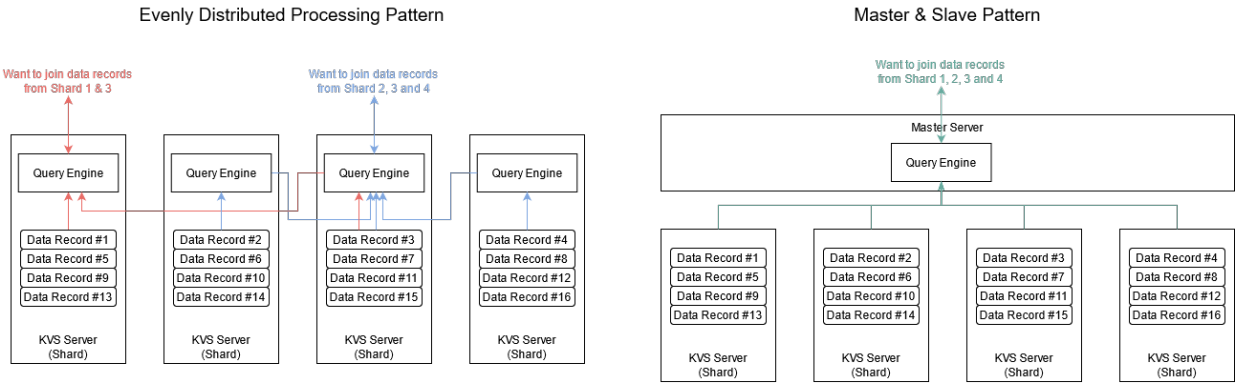



*Figure 4. Complex Query Behaviors in the Key-Value Store*

An example of data distribution algorithms and their impact on the scale-out/in operations are described below.

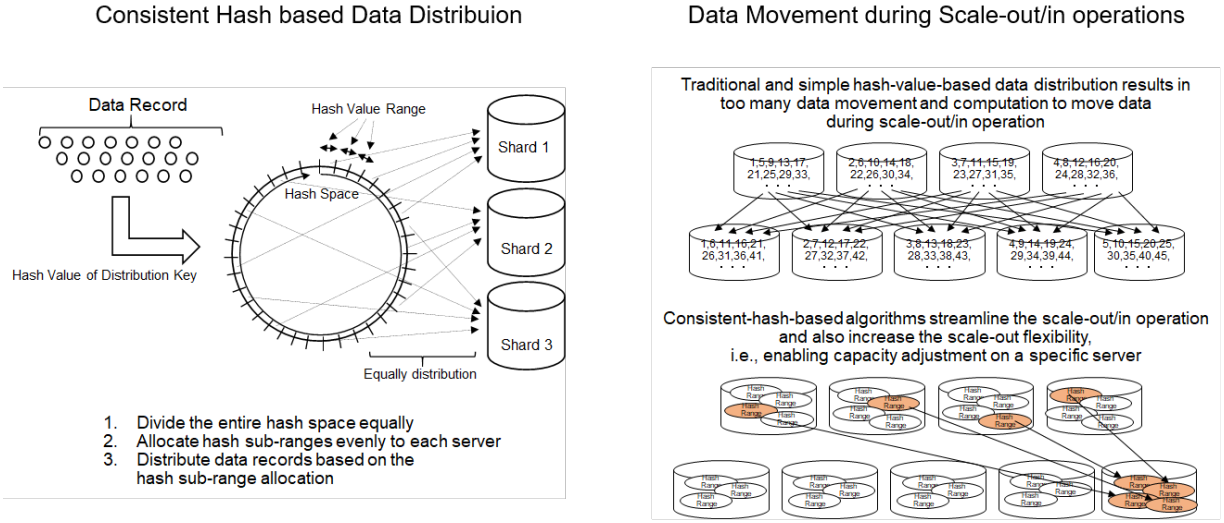



*Figure 5. Concept of the consistent hash-based data distribution*

## Gap Analysis

KVS is an excellent technology for processing a large amount of data with outstanding scalability. However, if we look at the implementation model above, we can notice several challenges.

- To decrease the response time, intelligent clients with the smart driver access a KVS Shard where the data resides in a pinpoint manner, by knowing data distribution in advance, for example, by obtaining configuration information from the metadata server to locate the data before accessing it. However, what happens when we scale out / in the KVS system? There will be a delay in delivering configuration change information to the client. Thus, a KVS that received a data access request from the client needs to forward it to the shard where the data resides.

- Too slow, cross-shard joins are problematic when considering use cases like 4D Map. Large data transfers across shards need to be accelerated and optimized no matter what deployment model is chosen. This becomes especially important when KVS clusters are built across data centers. Another issue exists around spatial and temporal data queries. The secondary indexes are well optimized for this type of query. In addition, spatial-temporal analysis is a cumbersome process compared to scalar value processing or matrix operations,

so it is necessary to have a suitable mechanism that enables efficient distributed processing on multiple servers.

- Dynamic scale-out / in operations are problematic because they involve moving data. During scale-out / in operations, online processing tends to be slow because many computing resources are used for the data movement process. Some data access requests need to be forwarded to another server, as requested data may already be moved to that server.

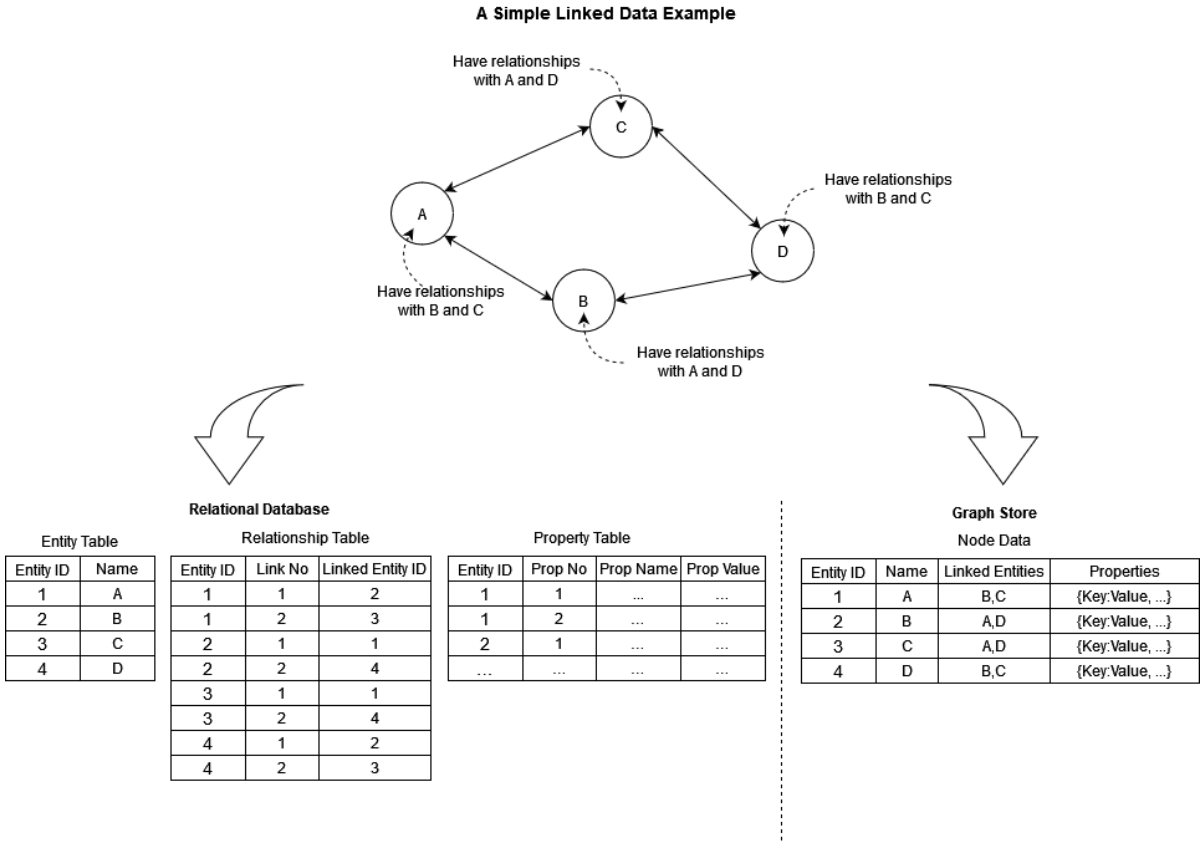
# 4.3. Graph Store

## Existing Technologies Overview

Graph store has existed for decades. In past years, its usage and development have seen significant advancement since its use for social network analysis and web analysis. This type of database stores a graph that is composed of nodes related to each by edges. Information related to a node is stored in properties attached to the node. The graph usually matches with an ontology that defines the schema and meaning of the possible nodes, edges, and properties.

## Today's implementation model

In Graph Store, linkage indexes between nodes are kept inside the node to speed up data exploration. In addition, various properties are also stored as Key-Value Pairs in nodes and edges, and indexes are added appropriately to speed up the search. If compared with relational DB, the difference is obvious. The figure below is a simple representation of the difference in data structure between Relational DB and Graph Store.




| Entity ID | Name |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |

| Entity ID | Link No | Linked Entity ID |
|---|---|---|
| 1 | 1 | 2 |
| 1 | 2 | 3 |
| 2 | 1 | 1 |
| 2 | 2 | 4 |
| 3 | 1 | 1 |
| 3 | 2 | 4 |
| 4 | 1 | 2 |
| 4 | 2 | 3 |

| Entity ID | Prop No | Prop Name | Prop Value |
|---|---|---|---|
| 1 | 1 | ... | ... |
| 1 | 2 | ... | ... |
| 2 | 1 | ... | ... |
| ... | ... | ... | ... |

| Entity ID | Name | Linked Entities | Properties |
|---|---|---|---|
| 1 | A | B,C | {Key:Value, ...} |
| 2 | B | A,D | {Key:Value, ...} |
| 3 | C | A,D | {Key:Value, ...} |
| 4 | D | B,C | {Key:Value, ...} |

*Figure 6. High-level Data Structure in the Graph Store*

To manage a large number of data records, i.e., nodes and edges, multiple servers are used, and data is distributed across them. Data placement is determined by the partition key. For better performance, the partition key is determined to collocate the most relevant records to the same server as much as possible to minimize cross-server links. If the data in the graph store needs to be updated frequently, a Graph Store cluster will be built by having two types of servers, one for data management and the other for data analysis. Such today's implementation model is described in the following diagram.

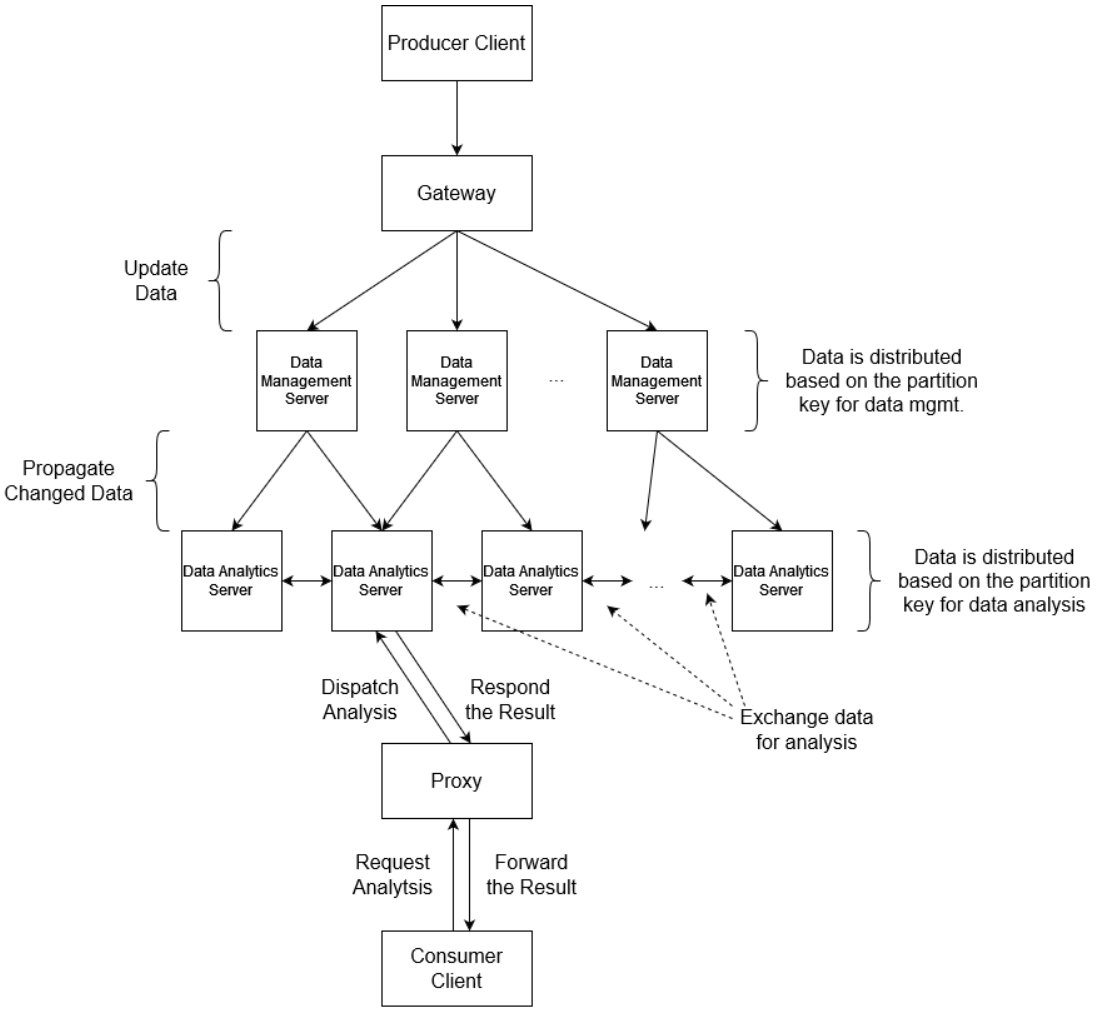



*Figure 7. Today's Implementation Models of the Graph Store*

## Gap Analysis

Data represented in a graph is well suited to represent the real world. However, Digital Twins such as in the Green Twin use case explained in section 6.2, are not only data but also analytics. The analytics process identifies current situations, predicts future situations, and recommends action from the data collected by sensors that describe raw information. For this purpose, analytics process time-series data. While properties can hold time-series information, a graph store is not well suited to efficiently handle this kind of information.

The Graph can also model geographic information, and geo-query are executed by applying filtering to the semantic query. The geo-information is treated the same way as any kind of information. Therefore, a graph store cannot differentiate it from any other type of filtering and optimize upon it, such as geo-distributed query optimization.

Another challenge is the high dynamicity of the Digital Twins data and the sensor measurements. Sensor data is pushed continuously. A graph database is not well suited for this high frequency of data updates.

Complementary to the previous issue, analytics attached to Digital Twins needs to be notified of new information matching the data of interest. A system for semantic subscription to the graph is needed. There are some ways of implementing it with nowadays graph store, such as issuing a message to a message broker for every node update. However, this kind of operation needs specific system integration.

Finally, Digital Twins are observed and represented by different domains. Taking the example of the Green Twin, the digital twin representation of a building is managed by the energy company (thru. smart meters), the local building owner (thru. installed sensors), and the event company that hosts an event in the building (thru. their monitoring system.) at the same time. This means that the graph of information is distributed between domains, but the graphs are not completely disjoint since the same node is available in many storage instances.

## 4.4. Message Broker

### Existing Technologies Overview

As a large amount of data is continuously transmitted from IoT devices, message broker technologies are well used today. The roles of such message broker technologies are ingesting data with high throughput, protecting the data from being lost, and smoothly passing it to the application that uses the data. Examples of such message broker technologies are Kafka and Cloud-based offerings. These technologies are designed to achieve higher-throughput and sequential data writing and reading while providing a unified interface for ingesting data from various devices. However, their main design targets are small payload data such as IoT data and notification messages.

### Today’s Implementation Model

A typical implementation model of today’s message broker system is shown below.

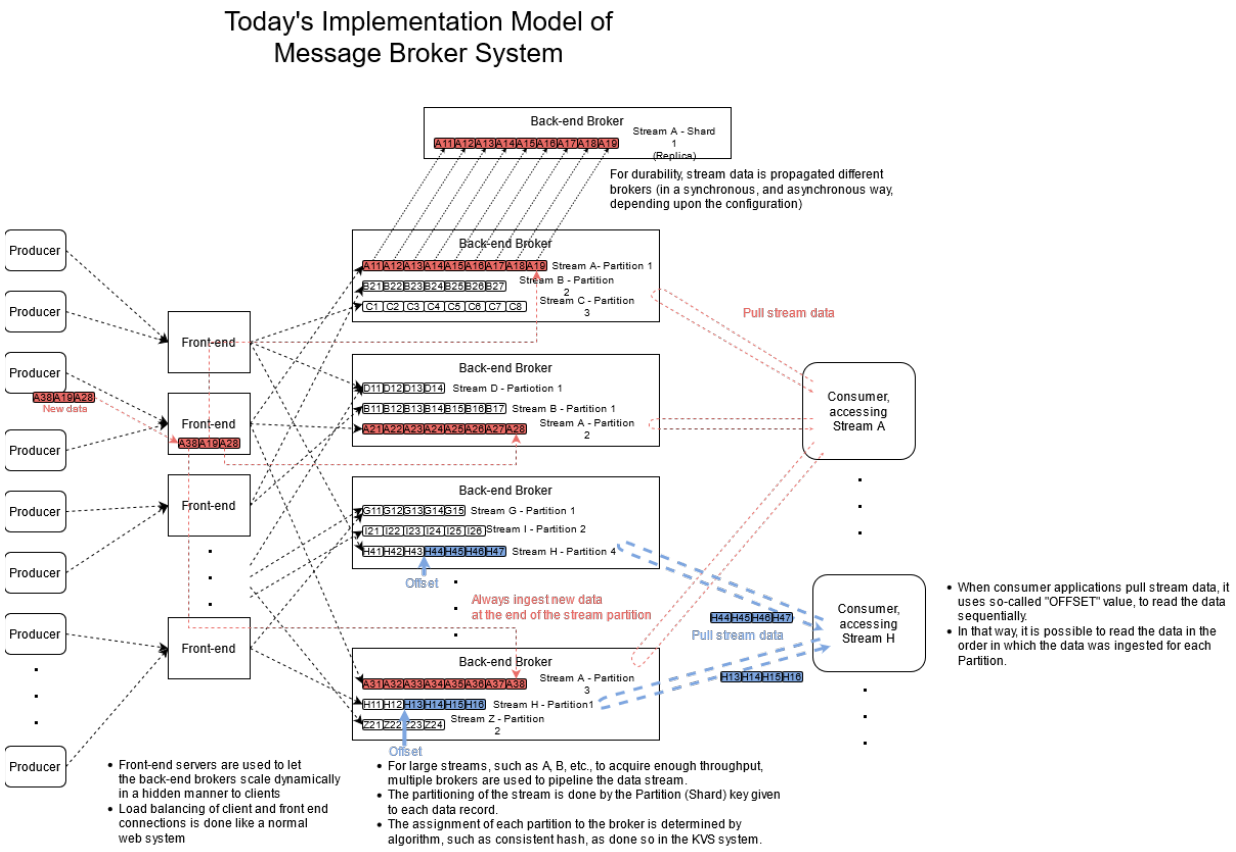



*Figure 8. Today's Implementation Models of the Message Broker*

## Gap Analysis

Today's message broker systems are well designed and can provide high throughput stably. However, there are still some issues to be solved as described below:

- In the use case of mixture data, large-size data will also be handled at the same throughput and latency as small data. However, there are few systems in the current Message broker that can efficiently handle data above a specific size. Most existing systems achieve over 100K message/s/broker when message size is smaller than 10KiB but only handle messages with a 2MiB payload at 0.1K message/s/broker.

- The time lag between when data is ingested and when data becomes available for consumers, is not small, e.g., 50 - 100 milliseconds in a Cloud-based typical configuration. This is due to propagating the data so that it is not lost. Especially in the public cloud, it should be noted that data is propagated to adjacent data centers (availability zones) as a fault tolerance measure.

- Only sequential reads are supported. It cannot be used for condition-based record extraction. For 4D maps, or some intelligent applications require condition-based data retrieval because, depending upon the situation, the records of concern are scattered apart in a data stream. To support such a conditional query, today's implementation requires another data management component. The user has to ingest data to the message broker first, then extract data from there, re-ingest data to another data management, and run the conditional query. Such a series of data processing seems to be overhead for large data streams since these don't create any business value.

- Dynamic scale-out and scale-in are slow. Today's implementation model requires moving from one broker to another during scale-out/in operations while inserting and reading data simultaneously. This is a challenging problem. Ideally, the vast memory space that pipelines the information needs to be built and shared by multiple brokers to be scaled out / in without moving the data.

- In the use case of IOWN GF, the message broker is accessible by the end-user, which requires the message broker to many subscribers. For instance, in a Mobility Service use case, millions of cars send and receive messages throughput message brokers. Today's implementation model does not focus on such a scene.

- It is impossible to handle cases where many subscribe or unsubscribe events are expected to a specific topic. For example, in a Mobility Service use case, if there are hundreds of thousands of cars in a particular area, and many of them go in and out of the area, the topic model of the message broker will be challenging to handle messages.

## 4.5. Object Storage

### Existing Technologies Overview

Object Storage is a great technology used frequently within the public cloud. This technology will not provide legacy storage access methods to clients, such as data block-based access as in the local file system and an NFS/CIFS/SMB type of file-based access as in the file server system[*1]. Such low-layer data access methods/protocols are hidden in the Object Storage, and just the REST APIs are provided to access the data. By doing so, Object Storage maximizes utilization of the storage resources and compute resources for the data access (i.e., servers to host APIs) and offers the storage service at a very low cost.

*1 By placing the so-called gateway component in front of the Object Storage service, the client can access data stored in Object Storage through file system protocols. However, only stateless APIs such as REST API is included as native APIs for object storage.

### Today's Implementation Model

A typical implementation model of today's Object Storage is shown below.

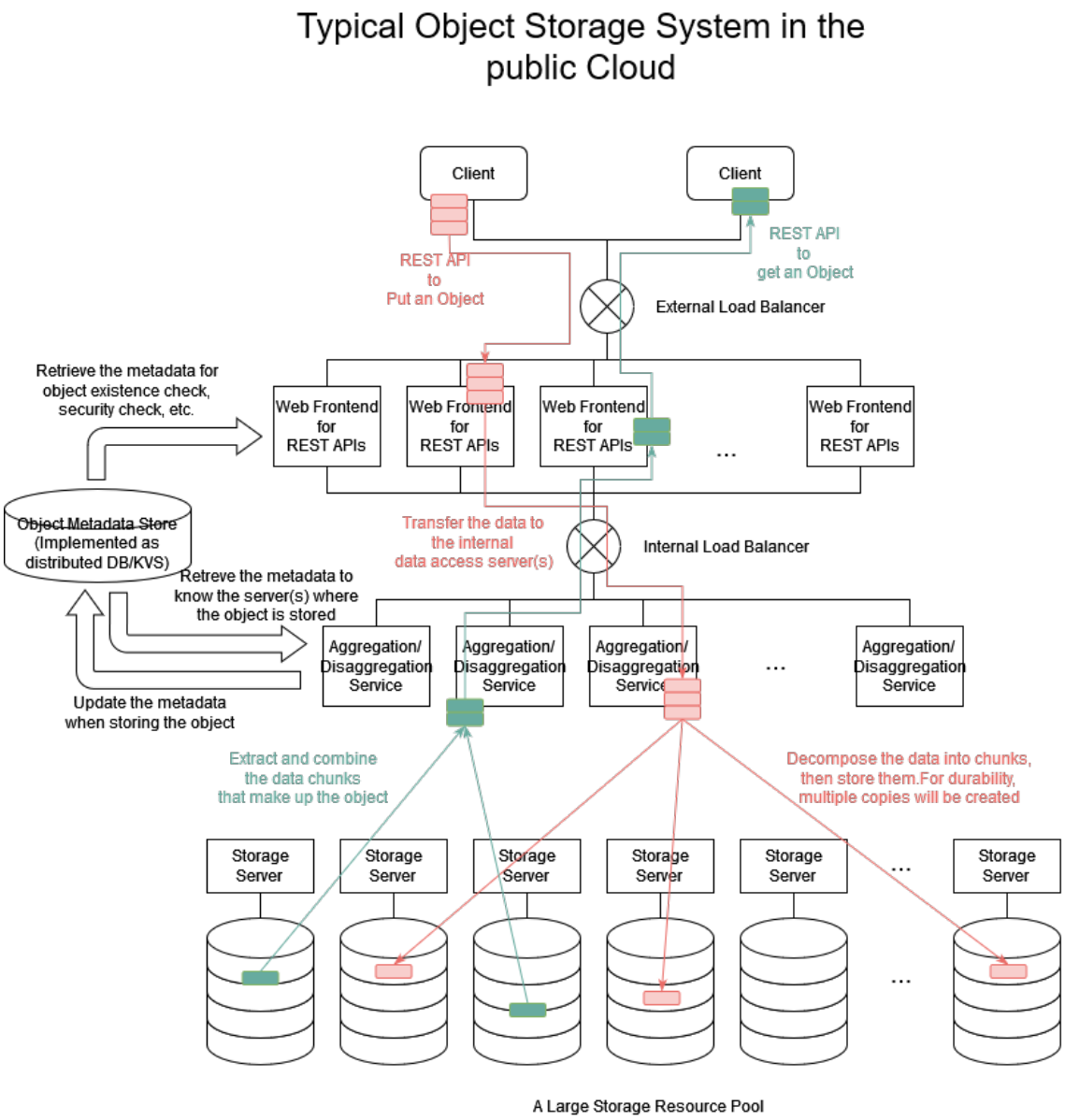



*Figure 9. Today's Implementation Models of the Object Storage*

## Gap Analysis

As shown in the above diagram, the object storage system is configured to allow multiple tenants to share the storage resources and scale significantly in the public cloud. However, as a side effect, it is challenging to guarantee stable performance. For instance, it is very difficult to guarantee the response time until the system receives the first byte of requested data and the throughput/bandwidth to continuously extract large amounts of data. As a result, Object Storage is not used for near-real-time processing today. When used as a data lake infrastructure for big data, it also impacts price/performance ratios and power consumption. Moving data to a server cluster for analysis takes time, while many CPU and GPU resources are left unused. In addition, there is a problem that when you put copies of various data into the cloud object storage to build a Data Lake. In many cases, it will end up becoming an unorganized data repository, a so-called "Data Swamp." Furthermore, in some Object Storage systems, consistency is not guaranteed. This means when the application client overwrote the Object X, another client may retrieve the older version of the Object X. The delay in data propagation causes this to make multiple copies of object data chunks in the public cloud data center, which is defaulted by the cloud vendor to prevent data loss.

The new Object Storage implementations should be studied to enable us to manage and utilize larger amounts of data more efficiently across multiple data centers.

# 5. IDH Reference Implementation Models

This section will summarize bottleneck points seen in today's implementation models and then discuss how IOWN technologies and architectures will eliminate those bottleneck points and how the IDH Reference Implementation Models that incorporate such technologies will look.

## 5.1. IDH Generic Architecture and Acceleration Methods

Figures 10 and 11 below respectively show the overall abstracted structures and common internal structures of today's IDH service implementation models, which consist of four components and three network segments, as outlined below:

**Components**

- **Client**: Request data to an IDH Service
- **Frontend Tier**: Provides load balancing, query routing, service protocol translation, message aggregation, caching
- **Data Service Tier**: Serves/updates data in response to data access requests from its local cache, if available, or the storage tier it relies on.
- **Storage Tier**: Stores data.

**Networks Segments**

- **Frontend Connection**: connects a Client and a Frontend node.
- **Mid Connection:** connects a Frontend node tier and a Data Service node.
- **Backend Connection**: connects a Data Service node and a Storage node.

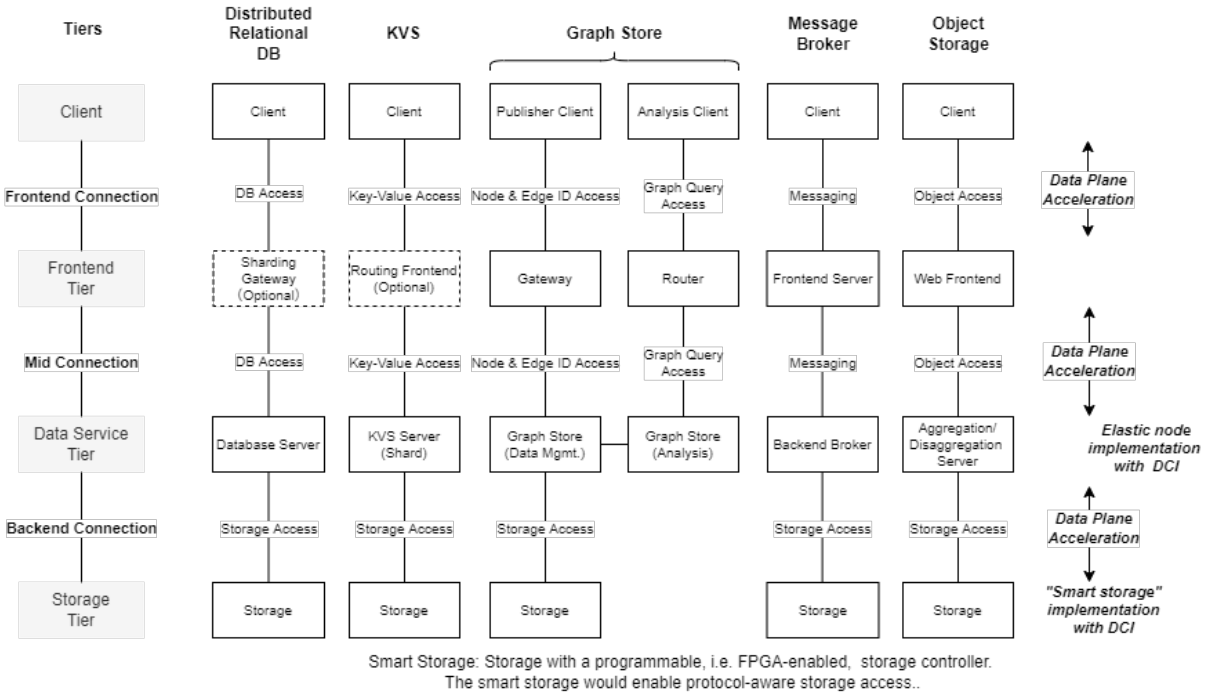



*Figure 10. Overall Structures of Today's Implementation Models of IDH Services*

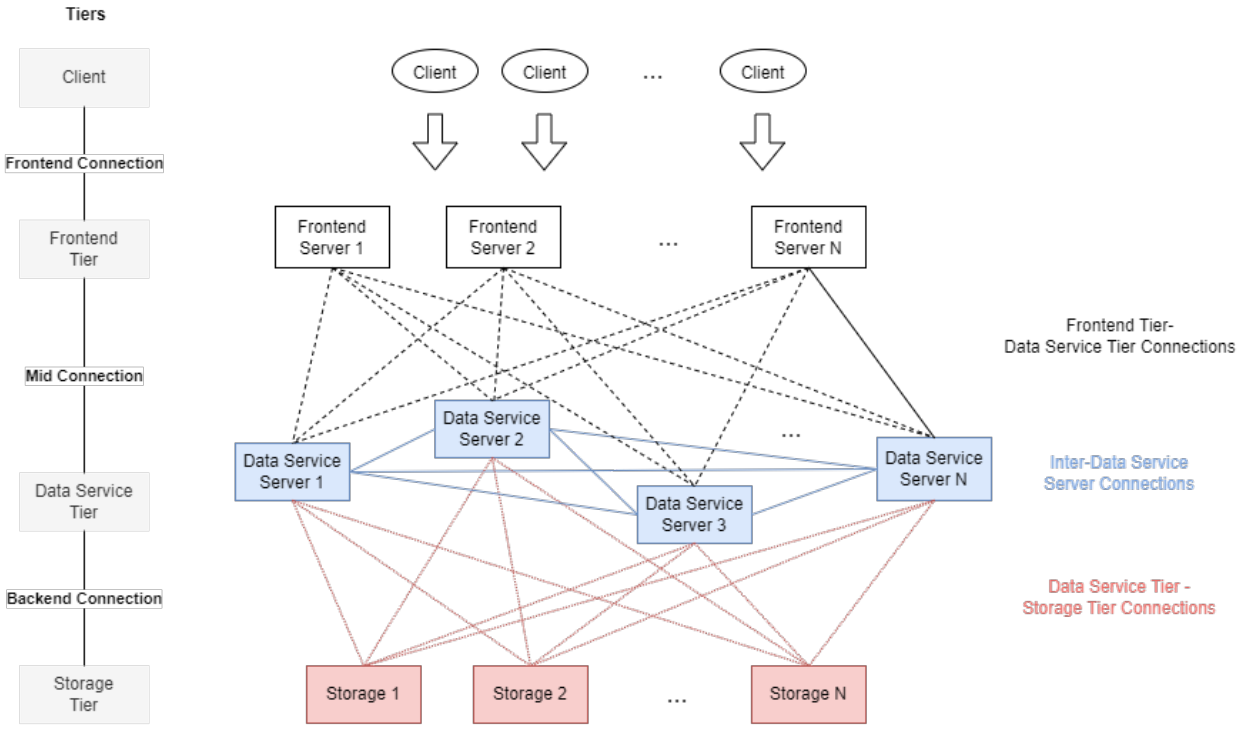



*Figure 11. Common Internal Structure of Today's Implementation Models of IDH Services*

There will be various bottleneck points of the IDH Services that arise for the future digitization services as described as gaps in chapter 4, for instance;

- When the client uploads or downloads a large amount of object data, such as a 10 GB log file, from some IDH service, Client - Frontend Tier or Frontend Tier - Data Service Tier connection becomes a bottleneck
- For complex queries that need to shuffle or join a few PB data across multiple data service servers, inter-Data Service server connection becomes a bottleneck
- For full scanning type queries that read a few EB data from the storage, Data Service Tier - Storage Tier connection becomes a bottleneck

To meet future digitization demands, these bottleneck points must be eliminated. The common solutions using IOWN technologies for this are described below.

### 5.1.1. Solution to eliminate bottlenecks around Client - Frontend - Data Service Tier connections

When IDH Services are used, the client has to connect to the data services, often through the Frontend tier, and data access requests and response data are exchanged. To achieve near-real-time processing, the network latency for these communications must be shortened, and the bandwidth must be widened, especially when a large amount of data needs to be transferred repeatedly.

Half of these objectives are achieved with Open APN and IOWN GF DCI architecture. Open APN provides a direct optical communication path between two endpoints. Thus, the latency is shortened, and the bandwidth is widened. IOWN GF DCI architecture controls a flexible and dynamic establishment of the high-speed network on top of Open APN. It governs the inter-node communication within the cluster and the cluster-to-cluster communication through the DCI Gateway.

The other half is realized by a control plane mechanism embedded in the IOWN Data Hub Technologies. As the data is distributed across multiple data service nodes, data access requests must be forwarded in consideration of data location. The IOWN Data Hub Technologies will deliver the latest data placement knowledge to the Clients and/or Frontend tier. By knowing the data placement, the Client and/or Frontend can become smart* and directly send the data access requests to the most appropriate Data Service server. This reduces extra data transfer, thus can improve latency and bandwidth.

* After this, we will call the client that becomes smart a smart client.

### 5.1.2. Solution to eliminate bottlenecks around inter-Data Service Server communications

When data processing, such as complex join, simultaneous update of multiple records, etc., is triggered frequently, large amounts of data are exchanged among Data Service servers. If the network speed is slow, it wastes CPU resources on Data Service servers. That means, especially when the blocking protocol is used, as is often the case with today's implementation models, CPU cycles will be wasted due to I/O wait. The non-blocking protocol could alleviate this issue, but it depends on a highly reliable network and has a problem of increasing complexity for distributed state management.

The Open APN helps eliminate bottlenecks in both cases because it provides a high-speed and reliable network. On top of it, the IOWN GF DCI can establish such a high-performance IDH cluster dynamically combining multiple resources. In addition, IDH Services will be designed to leverage the power of non-blocking protocols, i.e., RDMA in DCI, and to process and manage huge amounts of data in near real-time.

### 5.1.3. Solution to eliminate bottlenecks around Data Service Server - Storage Tier communications

In IDH Services such as Object Storage and Data Lake, the exchange of data between the storage tier and the data service tier is extensive. Relying on local storage to solve this problem is not desirable if considering Cloud-Native principles. That means modern system architectures should be determined to scale compute and storage separately on demand. In that sense, the storage becomes a type of remote server in the IDH Reference Implementation Models. In many of today's cloud-based implementation models, the storage is built as such a server. However, these storages are not very smart. It just responds to the requester with the requested data blocks or objects. In today's implementation models, the amount of data transferred at that time can be hundreds of terabytes with one request, and the system will be unresponsive for a long time. Therefore, today's IDH services often consume a lot of the network bandwidth for communications between the Data Service servers and the storage.

The Open APN will solve this problem due to its high-speed network services established between Data Service servers and storage. In addition, the data transfer between Data Service servers and storage will be accelerated by RDMA in the IOWN GF DCI. This is because the TCP protocol that responds to Ack each time has a high overhead to transfer a large amount of data. Using IOWN technologies, such tuning at the protocol level is also thoroughly done. Also, in IDH services, data preprocessing will be actively offloaded on the storage side using Smart NICs and the like. By allowing data to be filtered and aggregated on the storage side, data transfer across the network will be reduced by a factor of 100 or more. With these mechanisms, IDH services will run much faster in the IOWN GF Reference Implementation Models than in today's Implementation Models.

## 5.2. IOWN Reference Implementation Model for Each IDH Service Class

On top of the common solutions using IOWN technologies described in Section 5.1, this section explains additional things for each IDH service, such as unique requirements and special designs required to fulfill these. And it shows the IOWN GF Reference Implementation Models that reflect such design ingenuity, too.

### 5.2.1. Distributed Relational Database (Distributed RDB)

The basic structure of Distributed Relational DB is shown in the following figure, which consists of four layers; smart client, gateway (optional), DB server, and Storage, each of which is interconnected via a network.

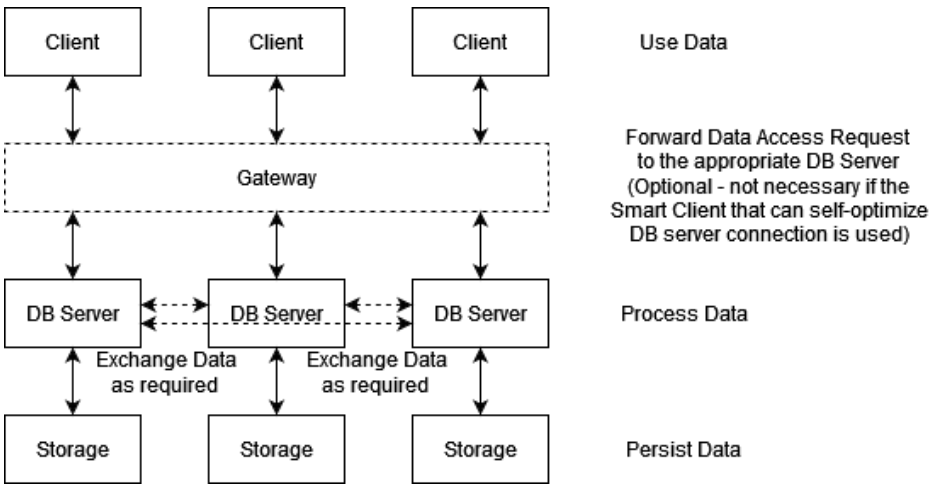



*Figure 12. Basic Structure of Distributed Relational DB*

As described in section 5.1, Open APN and IOWN GF DCI will accelerate communications between components that make up the Distributed Relational Database and data extraction from storage to improve its performance. In addition to these, some ingenuity will be added to the IDH Distributed Relational DB to accelerate its characteristic functionality, such as updating large amounts of data while maintaining consistency, high-speed data query from various perspectives, etc. Below is a list of the components and network connections that make up the IDH Distributed Relational DB, with explanations of ingenuity implemented, if any.

- **Smart Client**
  As described in section 5.1, the clients will become smart to send the data access requests to the most appropriate DB server where more request data exists in the cache. Given the IOWN GF use case, we'll find very low-latency data management requirements, such as digital twin infrastructure representing real-world situations in virtual space within tens of milliseconds or less. To achieve such low latency data processing, extra hops that require the software-defined controls in the communication path should be eliminated, which requires the client to be smart. To assist such a smart client, IDH Distributed Relational DB will deliver the data placement knowledge to all clients and ensure the information inside the clients is always up to date. The smart client also measures the latency required to communicate with each DB server to prioritize the DB servers to connect to when data is replicated to multiple DB servers.

- **Gateway**
  An endpoint capability to securely connect various clients of various tenants to the Distributed RDB on a shared environment will be embedded in the gateway component if the Distributed RDB is configured multi-tenant. With this feature, the tenant to which the access request belongs is immediately determined, and after applying appropriate security checks, connection to the appropriate DB instance on the same physical server is permitted. In today's cloud, this is implemented as software-defined controls against the tag attached to each network packet, which is one factor that causes a delay in accessing DB services. As a result, a typical DB service response time becomes more than a few milliseconds. Using IOWN technology, this delay will be reduced by leveraging the power of Open APN and IOWN GF DCI.

- **DB Server**
  In the DB server, the DB engine will become more intelligent to optimize the query optimization plan by considering the data placement, available network bandwidth, and latency for all server combinations in the cluster. It will optimize the overall data processing plan, such as which DB server to collect data from the Storages, process data on which server, transfer data processing requests or data when cross-server processing is required, etc.

- **Storage**
  Since tabular data is stored in the storage, several accelerators suitable to process it will be embedded, for example, speeding up encryption/decryption, performing column-oriented compression, creating a summary index for each data block to speed up full-scan queries, etc.

- **Smart Client - Gateway - DB Server Connection**

- **Inter-DB Server Connection**
  The DB server cluster may be configured to be geo-redundant, as is done for the high-availability configuration in today’s cloud-based implementation models. Even in such cases, the flexibility implemented in the IOWN GF DCI Gateway provides a faster network to connect DB servers between different data centers. This makes it possible to run DB Services that require high reliability faster than ever.

- **DB Server - Storage Connection**

The following figure shows images of the IDH Distributed Relational DB Reference Implementation Models with such ingenuity described above.

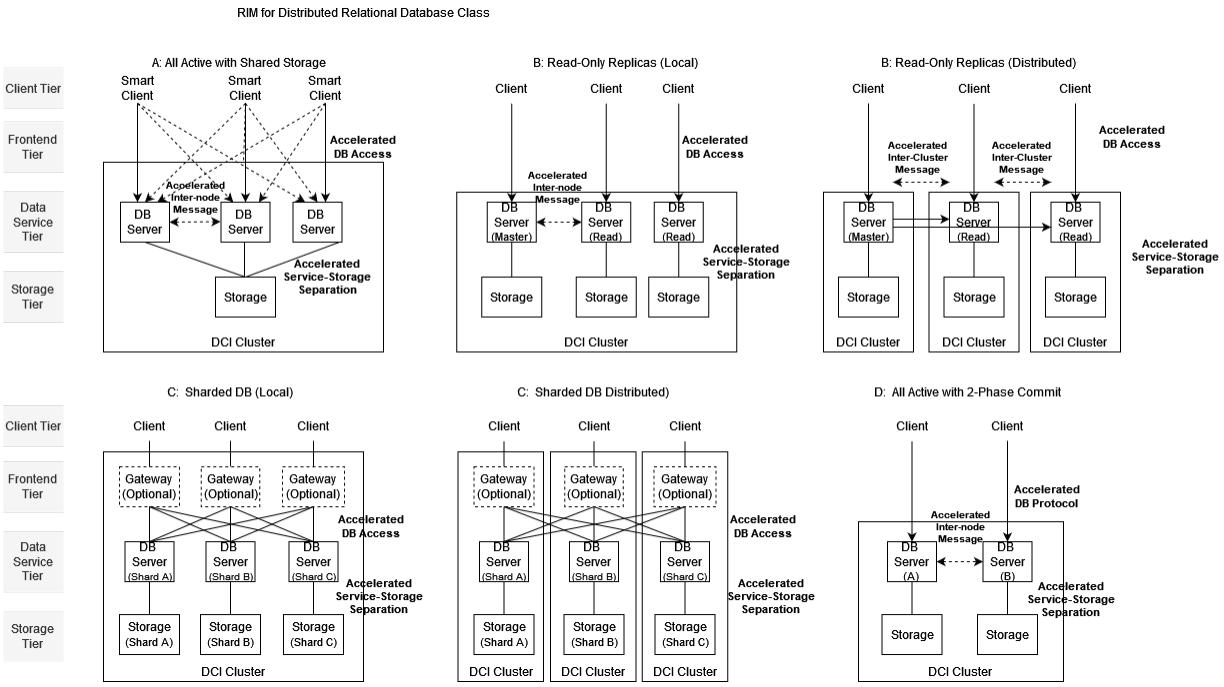



*Figure 13. Reference Implementation Models of IOWN Data Hub Distributed Relational DB*

## 5.2.2. Key-Value Store (KVS)

The basic structure of the Key-Value Store (KVS) is shown in the following diagram, which consists of four layers; Clients, Routing Frontend (optional), KVS servers, and Storages, each of which is interconnected via a network.

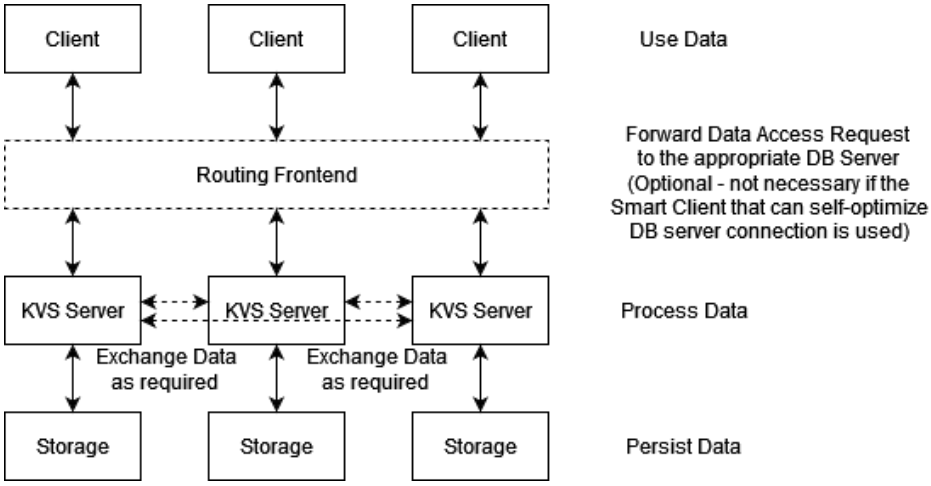



*Figure 14. Basic Structure of Key-Value Store*

KVS performance is accelerated through mechanisms described in section 5.1 while ensuring scalability of key-based simple access. On top of that, to support various data query capabilities and to increase flexibility when data distribution and/or workload distribution changes, additional ingenuity will be implemented. Below is a list of the components and network connections that make up the IDH KVS and explanations of ingenuity, if any.

- **Smart Client**
  Like the IDH Distributed RDB case, the clients will become smart to send the data access requests to the most appropriate KVS server where (most of) the request data exists in the cache.

- **Frontend**

- **Key-Value Store (KVS) Server**
  Using IOWN technology, the KVS server will be decoupled from the storage. This means improvements to the current situation in which KVS servers rely on the local storage and are coupled with Storages tightly. With such a configuration, if the distribution of data and workloads were biased, it would have been necessary to change the number of servers and relocate the data to level out the data and workload. However, such operations are difficult, even if a good algorithm such as consistent hash is used. Using IOWN technologies, it is only required to dynamically adjust the data allocation from Storage to KVS servers, depending on the amount of data and the workloads. The high-speed network using IOWN technology makes this possible by allowing remote storage to be used as if it were local storage.

- **Storage**
  In IDH KVS, the Storage becomes a storage server that provides block access to data. In such a storage server, flexible volume management is implemented so that data blocks according to the hash value range in the consistent hash algorithm can be dynamically assigned to any KVS server. Such an advanced storage server enables a more flexible, scalable, and high-performance KVS service.

- **Smart Client - Frontend Server - KVS Server Connection**

- **Inter-KVS Server Connection**

- **KVS Server - Storage Connection**

The following figure shows images of the IOWN Data Hub KVS Reference Implementation Model with such ingenuity described above.

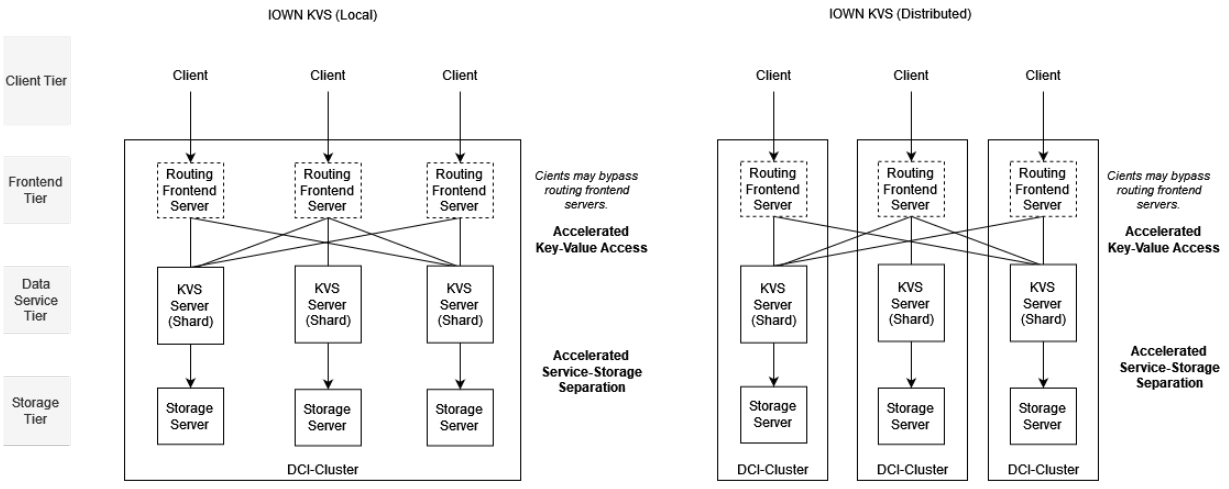



*Figure 15. Reference Implementation Models of IOWN Data Hub Key-Value Store*

## 5.2.3. Graph Store

In the architecture defined in IOWN GF, IDH Graph Store will consist of four layers; Clients, Gateway, Graph Store server, and Storages, each of which are interconnected via an ultra-fast network, too.

To develop an advanced Graph Store, we must consider two types of workloads: i) fast updates of data records, including connections, and ii) searching linked data and analyzing relationships. The workload type i) can be addressed in the same way as the Distributed RDB. However, additional considerations are needed for workload type ii). To speed up the search and analysis of graphs, it is important to introduce massively parallel processing using multiple compute resources and a mechanism for efficiently processing link information. Below is a list of the components and network connections that make up the IDH Graph Store, and explanations of new technology advances added, if any.

- **Client**

- **Gateway**
  In the IDH Graph Store, the Gateway considers the placement of data and the workload status on each Graph Store server more precisely. This is because the execution time of a graph search and analysis job can be very long. Therefore, while trying to publish jobs to servers where more data is cached, jobs are dispatched to the appropriate server set in consideration of expected CPU cycle usage.

- **Graph Store server**
  In a typical today's implementation model, two types of servers, one for data management and the other for data analytics, are used, and the graph data is replicated between them. In IOWN GF Reference Implementation Model, these two server roles are combined. This is achieved because various CPUs, accelerators, etc. can directly read and write the data stored in the server’s memory via RDMA in IOWN GF DCI. Therefore, unnecessary data replication is abolished, system resources are reduced, and processing speed is increased. It also speeds up communication between Graph Store servers, which speeds up queries that search graph data across servers.

- **Storage**

- **Smart Client - Gateway - Graph Store Server Connection**

- **Inter-Graph Store Server Connection**
  To speed up graph search and analysis by massively parallel processing with hundreds of system resources, it is important that those system resources are connected by a low-latency network. The IOWN GF DCI provides such a high-speed network, and the IDH Graph Store server cluster is configured on top of it.

- **Graph Store Server - Storage Connection**
  The unique characteristic of graph data is the link information, including hundreds, thousands, or tens of thousands of linked objects. A special encoding and a special index format will be required to manage such data effectively and efficiently. The IOWN GF DCI will provide a mechanism to offload such data processing from the Graph Store server to resources such as Smart NICs in the storage tier will accelerate the overall data process.

The following figure shows images of the IDH Graph Store Reference Implementation Model with such ingenuity described above.

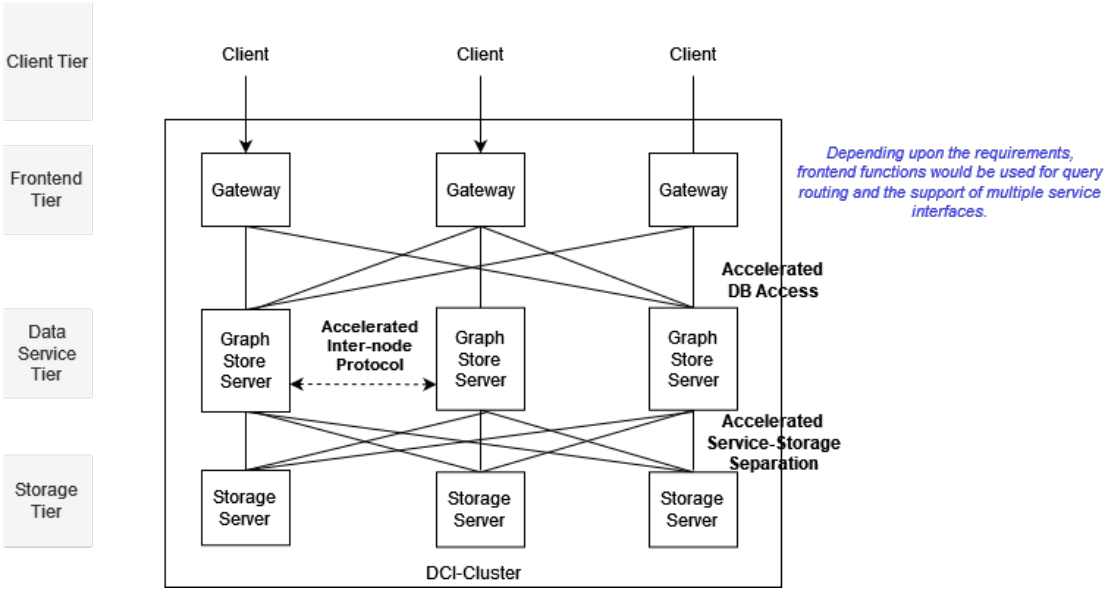



*Figure 16. Reference Implementation Models of IOWN Data Hub Graph Store*

## 5.2.4. Message Broker

In a message broker system, data of various sizes, types, and formats are ingested from various devices and passed to the subsequent consumer applications in which data is processed in different ways. To support not only sequential and delayed queuing, but also real-time event integration and query-based flexible data extraction, the message broker is comprised of two types of components, “Data Broker” and ”Event Broker”, in the IDH Reference Implementation Model. “Data Broker” is designed to queue data that is too large for immediate processing in other systems, so that data can be processed sequentially or in a batch manner. “Event Broker” is designed to pipeline small messages as well as the metadata of large messages, so that processes at the subsequent applications can be triggered in an event-driven manner with low latency. It also supports queries to know which data is ingested and kept in the message broker.

Therefore, the IDH Message Broker can be characterized as:

1. The use case of mixture data: It can handle both small size data and extensive size data.
2. High throughput: It achieves high throughput regardless of the size, type, and format of data.
3. Low latency: It notifies data generation to clients asynchronously to avoid the bottleneck of data transmission.

Below is a list of the components and network connections that make up the IDH Message Broker and explanations of ingenuity implemented.

- **Client**
  The client library to utilize features of IDH Message Broker is installed in the client. The client library provides Kafka-feel APIs to interact with Data Broker and Event Broker, including:
  - Topic management such as Create, Delete, and List topics.
  - Partition management such as Create, Delete, and List partitions.
  - Publish messages to the Message Broker
  - Subscribe the topics/partitions so that message arrival is notified immediately
  - Queries message metadata to get an overview of what data is being ingested

- Retrieve messages sequentially or conditionally from the Message Broker

One of the main distinctions between this reference implementation model and existing products is that the Client Library automatically divides and transfers messages to Data Broker and Event Broker for the lowest latency and the highest throughput.

- **Frontend**
  The Frontend is an optional component for high availability and workload distribution. It provides a unified endpoint for the end-user and transfers messages to one or more brokers by considering message size. Without having it, the high availability requirement is not well supported as all messages will be sent to the Data Broker. Since the Event Broker will be populated from the Data Broker, latency is increased.

- **Data Broker and High Throughput Storage**
  Data Broker and high throughput storage are designed to receive and persistent large-scale data streams at very high throughput. To achieve this, any incoming stream can be treated as a byte stream to achieve the best performance. If configured, it also ingests the metadata of received data to the Event Broker.

- **Event Broker and Real-time Storage**
  Event Broker and real-time storage are responsible for routing and storing small messages as a single object at low latency. To minimize the latency, rapid storage such as persistent memory is used to store data. If configured, it notifies subscribed clients about message arrival in an event-driven manner to enable real-time processing. It also provides query capability to enable not only sequential reads but also conditional reads. In existing message brokers, each stream is composed of resources with a fixed throughput, so one stream often contains multiple sensor data. Since only pull-type sequential reads were supported, there are issues with real-time delivery and processing efficiency. The Event Broker will eliminate these issues.

The following figure shows images of the IDH Message Broker Reference Implementation Model with such ingenuity described above.

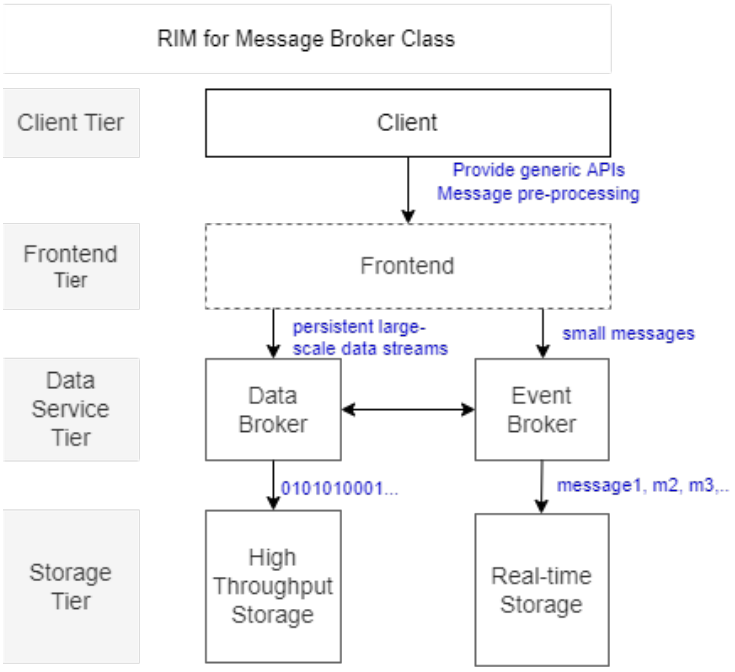



*Figure 17. Reference Implementation Models of IOWN Data Hub Message Broker*

## 5.2.5. Object Storage

The basic structure of Object Storage is shown in the following diagram, which consists of four layers; clients, web frontend servers, data service servers, and storage, each of which is interconnected via a network.

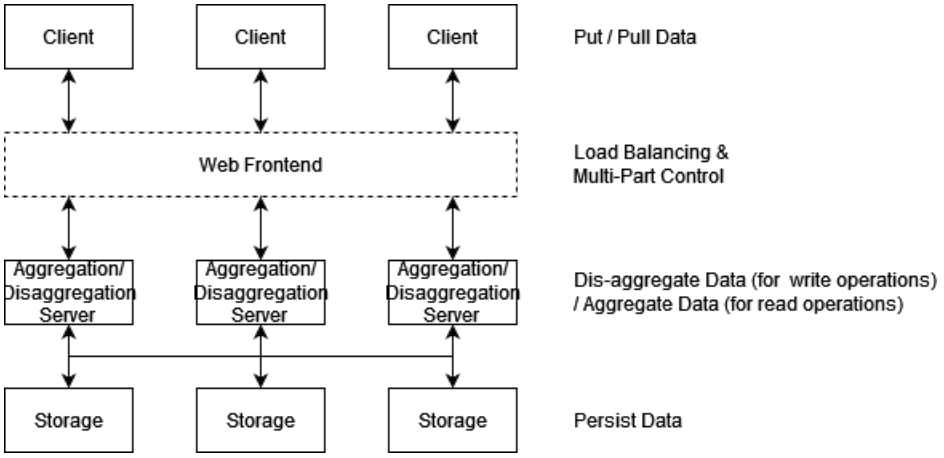



*Figure 18. Basic Structure of Object Storage*

With the explosion of data volume, Object Storage has become the core of the data lake. Using IOWN technologies, some ingenuity will be added to improve the performance of Object Storage. Below is a list of the components and network connections that make up the IDH Object Storage and explanations of ingenuity implemented, if any.

- **Client**
  On the client-side, only Put and Get REST APIs are used to access the data stored in Object Storage in today's implementation model. In the model discussed in IOWN GF, these access methods are extended by the client library. For instance, a client library that can:
    - put and get objects through REST APIs, including AWS S3 compatible ones
    - accommodate other protocols such as CIFS/SMB,
    - efficiently upload and/or download data based on UDP and/or RDMA protocols, as well as multi-part access controls, on top of Open APN
    - designate query conditions to retrieve only the data chunks that the client needs, such as blocks in the parquet file and/or video segments in the video file, etc.

  These new features are made possible by improving the Object Storage implementation model as described in this section.

- **Web Frontend**
  The roles of Web Frontend are 1) to interact with Clients, including authenticate and authorization, confirmation of the existence of the requested object, enabling access to the object data for its clients by providing a set of general APIs compatible with current cloud services and existing on-premise storage systems, coordinating multipart upload and download, etc., 2) managing metadata of the object and also to provide internal load balancing controls for performance and availability. In today's implementation model for object storage, the ratio of computing resources to storage resources is fixed. Therefore, the Web Frontend only provides very limited APIs, such as Put and Get. In the model discussed in IOWN GF, the resource pool can be configured more flexibly and dynamically. Therefore, other types of functionalities can be added to the Web Frontend, such as the file system protocols to streamline the data access in the object storage without requiring a gateway server needed in today's implementation model.

- **Aggregation / Dis-aggregation Server**
  The roles of the Aggregation / Dis-aggregation server are 1) to dis-aggregate the received data into multiple blocks before storing them when the client uploads an object to the Object Storage, and 2) aggregate the several blocks that make up the object to pass it to the client when the client downloads an object from the Object Storage. In today's implementation model, splitting this data into parts is simply based on data size. In the model discussed in IOWN GF, splitting into parts will become more intelligent by considering the content. For instance, the parts for the parquet file will be determined by the parquet's internal blocks. The parts for the video file will be determined by the video file's internal segmentation, such as a chapter or key-frame-based segmentation. In addition, index information for these chunks of data will be created, such as minimum-maximum values for each column stored in the parquet block and collection of subtitle texts, start and end timestamps, etc., for each video segment. By having such a structure and allowing the client to specify the data retrieval conditions, only the portion of the data that the client needs can be extracted from the object storage efficiently and at high speed. This resolves inefficiencies we frequently see in object storage usage today. To retrieve only the small portion within the object, the entire object needs to be retrieved from the object storage, which is time-consuming and costly for its users.

- **Storage Tier**
  The role of the storage tier is to persist the object chunks created above and internal index information securely. To guarantee data persistence, data is replicated three times among different servers typically located some distance apart. IOWN GF DCI technologies for the inter-cluster network will accelerate this data replication. In addition, to protect the data-at-rest from any unauthorized access, the data will be encrypted. IOWN GF DCI will offload such tasks from the CPU to the storage, e.g., Smart NIC installed there, and encrypt the data with a customer-specified encryption key.
  In addition, the storage tier may filter portions of the object and transfer only the required elements to the Data Service server. This behavior occurs when the client designates the content-aware filtering conditions, and the required compute resource is available at the storage.

- **Client - Web Frontend Connection**
  If a particular client application uploads and/or downloads a large amount of object data to/from the Object Storage continuously, such a data transfer will be accelerated by the Open APN and the IOWN GF DCI.

- **Web Frontend - Aggregation / Dis-aggregation Server - Storage Connection**

The following figure shows images of the IDH Object Storage Reference Implementation Model with such ingenuity described above.

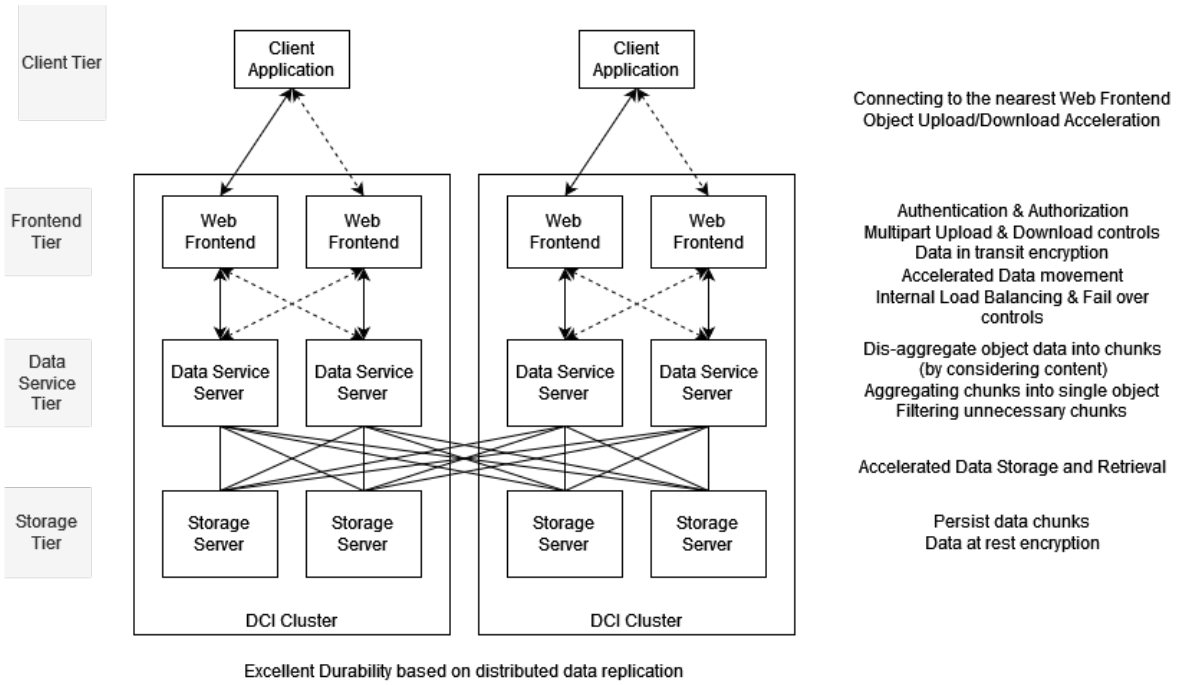



*Figure 19. Reference Implementation Models of IOWN Data Hub Object Storage*

## 5.2.6. Converged DB

In today's typical cloud-based implementation model, each IDH service provides a limited set of capabilities only. For instance, the Relational DB can manage tabular format data but not other types of data. The message broker can accept the massive data ingestion and deliver it to the client application in the almost* same order the data is ingested but does not support conditional read. Therefore, if the DX apparitions that IOWN GF assumes are built on top of such limited IDH services, many IDH services need to be combined, and massive data must be transferred across many components.

* If several shards are used to pipeline the same stream, the order will not be guaranteed for inter-shard reads.

The Converged DB is a new concept that eliminates these inefficiencies and streamlines DX application development by offering multiple capabilities over different types and formats of data from a single box. Some example scenarios are 1) querying and joining the tabular and the semi-structured data such as parquet and/or JSON file together in one place, 2) running transaction processing and heavy analytical processing in parallel, and 3) supporting data extraction through the conditional queries and data analysis with its inherent compute resources while accepting a large amount of data ingestion.

To build such a Converged DB, more resources need to be dynamically coupled because workloads and data volume of the Converged DB will fluctuate significantly. On top of it, various software-level ingenuity needs to be implemented to optimize, such as data placement and the number of copies to be kept in the server cluster, indexing of the content, and the data organization in both heap area and data persistence layer. The reference implementation model that reflects those points are described below:

- **Smart Client**
  Like the IDH Distributed RDB case, the clients will become smart to send the data access requests to the most appropriate Converged DB server where (most of) the request data exists in the cache.

- **Gateway**
  If the Converged DB needs to accommodate millions of clients from various locations, it may be challenging

to deliver the data placement and data distribution awareness to all of the clients. In this case, the system will place the gateway server tier that intermediates connection and data access requests from clients to the Converged DB servers.
The gateway may also include the security endpoint component if Converged DB is configured multi-tenant, as in the case of Distributed RDB.

- **Converged DB**
  In the Converged DB, the DB Engine will become “smarter” to optimize the query optimization plan by considering the available resources at each server as well as the network latency and the bandwidth between servers. It will determine which DB server(s) to preprocess data, whether to transfer queries or data across servers when inter-server processing is required, where and how to join and combine the data, etc.

- **Storage**
  In the Converged DB storage, a huge amount of data of multiple types is stored together. Various accelerators will be built in here to speed up data processing in consideration of such a situation. For instance, i) an accelerate to encrypt and decrypt data with different encryption keys based on data content-aware policy, ii) an accelerate that encodes and decodes in an optimal way according to the content of the data, and iii) an accelerate to manage summary index that describes the maximum and minimum values of each column for each fixed number of records, will be used.

- **Smart Client - DB Server Connection**

- **Inter-DB Server Connection**

- **DB Server - Storage Connection**

The following figure shows images of the IDH Converged DB Reference Implementation Model with such ingenuity described above.

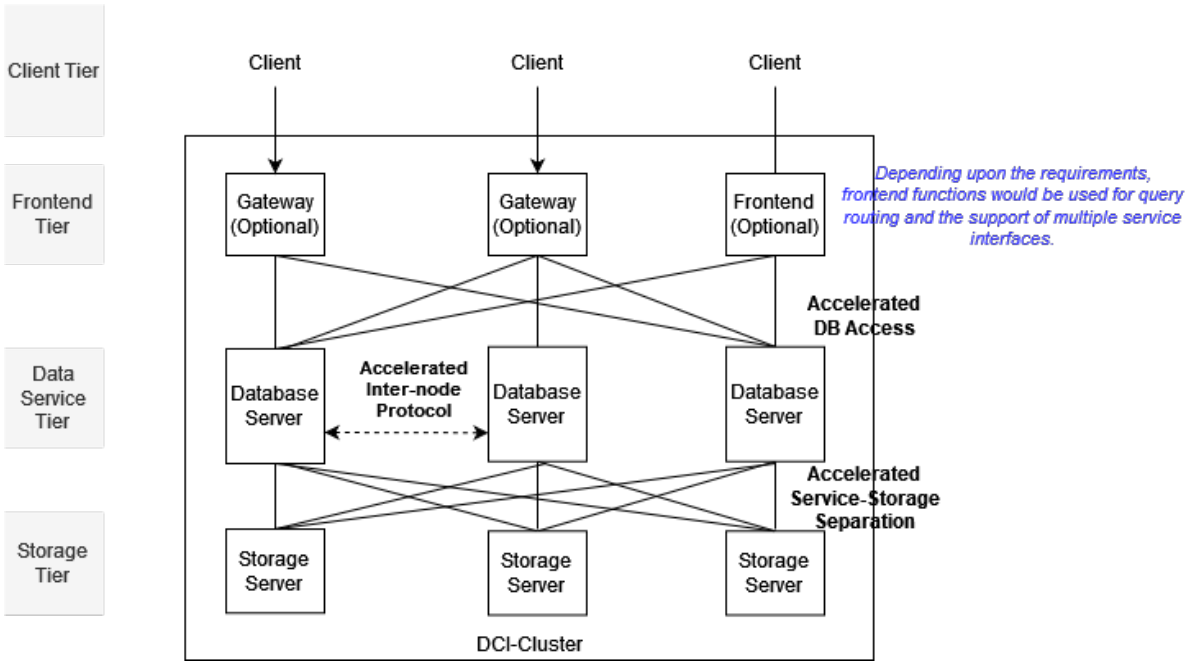



*Figure 20. Reference Implementation Models of IOWN Data Hub Converged DB*

## 5.2.7. Context Broker

The context broker has a distributed nature since it intermediates the access of context data between the clients and storage or other context broker instances and context sources. Context data is handled by different domains that collaborate with each other. However, the context consumer needs to issue a request to only a single context broker that is responsible for provisioning data from its own managed context and external context sources or domains.

The following diagram shows the Context Broker Reference Implementation Model adopted by IOWN Data Hub, with the interaction of subcomponents of the context broker divided into the client, frontend, data service, and storage tiers. The picture below is a meshed configuration; however, there might be a hierarchical configuration or hybrid configurations.

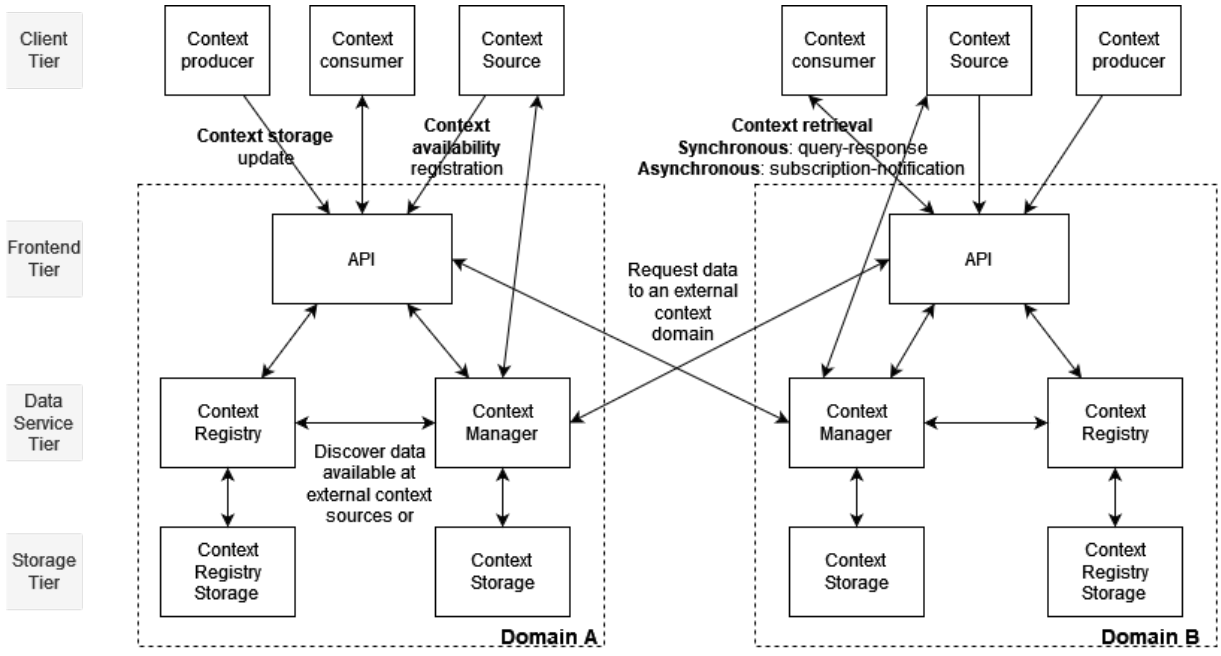



*Figure 21. Reference Implementation Models of IOWN Data Hub Context Broker*

In particular, we have the following components:

- **Context Producer**
  A context producer produces context (such as a sensor) and pushes it to the context broker. The storage managed by the involved context broker stores it.

- **Context Consumer**
  A context consumer requests context either synchronously or asynchronously. Context is requested with a context scope, such as data related to a thing, related to a type of a thing, related to a geographic scope, related to a temporal scope, or a combination of them. Synchronous requests follow the query-response pattern. The context consumer expects a single message response with the requested context. The requested context might be collected from any contacted domain, an external context source handled by the contacted domain or an external domain. In the case of asynchronous requests, a continuous context stream is established towards the context consumer. As soon new matching context is available either within the contacted domain, an external context source, or an external domain, it is notified to the context consumer. IOWN GF DCI might optimize the data traffic, especially for asynchronous communication, in order to avoid bottlenecks and reduce duplication of notifications. The subscription request might specify a *time Interval* value to configure a periodic notification of context. This configuration value should be taken into consideration for message exchange optimization.

- **Context Source**
  Context source is a source of context that exposes an API for context retrieval. This source can provision static context data, fresh sensor measurements, or the outcome of analytics processes.

- **API**
  API exposes a domain-specific language to access context data synchronously or asynchronously. API also exposes operations to register new context sources.

- **Context Manager**
  Context manager serves requests from context producers and context consumers. When a context producer sends a context update, the context manager stores it into the database and matches it against any established subscription. When a context consumer sends a context retrieval request, the context manager discovers context availability matching the request from the context registry and local index. The context might be available from zero to all of the following sources: domain context storage, locally managed context sources, external domain. In a synchronous request, the context manager gathers context from all the involved sources and retrieves a single context message. In case of asynchronous requests, the context manager establishes a subscription channel to any involved sources.

- **Context Registry**
  Context registry manages the list of available context sources and external domains with their respective set of contexts. The information contained in this component describes the topology of context sources and their available context. This information might be used for optimized traffic and intelligent context distribution.

- **Context Storage and Context Registry Storage**
  Basic service classes implement these components. A message broker, a graph store, and a distributed relational database can be used simultaneously for a different part of context data. In particular, the message broker handles the continuous updates of context from context producers and towards context consumers. The graph store keeps the semantic representation of the context and the relationships between pieces of information. The distributed relation database stores fundamental structured data such as spatial information and temporal information.

## 5.2.8. Virtual Data Lake (Federated Object Storage)

In the future digital era, more data generated from different locations will be managed in a distributed manner, such as in the edge clouds. Besides all that, Object Storage remains a core component to store big data as it provides the cheapest data storage mechanism. Thus, the Object Storage itself needs to become geo-distributed. We call such a geo-distributed Object Storage a “Virtual Data Lake.”
The following figure shows a new IOWN Data Hub concept on Virtual Data Lake that integrates multiple object storage locations. Each Object Storage may belong to a different organization, but each must be configured to be trusted and federated with one another.

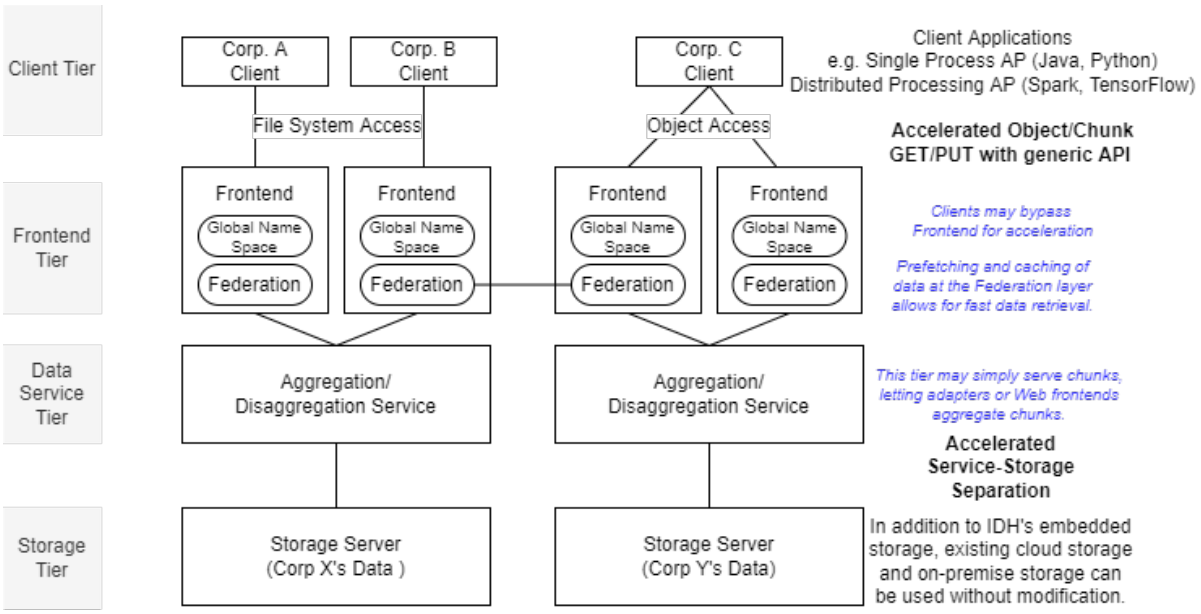



*Figure 22. Reference Implementation Models of IOWN Data Hub Virtual Data Lake*

To realize fast and quality Virtual Data Lake, IOWN Data Hub will add some ingenuity to it and what has been added to the basic Object Storage. Below is a list of the components and network connections that make up the IDH Virtual Data Lake and explanations of ingenuity implemented, if any.

- **Client**
  To realize the data transfer between multiple data owners of IDH, those tenants should be able to get data managed by other tenants through a general API or I/F regardless of location and data storing method.

- **Frontend**

  - Global Name Space and Unique Object ID

    By exposing the globally integrated namespace (Unified Namespace), the clients can access files without being aware of the location and owner of data. (E.g., a client application can get a file as if it gets the file from a directory of the local storage).

  - Federation
    Unlike simple Object Storage that collects all data in a single place, collecting only the metadata from a storage object and keeping the complete master data in the storage Tier makes it possible to provide centralized virtual management. Using this new approach, a client application can access the necessary data by referring to the metadata. By doing so, the system avoids creating the enormous amount of data copies that occur in today's implementation model, none of which support a geo-distributed solution. By requesting only the necessary data, the system ensures high-speed and efficient on-demand data transfers. Cache sharing function and pre-fetch function in accordance with the access pattern and frequency will be provided.

  - Basic authentication and authorization functions are provided to allow data access only for authorized clients. This model is equipped with an access control function unique to a data distribution platform that takes into account data provenance and lineage (e.g., X company data + Y company data => data processed by A company). Furthermore, as an optimization technique for multi-tenant premises, it is equipped with data acquisition speedup by sharing cache between tenants, data acquisition/deployment functions considering the locality of each component, and so on. To realize

such a function, frequent data transfer is required within the Frontend Tier (between Federation components), but the effect can be minimized by improving the efficiency of data transfer by DCI.

- **Aggregation / Dis-aggregation Server**
- **Storage Tier**
- **Client - Web Frontend Connection**
- **Web Frontend - Aggregation / Dis-aggregation Server - Storage Connection**

## 5.2.9. Virtual Data Lake House

In addition to the Virtual Data Lake concept described above, there will be more advanced requirements to analyze and utilize more data from various data sources. We call the system to support such requirements the “Virtual Data Lake House.”

The following figure shows the IOWN Data Hub concept of the Virtual Data Lake House that virtually aggregates and manages data from various data sources with multiple data owners.

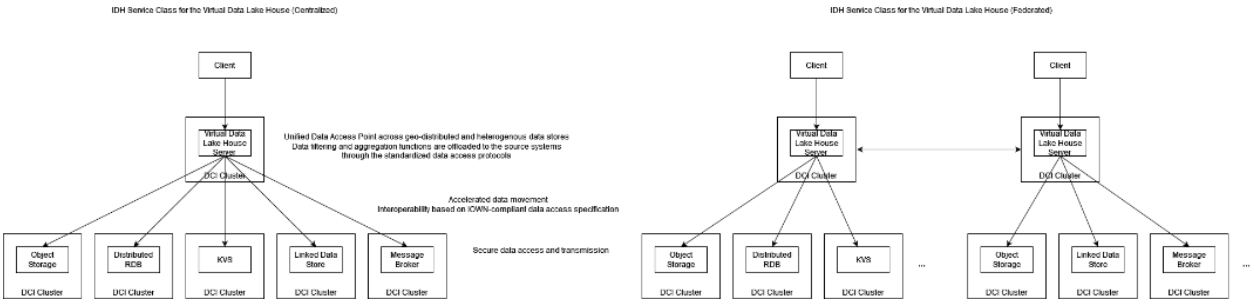



*Figure 23. Reference Implementation Models of IOWN Data Hub Virtual Data Lake House*

To enable such a Virtual Data Lake House concept, IOWN Data Hub will introduce some ingenuity on top of other IDH services. Below is a list of the components and network connections that make up the IDH Virtual Data Lake House and explanations of ingenuity implemented, if any.

- **Client**
- **Virtual Data Lake House Server**
  This component will provide a single point to access data for its client on top of various IDH Services. That means, if the Client application publishes SQL query to the Virtual Data Lake House, then the Virtual Data Lake House disassembles the query by considering other IDH Services to be contacted and forwards these disassembled queries to them. Then, each IDH Service that has received such a disassembled query will preprocess data as requested and reply only to the Virtual Data Lake House. Through such a mechanism, global data access is streamlined.
  To provide unified accesses across multiple IDH services, wrapper components that manage their interaction are added. In addition, inter-server parallel processing is performed on multiple Virtual Data Lake House servers to combine data from multiple IDH services and obtain the final result.
  In addition, it is possible to federate multiple Virtual Data Lake Houses. To realize a better society, it is necessary to combine and utilize various data from multiple companies safely. Virtual Data Lake House will be the foundation for such a better future.

- **Virtual Data Lake House Server - Other IDH service Connection**
  To accelerate data access, connections between Virtual Data Lake House and other IDH services will be accelerated by Open APN and IOWN GF DCI technologies.

# 6. Example Use Cases

### 6.1.1. Live 4D Map

To realize the future services described in the IOWN GF use case reports, it is required to manage and analyze various types of big data, know the real-world situation, record the events of concern, and predict the near future very precisely in real-time. For instance, the Live 4D Map needs to be created by combining multiple IDH services to build a safeguarding system. In the Live 4D Map, as in the following diagram, various types of data need to be managed, linked, and used together.

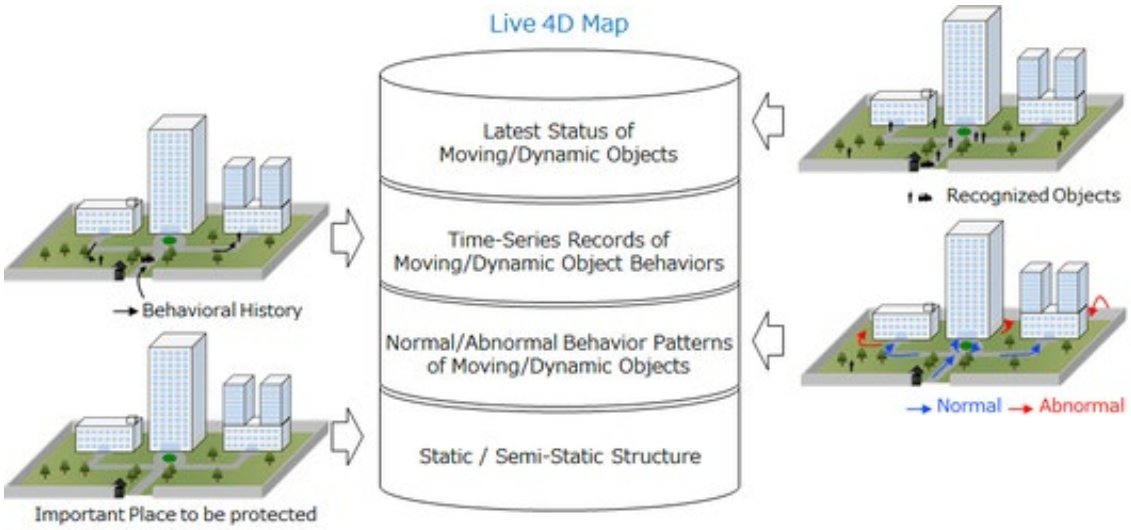



*Figure 24. Live 4D Map Concept*

Depending upon the use case scenario, the throughput and response time requirements vary. However, it would be reasonable to assume the following requirement numbers and IDH services to be applied to satisfy those numbers.

- **For the Latest Status Management Tier:**
  - **Requirements**
    - Manage more than millions of data records, each of which has several KB or more
    - Accept more than millions of updates per second at a latency of milliseconds or less to update the statuses of moving objects to be monitored, e.g., positions, actions, relationship with other objects, etc.
    - Respond very quickly to the query that retrieves historical records for specified objects at a latency of 10 milliseconds or less to support analysis of historical movements/behaviors of the objects.
  - **Solution**
    - Distributed RDB, which handles only fixed columns in tabular format, is the best way to get a high-speed overview of the situation. It is also good at extracting data at high speed based on attributes such as object type and coordinates.
- **For the Time-Series / Historical Records Tier:**

- **Requirements**
  - Manage more than billions or trillions of data records, each of which has several KB or more
  - Accept more than millions of inserts per second to record real-world events, object movement, actions, etc.
  - Respond to the query that retrieves historical records for specified objects very quickly, such as at a latency of 10 milliseconds or less, to support analysis of historical movements/behaviors of the objects.
- **Solution**
  - A converged DB will be the most appropriate solution because a huge amount of data with various formats needs to be managed and analyzed together. For instance, object recognition results may be represented as JSON, as it includes various attributes, and object movement trajectories may be managed as a collection of timestamps and coordinate key-value pairs. These data are managed together and associated as needed as in the Graph Store to understand the situation. Such workloads are handled well in the Converged DB.

- **For the Behavior Pattern / Knowledge Tier**
  - **Requirements**
    - Manage as many patterns as needed
    - Manage interactions and relationships between two or many objects
    - Compare each monitored object behavior with registered patterns, at a latency of 10 milliseconds or less
  - **Solution**
    - Graph store is best suited, as knowledge about behavior patterns is expressed not only by the target object but also by its relative relationship with surrounding static structures and other objects. Graph store can manage such a relationship between arbitrary objects well.

- **For the Static and Semi-static structure Tier**
  - **Requirements**
    - Manage geometry objects precisely, e.g., with cm-level accuracy
    - Support both the geospatial query and the update transactions quickly, at 10 milliseconds or less latency.
      Note: The semi-static structure includes desks, chairs, etc., which could be moved occasionally
  - **Solution**
    - A key-value store with geometry information and spatial indexes is a good choice because the static structures to be represented are very diverse, have various shapes and positions, and the number of records is huge, but the update frequency is not very high. In such cases, a KVS type Data Hub service is the best choice.

As highlighted in the above requirements, to realize IOWN GF use cases, the various types of Big Data need to be jointly managed and analyzed with very low latency and in a very scalable manner. As these requirements are impossible to achieve with today's solutions, IDH services built on the Open APN and the IOWN GF DCI will be required. The following figure shows such a possible high-level design of the Live 4D Map built on top of multiple IDH Services.

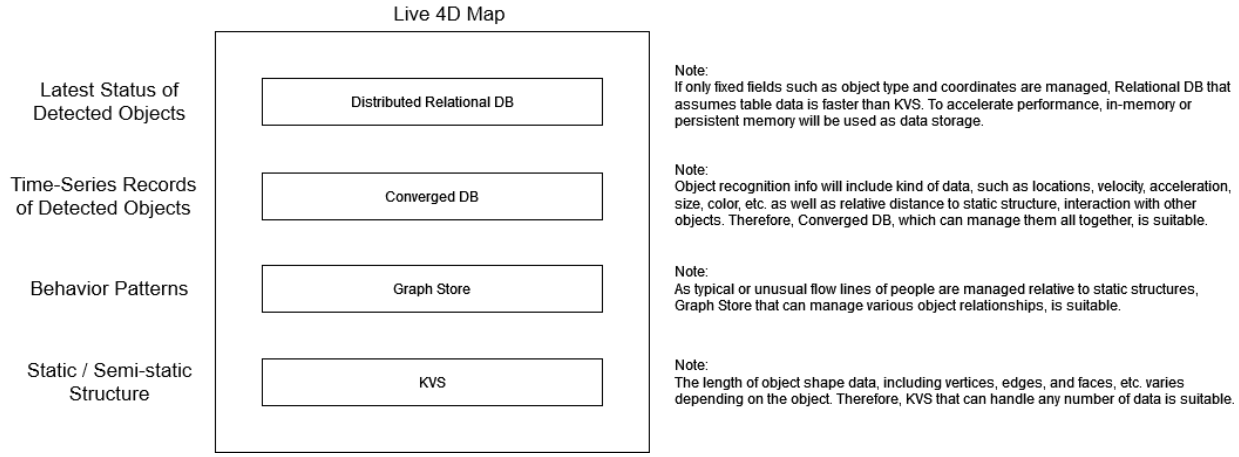



*Figure 25. IDH Service Mapping to build Live 4D Map*

## 6.1.2. Green Twin

Another example is the so-called Green Twin that monitors and coordinates people's activities to save energy while increasing their living value. For example, in urban areas, many cars drive the same place many times to find a parking spot. This wastes energy, emits $CO_2$, and reduces the value of people's lives. What would happen if we could monitor the population's living spaces, manage the efficiency of their energy use in digital space, and deliver appropriate recommendations to individuals to help them save resources? Obviously, the energy consumption in the real world would be reduced, and people's quality of life would be enhanced. In addition, the Digital Twin of a neighborhood environment, such as a parking spot, is related to other environments such as building management and public transportation. For example, parking status can be used to assess building activity patterns while still accommodating scheduled events to predict parking availability in the future. We can see that analytics services applied in the digital world are meant for understanding current situations, predicting future situations, and recommending actions. Digital Twin data is shared among twins, and analytics services might relate to multiple Digital Twins. The granularity of the Digital Twins is not fixed; it can be infinitely coarse or fine-grained. Digital Twins interact with each other in the sense of Digital Twin relationship a different granularity, Digital Twin analytics services related to multiple twins, and shared data among Digital Twins. In addition, Digital Twin represents physical twins in the real world. Therefore, the digital twin analytics process needs to have a spatio-temporal understanding of the real-world dimensions.

The figure below shows such a possible Green Twin use case.

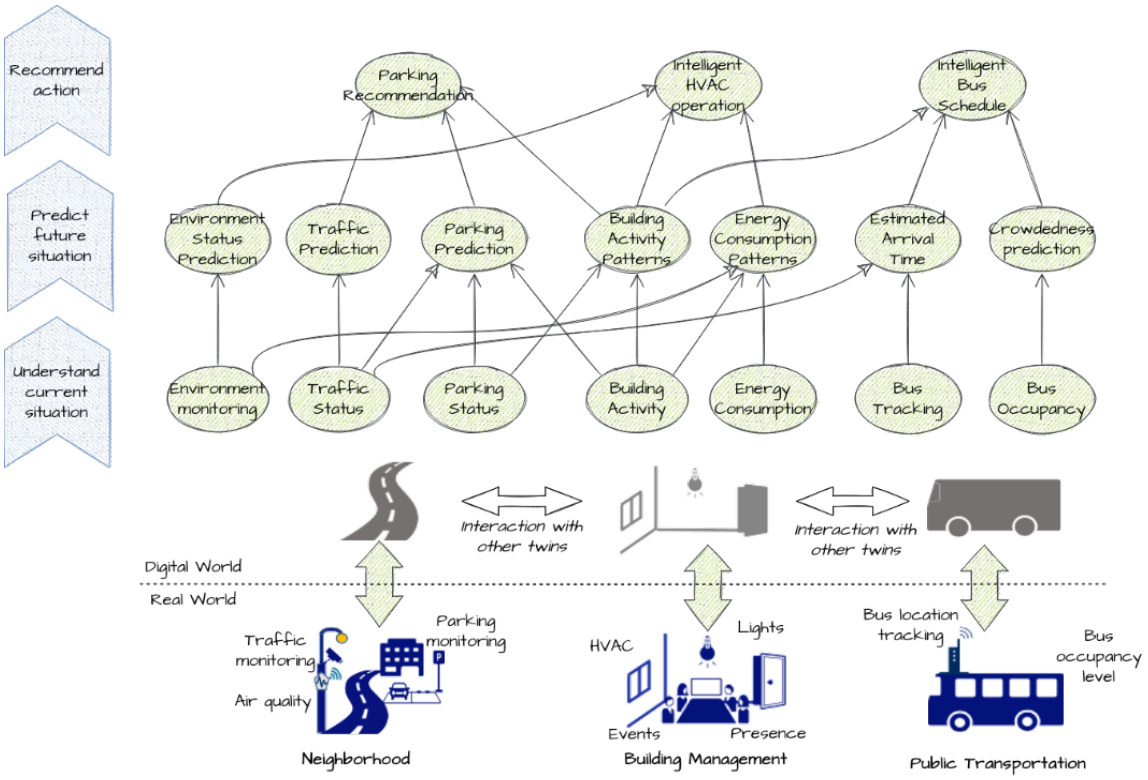



*Figure 26. Green Twin Use Case*

To realize such a Green Twin platform, it is required to curate, analyze and utilize a large amount of various data at any time. Such functional requirements are summarized below:

- Digital Twin represents the real world; thus, spatio-temporal characteristics should be considered as native information.
    - Geographic discovery of Digital Twin
    - Type-based discovery of Digital Twin (e.g., request for all *parking slots)*
    - Historical records of Digital Twin
- Digital Twins are related to other Digital Twins without fixed boundaries of granularity:
    - sensors to objects: e.g., *device1* and *device2* sense in *room1*
    - objects to events: e.g., *meeting1* happens in *room1*, *person1* and *person2* are involved in *meeting1*
    - objects to objects: e.g., *room1* is in *building1, parkingSlot1* is in front of *building1, car1* is parked in *parkingSlot1*
- Heterogeneous data sources:
    - Camera stream linked to a Digital Twin
    - Sensor stream linked to a Digital Twin
- Digital Twin framework comprises both data and analytics services. Analytics service might request data synchronously (query-response) and asynchronously (subscription-notification).

IOWN Data Hub services make it easy to build such a foundation at low cost. More specifically, it may be possible to efficiently collect a large amount of data from various devices through a message broker, associate data rapidly with a context broker, perform high-speed analysis on top of the context broker, and store the results to a KVS to deliver the recommendation to people or automatic systems. The figure below shows an image of Green Twin based on IOWN Data Hub services.

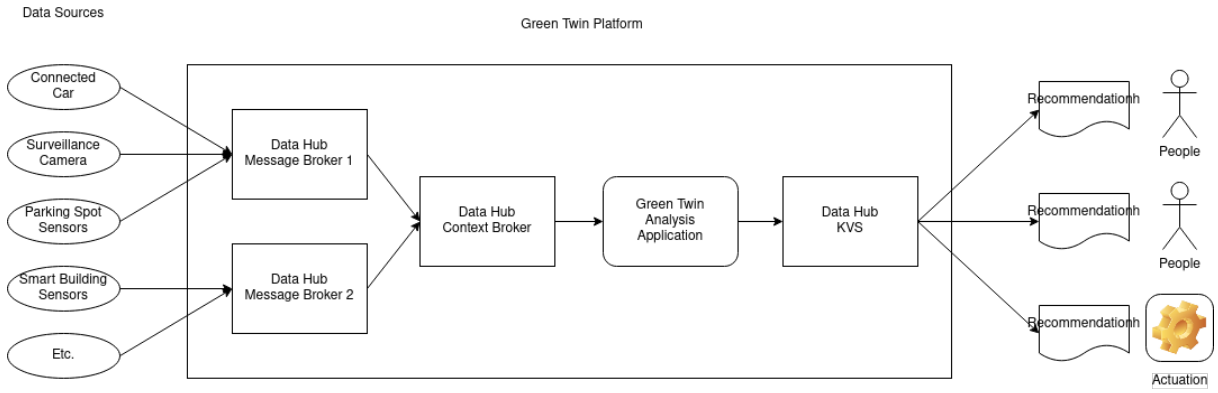



*Figure 27. IDH Service Mapping to build Green Twin*

If we built it on today's cloud environment, it would be an expensive system because we have to combine many Data Hub services, and there are various bottlenecks inside, as we have discussed in this paper. Using IOWN technologies, these issues can be solved because it improves the performance of the Data Hub services and makes them multi-functional. Thus, the number of Data Hub services is reduced, the system is simplified, and the system cost is saved.

# 7. Next Steps

This document summarizes the technical position of IDH in IOWN GF DCI and presents minimum viable versions that solve the bottleneck of existing cloud-based data processing. On the other hand, the analysis presented here is conducted based on some limited scenarios. It is likely that more advanced features will be required for the more advanced data processing flows that future applications using IOWN technologies will face. This section will briefly introduce the IDH features that may be needed for the next step.

## Integrated authentication and authorization

IDH needs to handle data from multiple organizations. While this document only describes simple requirements, more advanced functionality will likely be required when considering scenarios where various organizations use data. To utilize data from multiple data owners in a unified manner, the IOWN Data Hub needs a standardized API to support authentication/authorization and data management functions. This document version does not provide sufficient analysis of this functional requirement, but the next step should be to define the common functionality that IDH should support. This would include user/entity identification, authentication at an appropriate level of reliability, and flexible authorization of how the data is used.

## Policy control, enforcement, and auditing

Access to and usage of the data should be controlled based on the policies of the data owner, and the data owner or authorized entity should be able to review the history of data usage. There needs to be a mechanism for data owners and data users to agree on and approve the content of data use (e.g., data quality and data processing methods, etc.), which will also help protect privacy and ensure data authenticity. It is also necessary to guarantee the CIA (Confidentiality, Integrity, Availability) of the data stored and in transit and to be able to detect and respond to threats that compromise them at an early stage.

## Secure/Confidential computing

In a more advanced example, some of the IOWN GF use cases are scenarios where AI is required to process data from multiple organizations across a wide range of societies and cities to predict social trends. In such scenarios, the challenge is to simultaneously meet the sometimes-conflicting demands of the stakeholders involved, e.g., sharing the results of integrated processing of competitors' data without disclosing the data to each other, obtaining the results of analysis of one's valuable data by another company's secret analysis program without revealing the data and analysis program to each other, and obtaining only the results to solve business and social problems. For this purpose, we should consider using secure computing and confidential computing technologies, which have been the subject of much research and development in recent years, and execute them at high speed using DCI.

# References


| | |
|---|---|
| **[IOWN GF AIC UC]** | IOWN Global Forum, “AI-Integrated Communications Use Case Interim Report,” 2021. https://iowngf.org/use-cases/ |
| **[IOWN GF CPS UC]** | IOWN Global Forum, “Cyber-Physical System Use Case Interim Report, Version 2.0,” 2021 https://iowngf.org/use-cases/ |
| **[IOWN GF DCI]** | IOWN Global Forum, “Data-Centric Infrastructure Functional Architecture,” 2022. |
| **[IOWN GF Open APN]** | IOWN Global Forum, “Open All Photonic Network Functional Architecture,” 2022. |
| **[IOWN GF RIM]** | IOWN Global Forum, “Cyber-Physical System, Reference Implementation Model,” 2022 |

# History

| Revision | Release Date | Summary of Changes |
|---|---|---|
| 1.0 | January 27, 2022 | Initial Release |