InfluxDB 3.0: System Architecture

By Nga Tran, Paul Dix, Andrew Lamb, Marko Mikulicic

InfluxDB 3.0 (previously known as InfluxDB IOx) is a (cloud) scalable database that offers high performance for both data loading and querying, and focuses on time series use cases. This article describes the system architecture of the database.

Figure 1 shows the architecture of InfluxDB 3.0 that includes four major components and two main storages.

The four components each operate almost independently and are responsible for:

- ***data ingestion*** illustrated in blue,
- ***data querying*** demonstrated in green,
- ***data compaction*** shown in red, and
- ***garbage collection*** drawn in pink respectively.

For the two storage types, one is dedicated to the cluster metadata named ***Catalog*** and the other is a lot larger and stores the actual data and named ***Object Storage***, such as Amazon AWS S3. In addition to these main storage locations, there are much smaller data stores called ***Write Ahead Log*** (WAL) used by the ingestion component only for crash recovery during data loading.

The arrows in the diagram show the data flow direction; how to communicate for pulling or pushing the data is beyond the scope of this article. For data already persisted, we designed the system to have the Catalog and Object Storage as the only state and enable each component to only read these storages without the need to communicate with other components. For the not-yet-persisted data, the data ingestion component manages the state to send to the data querying component when a query arrives. Let us delve into this architecture by going through each component one-by-one.

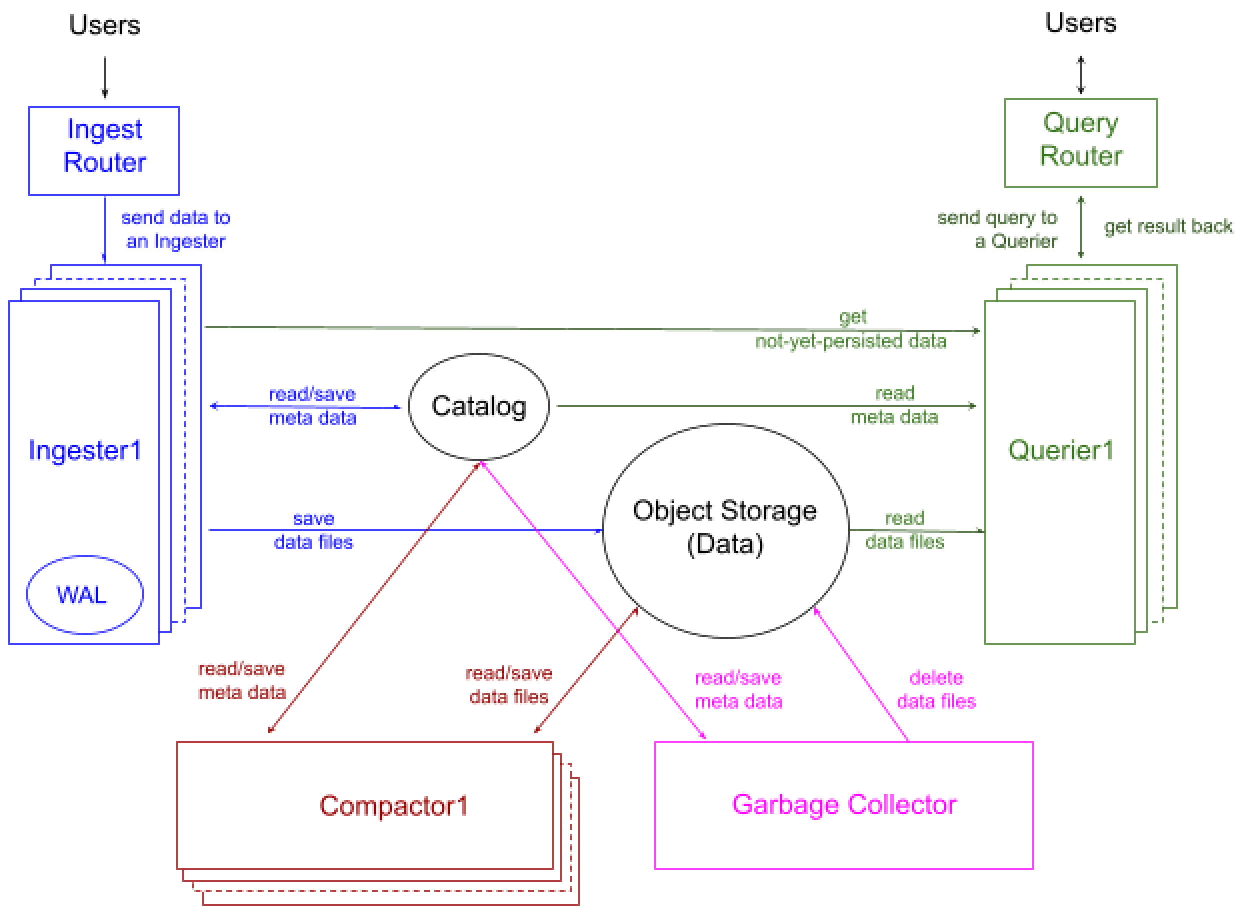

Figure 1: InfluxDB 3.0 Architecture

Data ingestion

Figure 2 demonstrates the design of the data ingestion in InfluxDB 3.0. Users write data to the ***Ingest Router*** which shards the data to one of the ***Ingesters***. The number of the ingesters in the cluster can be scaled up and down depending on the data workload. We use these scaling principles to shard the data. Each ingester has an attached storage, such as Amazon EBS, used as a write ahead log (WAL) for crash recovery.

Each ingester performs these major steps:

- **Identify tables of the data**: Unlike many other databases, users do not need to define their tables and their column schema before loading data into InfluxDB. They will be discovered and implicitly added by the ingester.
- **Validate data schema**: The data types provided in a user's write are strictly validated synchronously with the write request. This prevents type conflicts propagating to the rest of the system and provides the user with instantaneous feedback.

- **Partition the data**: In a large-scale database such as InfluxDB, there are a lot of benefits to partitioning the data. The ingester is responsible for the partitioning job and currently it partitions the data by day on the ‘time’ column. If the ingesting data has no time column, the Ingest Router implicitly adds it and sets its value as the data loading time.
- **Deduplicate the data**: In time series use cases, it is common to see the same data ingested multiple times, so InfluxDB 3.0 performs the deduplication process. The ingester builds an efficient multi-column sort merge plan for the deduplication job. Because InfluxDB uses DataFusion for its Query Execution and Arrow as its internal data representation, building a sort merge plan involves simply putting DataFusion’s sort and merge operators together. Running that sort merge plan effectively on multiple columns is part of the work the InfluxDB team contributed to DataFusion.
- **Persist the data**: The processed and sorted data then persists as a Parquet file. Because data is encoded/compressed very effectively if it is sorted on the least cardinality columns, the ingester finds and picks the least cardinality columns for the sort order of the sort mentioned above. As a result, the size of the file is often 10-100x smaller than its raw form.q
- **Update the Catalog**: The ingester then updates the Catalog about the existence of the newly created file. This is a signal to let the other two components, **Querier** and **Compactor**, know that new data has arrived.

Even though the ingester performs many steps, InfluxDB 3.0 optimizes the write path, keeping write latency minimal, on the order of milliseconds. This may lead to a lot of small files in the system. However, we do not keep them around for long. The compactors, described in a later section, compact these files in the background.

The ingesters also support fault tolerance, which is beyond the scope of this article. The detailed design and implementation of ingesters deserve their own blog posts.

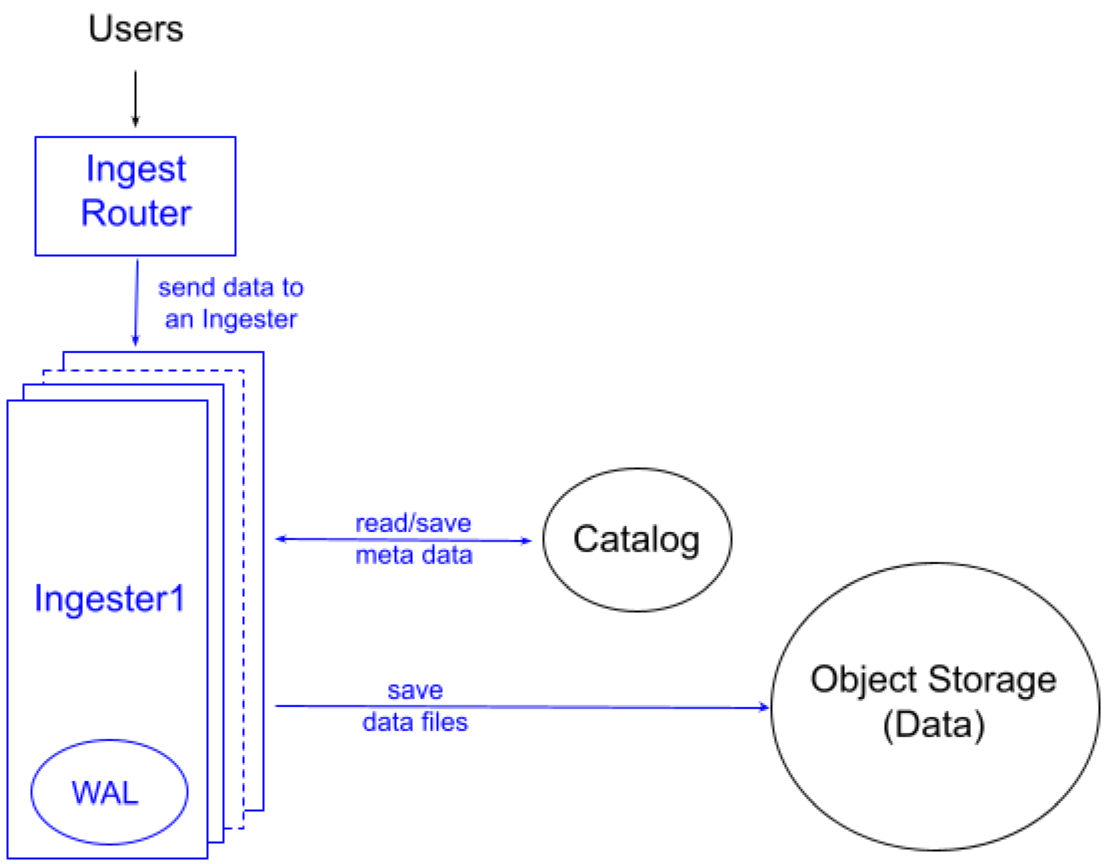

Figure 2: Data Ingestion

Data querying

Figure 3 shows how InfluxDB 3.0 queries data. Users send a SQL or an InfluxQL query to the **Query Router** that forwards them to a **Querier**, which reads needed data, builds a plan for the query, runs the plan, and returns the result back to the users. The number of queriers can be scaled up and down depending on the query workload using the same scaling principles used in the design of the ingesters.

Each querier performs these major tasks:

- **Cache metadata**: To support high query workload effectively, the querier keeps synchronizing its metadata cache with the central catalog to have up-to-date tables and their ingested metadata.
- **Read and cache data:** When a query arrives, if its data is not available in the querier's data cache, the querier reads the data into the cache first because we know from statistics that the same files will be read multiple times. Querier only caches the content of the file needed to answer the query; the other part of the file that the query does not need based on the querier's pruning strategy is never cached.

- **Get not-yet-persisted data from ingesters**: Because there may be data in the ingesters not yet persisted into the Object Storage, the querier must communicate with the corresponding ingesters to get that data. From this communication, the querier also learns from the ingester whether there are newer tables and data to invalidate and update its caches to have an up-to-date view of the whole system.
- **Build and execute an optimal query plan**: Like many other databases, the InfluxDB 3.0 Querier contains a Query Optimizer. The querier builds the best-suited query plan (aka optimal plan) that executes on the data from the cache and ingesters, and finishes in the least amount of time. Similar to the design of the ingester, the querier uses DataFusion and Arrow to build and execute custom query plans for SQL (and soon InfluxQL). The querier takes advantage of the data partitioning done in the ingester to parallelize its query plan and prune unnecessary data before executing the plan. The querier also applies common techniques of predicate and projection pushdown to further prune data as soon as possible.

  Even though data in each file does not contain duplicates itself, data in different files and data that is not yet persisted sent to the querier from the ingesters may include duplicates. Thus the deduplication process is also necessary at query time. Similar to the ingester, the querier uses the same multi-column sort merge operators described above for the deduplication job. Unlike the plan built for the ingester, these operators are just a part of a bigger and more complex query plan built to execute the query. This ensures the data streams through the rest of the plan after deduplication.

  It is worth noting that even with an advanced multi-column sort merge operator, its execution cost is not trivial. The querier optimizes further the plan to only deduplicate overlapped files in which duplicates may happen. Furthermore, to provide high query performance in the querier, InfluxDB 3.0 avoids as much deduplication as possible during query time by compacting data beforehand. The next section describes the compaction process.

The detailed design and implementation of the querier tasks described briefly above deserve their own blog posts.

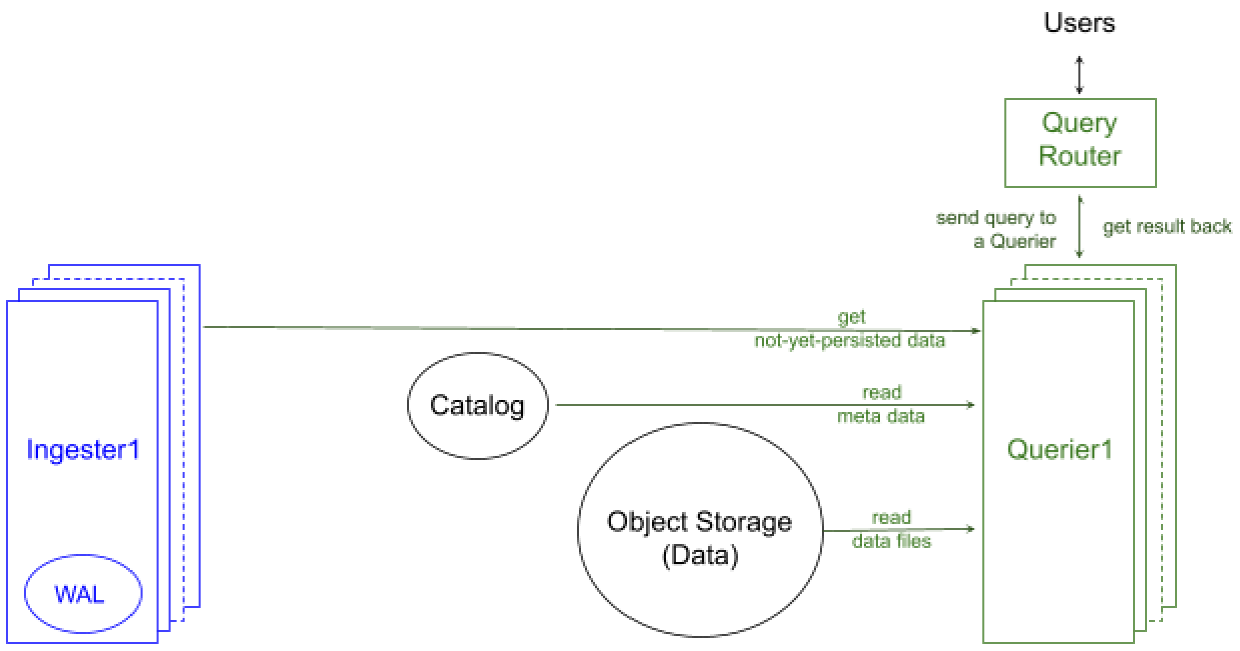

Figure 3: Data Querying

Data compaction

As described in the “Data ingestion” section, to reduce the ingest latency, the amount of data processed and persisted into each file by an ingester is very minimal. This leaves many small files stored in the Object Storage which in turn create significant I/O during query time and reduce the query performance. Furthermore, as discussed in the “Data querying” section, overlapped files may contain duplicates that need deduplication during query time, which reduces query performance. The job of data compaction is to compact many small files ingested by the ingesters to fewer, larger, and non-overlapped files to gain query performance.

Figure 4 illustrates the architecture of the data compaction, which includes one or many **Compactors.** Each compactor runs a background job that reads newly ingested files and compacts them together into fewer, larger, and non-overlapped files. The number of compactors can be scaled up and down depending on the compacting workload, which is a function of the number of tables with new data files, the number of new files per table, how large the files are, how many existing files the new files overlap with, and how wide a table is (aka how many columns are in a table).

In the article, Compactor: A hidden engine of database performance, we described the detailed tasks of a compactor: how it builds an optimized deduplication plan that merges data files, the sort order of different-column files that helps with the deduplication, using compaction levels to achieve non-overlapped files while minimizing recompactions, and building an optimized deduplication plan on a mix of non-overlapped and overlapped files in the querier.

Like the design of the ingester and querier, the compactor uses DataFusion and Arrow to build and execute custom query plans. Actually, all three components share the same compaction sub-plan that covers both data deduplication and merge.

The small and/or overlapped files compacted into larger and non-overlapped files must be deleted to reclaim space. To avoid deleting a file that is being read by a querier, the compactor never hard deletes any files. Instead, it marks the files as soft deleted in the catalog, and another background service named Garbage Collector eventually deletes the soft deleted files to reclaim storage.

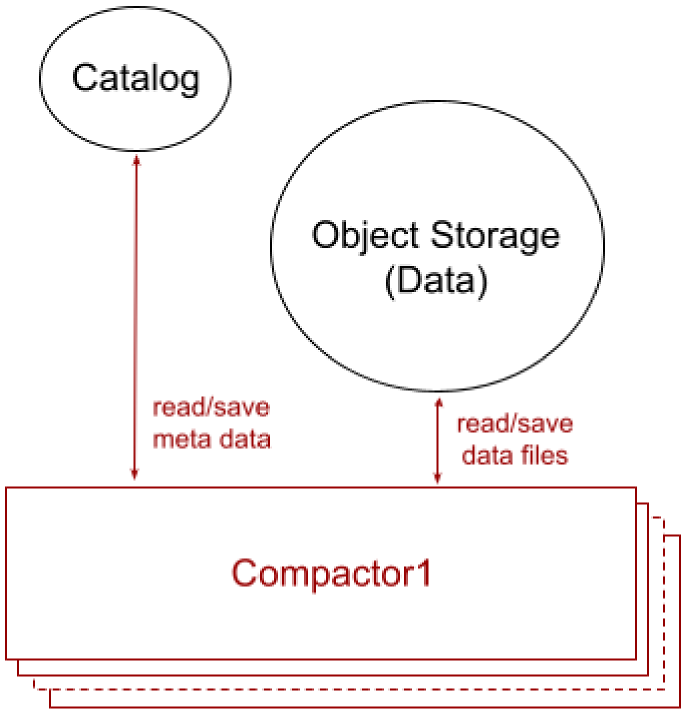

Figure 4: Data Compaction

Garbage collection

Figure 5 illustrates the design of InfluxDB 3.0 garbage collection that is responsible for data retention and space reclamation. **Garbage Collector** runs background jobs that schedule to soft and hard delete data.

Data retention:
InfluxDB provides an option for users to define their data retention policy and save it in the catalog. The scheduled background job of the garbage collector reads the catalog for tables that

are outside the retention period and marks their files as soft deleted in the catalog. This signals the queriers and compactors that these files are no longer available for querying and compacting, respectively.

Space reclamation:
Another scheduled background job of the garbage collector reads the catalog for metadata of the files that were soft deleted a certain time ago. It then removes the corresponding data files from the Object Storage and also removes the metadata from the Catalog.

Note that the soft deleted files came from different sources: compacted files deleted by the compactors, files outside the retention period deleted by the garbage collector itself, and files deleted through a delete command that InfluxDB 3.0 plans to support in the future. The hard delete job does not need to know where the soft deletes come from and treats them all the same.

Soft and hard deletes are another large topic that involves the work in the ingesters, queriers, compactors, and garbage collectors and deserve their own blog post.

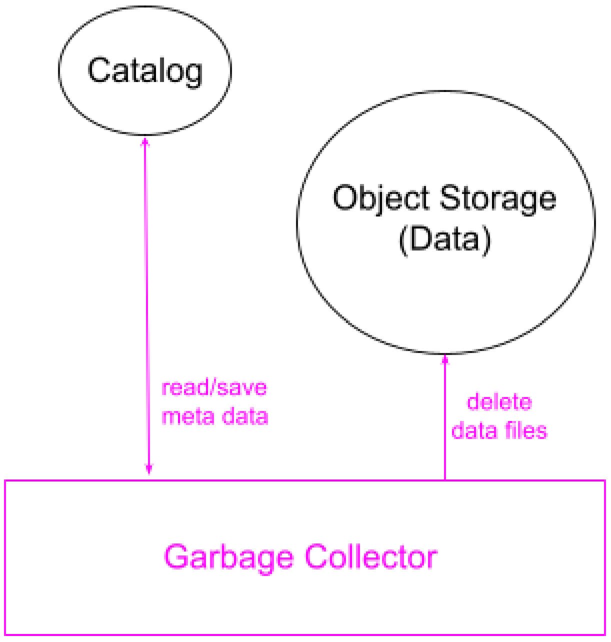

Figure 5: Garbage Collection

InfluxDB 3.0 cluster setup

Other than the queriers making requests to their corresponding ingesters for not-yet-persisted data, the four components do not talk with each other directly. All communication is done via the Catalog and Object Storage. The ingesters and queriers do not even know of the existence of the compactors and garbage collector. However, as emphasized above, InfluxDB 3.0 is designed to have all four components co-exist to deliver a high performance database.

In addition to those major components, InfluxDB also has other services such as **Billing** to bill customers based on their usage.

Catalog Storage

InfluxDB 3.0 Catalog includes metadata of the data such as database (aka namespace), tables, columns, and file information (e.g. the file location, size, row count, etc …). InfluxDB uses a Postgres compatible database to manage its catalog. For example, local cluster setup can use PostgreSQL while the AWS cloud setup can use Amazon RDS.

Object Storage

InfluxDB 3.0 data storage only contains Parquet files which can be stored on local disk for local setup and in Amazon S3 for AWS cloud setup. The database also works on Azure Blob Storage and Google Cloud Storage.

InfluxDB 3.0 cluster operation

InfluxDB 3.0 customers can set up multiple dedicated clusters, each operating independently to avoid "noisy neighbor" issues and contain potential reliability problems. Every cluster utilizes its own dedicated computational resources and can function on single or multiple Kubernetes clusters. This isolation also contains the potential blast radius of reliability issues that could emerge within a cluster due to activities in another.

Our innovative approach to infrastructure upgrades combines in-place updates and complete Blue/Green rollouts of entire Kubernetes clusters. The fact that most of the state in the InfluxDB 3.0 cluster is stored outside the Kubernetes clusters, such as in S3 and RDS, facilitates this process.

Our platform engineering system allows us to orchestrate operations across hundreds of clusters and offers customers control over specific cluster parameters that govern performance and costs. Continuous monitoring of each cluster's health is part of our operations, allowing a small team to manage numerous clusters effectively in a rapidly evolving software environment.