# Table of Contents

1. **Introduction to NosDB** .............................................................................................................. 6

2. **Components and Communication** .......................................................................................... 7
   2.1. P-nodes and R-nodes .............................................................................................................. 7
       2.1.1. Collections ..................................................................................................................... 8
       2.1.2. Cache ............................................................................................................................ 14
       2.1.3. Journal ........................................................................................................................ 15
       2.1.4. Op Log .......................................................................................................................... 16
       2.1.5. Storage Manager .......................................................................................................... 17
       2.1.6. File Storage .................................................................................................................. 17
       2.1.7. Index ............................................................................................................................ 18
       2.1.8. Communication between nodes ..................................................................................... 19
       2.1.9. Client operation on a node ............................................................................................ 20
   2.2. Shards .................................................................................................................................... 22
       2.2.1. Shard Behaviors ........................................................................................................... 23
       2.2.2. Communication between shards .................................................................................. 24
   2.3. Database Cluster .................................................................................................................... 25
       2.3.1. Shards ........................................................................................................................... 25
       2.3.2. Configuration Cluster .................................................................................................. 25
       2.3.3. Distributor .................................................................................................................... 27
   2.4. State Transfer Mechanism ................................................................................................... 27
   2.5. Shard Distribution Strategies ............................................................................................... 28
       2.5.1. Hash based Distribution Strategy .................................................................................. 28
       2.5.2. Range based Distribution Strategy ................................................................................ 29
       2.5.3. Non-Sharded Distribution Strategy .............................................................................. 31
   2.6. Replication ............................................................................................................................ 32
       2.6.1. Replication Process ....................................................................................................... 32
       2.6.2. Behavior ....................................................................................................................... 33
   2.7. Node Election Mechanism .................................................................................................... 34
   2.8. Split-Brain Handling ............................................................................................................ 36

3. **NosDB Data Models** ................................................................................................................. 37
   3.1. Embedded Data Model ........................................................................................................... 37
   3.2. Non-Embedded Data Model ................................................................................................ 38
   3.3. Hybrid Data Model ............................................................................................................... 39
4. Client Operations ............................................................................................................................................. 40
  4.1. Write Operations ....................................................................................................................................... 40
    4.1.1. Write Concern...................................................................................................................................... 41
  4.2. Read Operations ......................................................................................................................................... 42
    4.2.1. Read Preference.................................................................................................................................. 42

5. Storage ........................................................................................................................................................... 44
  5.1. In-Memory Cache ....................................................................................................................................... 44
  5.2. LMDB Storage Provider ............................................................................................................................ 45

6. Backup and Restore ....................................................................................................................................... 46
  6.1. Backup ....................................................................................................................................................... 46
    6.1.1. Full backup ......................................................................................................................................... 46
    6.1.2. Differential backup ............................................................................................................................ 47
  6.2. Restore ...................................................................................................................................................... 48
    6.2.1. Full Restore ....................................................................................................................................... 48
    6.2.2. Differential Restore ............................................................................................................................ 49
  6.3. Roles of NosDB Components ...................................................................................................................... 49

7. Security Model ................................................................................................................................................ 50
  7.1. Authentication ......................................................................................................................................... 50
  7.2. Authorization .......................................................................................................................................... 50
    7.2.1. Resources ........................................................................................................................................... 51
    7.2.2. Roles .................................................................................................................................................. 51
    7.2.3. Permissions ....................................................................................................................................... 53
    7.2.4. Role-Action Chart ............................................................................................................................... 53

8. Encryption ...................................................................................................................................................... 55
  8.1. Encryption Process ................................................................................................................................... 55

9. Data Expiration .............................................................................................................................................. 56
  9.1. Expiration Properties ............................................................................................................................... 57
    9.1.1. Expiration Types ................................................................................................................................. 57
    9.1.2. Document Specific ............................................................................................................................... 57
    9.1.3. Execution Preference ............................................................................................................................ 57
    9.1.4. Monitoring ......................................................................................................................................... 57
    9.1.5. Limitations ....................................................................................................................................... 57

10. Indexing ....................................................................................................................................................... 59
    10.1. Index Creation ..................................................................................................................................... 59
NosDB Conceptual Guide

10.2. Index Types ....................................................................................................................... 60
10.2.1. Single Attribute Indices ................................................................................................. 60
10.2.2. Compound Indices .......................................................................................................... 61
10.3. Index Behaviors ................................................................................................................. 62
10.3.1. Sparse Indices .................................................................................................................. 62
10.3.2. Caching ............................................................................................................................ 62
10.3.3. Journaling ........................................................................................................................ 63

11. CLR Triggers ......................................................................................................................... 64
11.1. Registering and Configuring Triggers ............................................................................... 65
11.2. Trigger Behavior ................................................................................................................. 65
11.2.1. Failure conditions .......................................................................................................... 65
11.2.2. Updating Triggers .......................................................................................................... 65

12. Querying in NosDB ............................................................................................................... 66
12.1. Extended SQL Properties ................................................................................................. 66
12.1.1. Infinite Embedded Support ............................................................................................ 66
12.1.2. Infinite Array Support ..................................................................................................... 67
12.1.3. Wild Card Searching ...................................................................................................... 67
12.1.4. Parametrized Queries .................................................................................................... 68
12.2. Functions ............................................................................................................................. 68
12.2.1. Built-in Functions .......................................................................................................... 68
12.2.2. CLR Functions .................................................................................................................. 68

12.3. SQL Reference ................................................................................................................... 70
12.3.1. Reserved Keywords ....................................................................................................... 70
12.3.2. Binary Operators .......................................................................................................... 71
12.3.3. Logical Query Operators ............................................................................................... 71
12.3.4. Basic Query Operators .................................................................................................. 71
12.3.5. Aggregate Functions ...................................................................................................... 73
12.3.6. Scalar Functions ............................................................................................................. 73
12.3.7. Special Operators for Arrays ......................................................................................... 74
12.3.8. Miscellaneous ................................................................................................................ 75
12.3.9. Special Considerations for WHERE Clause ................................................................. 76

13. Query Optimization ............................................................................................................... 77
13.1. Query Execution Path ........................................................................................................ 77
13.2. Query Optimization in NosDB .......................................................................................... 80
13.2.1. Factors Affecting Optimization................................................................. 80
13.2.2. Optimization Process.............................................................................. 82
14. Joins Support in NosDB................................................................................ 83
  14.1. Join Architecture in NosDB...................................................................... 83
    14.1.1. Map Phase......................................................................................... 83
    14.1.2. Reduce Phase...................................................................................... 83
  14.2. Join Behavior in NosDB............................................................................ 84
    14.2.1. Correlation Names.............................................................................. 84
    14.2.2. Projection............................................................................................ 84
    14.2.3. Nulls.................................................................................................... 84
15. MapReduce..................................................................................................... 86
  15.1. MapReduce Architecture.......................................................................... 86
  15.2. How does MapReduce Work?................................................................... 86
  15.3. Example MapReduce Execution................................................................. 87
1. Introduction to NosDB

Welcome to NosDB! NosDB is a schema-less and scalable NOSQL database solution to handle ad-hoc querying on huge amounts of real-time, unstructured data. As NosDB scales out to accommodate the rapidly increasing volume of your data, it applies robust data distribution strategies to ensure availability and fault tolerance at all times. Keeping in mind the suitability of NosDB for Big Data applications, MapReduce and Aggregation support has also been introduced to dramatically enhance performance due to parallel processing.

NosDB features and tools are designed to be tuned flexibly into applications of any size – from small to enterprise-wide global installations.

Support

NCache provides various sources of technical support. Please refer to Alachisoft’s Support page to select a support resource you find suitable for your issue.

To request additional features in the future, or if you notice any discrepancy regarding this document, please drop an email at support@alachisoft.com.

Document Conventions

The following conventions in text have been used throughout this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bold</strong></td>
<td>Specifies terms of importance for the reader.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Specifies any chapter names referred to in text.</td>
</tr>
<tr>
<td>Monospace</td>
<td>Specifies JSON document snippets within the text.</td>
</tr>
<tr>
<td>Monospace</td>
<td>Specifies any SQL query examples within the text.</td>
</tr>
<tr>
<td>Monospace</td>
<td>Specifies any code snippet within the text.</td>
</tr>
<tr>
<td></td>
<td>Specifies additional and vital information for the user.</td>
</tr>
</tbody>
</table>
2. Components and Communication

In This Chapter:

P-nodes and R-nodes
Shards
Database Cluster
State Transfer Mechanism
Shard Distribution Strategies
Replication
Node Election Mechanism
Split-Brain Handling

NosDB is as an intricate structure built up of various components that serve independently as well as collectively to provide optimum performance. The most basic building block of the NosDB architecture is a node that makes up a shard, which is eventually part of a cluster.

2.1. P-nodes and R-nodes

![Image: A single node with components]

Figure 2.1.1. A single node with components

A node is a server that constitutes for the basic unit of a NosDB cluster. This is where the collections of documents, caching, storage and logs are found.
Multiple nodes form a shard, so the nodes need to be categorized within the shard. A node can be part of different shards, but cannot exist in the same shard more than once. There are two types of nodes:

1. **Primary (P-node)**

   The primary node caters both read and write operations, hence it will contain the maximum operations and will be the first one to be updated. Write operations are only performed on the primary and then reflected to the Replica nodes, if specified through the write concern. However, in case of a read operation, the client specifies its read preference i.e. whether it wants to read from the primary node or not. There can be only one primary node in a shard, which is selected after an election is conducted between the nodes.

2. **Replica (R-node)**

   The replica node, as the name indicates, is to ensure back up. Hence, it will only allow read operations which are synchronized with the primary node through a pull mechanism. As mentioned above, the client can specify if it wants to read the data from the R-node.

A node comprises of multiple components to offer reliable persistence and successful operations:

**2.1.1. Collections**

A collection in NosDB could roughly be compared to an SQL table in terms of collecting related data in one place. However, there are major differences between the behaviors of both.

2.1.1.1. **Primary Keys**

The user can nominate any fields(s) to be used for unique identification of the document, as with primary keys in SQL. For example, *CustomerID* in *Customers*. This SQL-like behavior has been introduced in NosDB since the distribution of documents performed by the Distributor is based on a unique primary key to distinguish each document, and the index maintained for the key allows faster access during data retrieval.

**Composite keys**

A composite key comprises of multiple fields. Individually these fields might not be unique, but using them as a composite key ascertains the uniqueness. For example, as shown in Figure 1, *CustomerName* and *CustomerAddress* are not unique independently. However, once combined, the composite key [*CustomerName + CustomerAddress*] acts as a unique identifier and differentiates the documents from each other (see Figure 2).
Figure 1: Using CustomerName or CustomerAddress as a single primary key does not yield uniqueness.

Figure 2: Using [CustomerName, CustomerAddress] as a composite primary key yields uniqueness.

Primary Key Behavior

- The primary field is only assigned at the time of collection creation.
- The user can list any single-valued, root-level attribute(s) as the primary key.
- The field(s) value must pertain to .NET primitive data types and JSONDocument.
- Composite keys are order-sensitive. This means if CustomerName was specified before CustomerAddress, the same order should be applied if the values are to be copied to another collection.
- If the primary key is not defined during collection creation, NosDB automatically adds a default field "_key" to be used as a PK internally. If during insertion of the data no value is assigned, a GUID value is assigned to _key automatically.
- DocumentKeyException is thrown if the user-defined primary key is not assigned a value. The error code range for this exception lies between 90000 and 94999.
Limitations

- Primary fields are immutable; they cannot be updated or deleted once assigned.
- Null primary fields and `string.empty` are invalid values.
- Multi-value fields like arrays and embedded JSON cannot be used as a primary key.

2.1.1.2. Rows vs. Documents

1. Format

Instead of fixed rows containing pre-specified attributes in SQL tables, collections store the data in the form of native and custom .NET objects and JSON-like data structures called documents.

The following is an example of a document for the collection `Employee`:

```json
{
    "EmpID": "emp123",
    "FirstName": "John",
    "LastName": "Smith",
    "Department": "Engineering"
}
```

2. Variable fields

Another difference between a row and document is that the same collection can contain documents of varying fields as NosDB is schema-less. There is no need to predefine the schema and fill in the values accordingly like in relational databases, making NosDB more dynamic and prone to accommodate a large variety of unstructured data. This also eradicates any unnecessary null values, making the dataset precise and compact.

Example:

```json
{
    "EmpID": "emp123",
    "FirstName": "John",
    "LastName": "Smith",
    "Department": "Engineering"
}
{
    "EmpID": "emp345",
    "Name": "Mary Evans", "Department": "Marketing",
    "Supervising": ["East Wing", "West Wing"]
}
```

3. Embedded Documents

A document can also contain embedded documents which eradicate the need to refer data from various collections. So instead of writing complex queries to resolve and gather data from referrals, related documents can be embedded to accumulate associated data in one place which can be accessed even with a single query.
An example of an embedded document:

```json
{  
    "EmpID": "emp345",  
    "FirstName": "Mary",  
    "LastName": "Evans",  
    "Marital Status": true,  
    "Address": {  
        "Street": "This Street",  
        "Town": "Houston",  
        "State": "TX",  
        "ZipCode": "77002"  
    },  
    "Department": "Marketing",  
    "Title": "Assistant Manager",  
    "Supervising": ["East Wing", "West Wing"]
}
```

Multiple collections reside on a single node. NosDB supports four types of collections:

### 2.1.1.3. Normal Collections

A collection is an assembly of similar documents. Each document has a unique identifier, the primary key for the collection. However, apart from the primary key, indices can be defined on other fields as well for finer granularity while accessing data.

For example, the collection Employee discussed previously will contain documents containing Employee information indexed on the field `_key`, but if you want to index records according to the Department of each employee, that is possible too. Refer to the chapter *Indexing* for more details. Any write operation performed essentially modifies the collection - be it updating, deleting or adding of a new attribute or document.
2.1.1.4. Capped collections

Capped collections are the same as a normal collection but with circular behavior. This results in documents being overwritten in a FIFO manner once the configured file size has been reached. Though capped collections limit storage of documents as they are bound by size and number of documents, they are ideal for logging purposes because of the same reason.

Moreover, the FIFO nature results in faster insert operations because the data is added at the tail of the collection - meaning no need to find for a slot in the collection. Similarly, the data is fetched just as efficiently because of the sequential insertion of data. Hence, there are no default indices in a capped collection.

![Capped Collection Concept](image)

2.1.1.5. Single Sharded Collections

A shard is a collection of nodes. The collections are usually distributed among the shards as the shards are added or removed. The distribution is explained in detail in Distribution Strategies. However, single sharded collections are stored in one shard and are not redistributed upon shard removal or addition. This means that there is no need to rebuild the index for these collections, as their location remains the same.

Note that a single-sharded collection can be transferred from one shard to another. Moreover, a single sharded collection can be promoted to multi-shard by selecting a clustered distribution strategy at any time.

2.1.1.6. Attachments

Apart from storing data in the form of JSON documents, NosDB also allows storing BLOBs (images, videos, documents) in optimized system collections as attachments. The data is converted into binary form which is stored in NosDB. Creating the attachment collection is optional, which is specified during database creation. Each database contains a single attachment collection.

**Additional Metadata**

NosDB also allows storing any additional metadata in JSON document format about the attachment so it is easier to query over the files.
For example, the following metadata can be added to attachments containing images of a trending event, like New Year.

```
{
    "FileType": "png",
    "Description": "Images from New Year – 2016 at 12 AM",
    "Tags": ["New Year", "2016", "1 January", "12 AM", "New Year Celebrations"]
}
```

Note that NosDB also stores system metadata about the collection when an attachment is inserted. This includes the file ID and server ID. Thus, the user-specific metadata is added as an embedded document within the system metadata file internally.

For more detail on Attachments, please refer to the chapter Attachments in Programmer’s Guide.
2.1.2. Cache

Each node contains a cache which provides high performance to handle any transaction load. Instead of accessing data from the underlying storage, it is efficient if frequently used data is stored in-memory, allowing faster access. The cache is shared among all collections on a node. It is up to the user to allow replication of operations to majority of the connected nodes, else a response is returned immediately after the operation is performed on the primary node.

**Cache during Write Operation**

When a client requests for a write operation (add/update/delete) to be performed, it is first logged in the cache of the P-node and then logged in Oplog after persistence. This does not lag the operations as the response is returned to the user after being logged in the cache. The persistence to the file storage takes place in the background, so the user does not experience any wait and can carry on the operations smoothly.

If the write operation is not to be persisted, the data in cache will be modified, but the file storage will not be updated with the latest operation. Care should be taken for persistence if the data is critical.

In case the client requests a journal response, a response is returned as soon as the operation is cached and logged in the journal cache for persistence. Else, the response is returned when the data is stored in memory, (i.e., in cache).

**Cache during Read Operation**

When a client requests for a read operation, the item is first checked in the cache of either the P-node or the R-node if it exists there. If it is not present in the cache, the item is loaded transparently from the external data store into the cache for any future requests and returns the response to the client.

**Eviction in Cache**

If the data keeps on accumulating in the cache without being handled properly, the cache will eventually become full. To cater this problem, NosDB provides **need based eviction** to accommodate fresher incoming operations.

In need based eviction, persisted operations remain in the cache unless cache size is full and space is required. After that, only the persisted operations will get evicted from the cache through the Least Recently Used policy.

However, if the cache is full and all items are waiting to be persisted, the new operations will have to wait. This wait is expected in nanoseconds or milliseconds in the worst case. In this situation, the client will receive a delayed response for the operations performed.
2.1.3. Journal

The journal is a keystone in ensuring persistence. It is a temporary log file (`.NCJ`) found in each node, which tracks all in-memory client operations to prevent data loss on system crash or any failure causing data loss. It is created on runtime and deleted as soon as the operation is completed.

The size of a journal file is configurable. A journal may consist of one or more files, which are created once a file reaches its maximum file size. If the size of the operation is greater than the journal file, multiple files will be generated to accommodate the operation.

Journal Cache and Persistence

The journal cache resides with the journal log file. If an operation is specified to be persisted, it will first be stored in the journal cache and then get logged in the journal. The client will receive the response as soon as the operation is logged. Once it is persisted to the file storage system, it is removed from the journal log, completing the persistence process.

![Figure 2.1.4. Persistence Process](image-url)
2.1.4. Op Log

The Oplog is a database storage which logs the client operations in the same sequence as they are performed on the node. The operations are performed on the primary node (P-node) and initially logged in the primary’s local Oplog. This log helps in synchronizing all of the nodes in the shard, as the R-nodes (replica nodes) pull the operations on the basis of the primary Oplog.

The Oplog is circular: it will overwrite the logged operations (maintaining the sequence) once a specific duration is over so that its reserved space is reused. The duration and size of the Oplog is configurable.

Each log entry in the Oplog has a unique identifier which is a numeric value consisting of the timestamp and other internal information. The R-node periodically checks for any differences between the values of the primary Oplog and its local Oplog. The operations which have the latest operation ID or a difference in the value will get pulled by the R-node asynchronously, enabling synchronization on all nodes.

In case of an R-node down, it is necessary that the primary Oplog duration should be greater than the down time of the node. If the node rejoins, the primary Oplog should still contain the last entry which existed in the local Oplog (Figure 2.1.5), so that the operations may be pulled from that entry onwards (Figure 2.1.6).

In case the entries in the primary Oplog have been overwritten by the time the replica node rejoins, complete state transfer will take place between the nodes to ensure data consistency.

![Figure 2.1.5. Replica node leaves with last entry 12190012](image)

![Figure 2.1.6. Replica node rejoins and pulls operations from P-node after entry 12190012](image)
Oplog Cache

Similar to the Journal Cache, the client operations which modify the database are first stored in the local and journal caches, and then transferred to the Oplog cache. From here, once the operations are persisted, they are logged in the Oplog file.

2.1.5. Storage Manager

A storage manager acts as an interface between the underlying physical storage and the logical layer consisting of the client’s applications and operations. It communicates with the file manager of the file storage system to manage and ensure efficient document storage in the files due to the constant data transaction and distribution. This is why it has to guarantee that the adding, updating and fetching of data into and from the file system is time and space efficient.

Since the data being added is schema-less, the storage has to be flexible enough to host a variety of formats being added to it.

The storage manager also contains the storage provider, through which the actual file writing process takes place. The storage provider facilitates the manager in storing the data; hence NosDB utilizes Lightning Memory-Mapped Database (LMDB) as its third party storage provider which is a database itself and undergoes the same process internally to persist the data. For more detail on Storage, refer to the chapter Storage.

2.1.6. File Storage

NosDB stores the documents in a file storage system. A file is basically a sequence of binary data and records which are stored on secondary storage.

NosDB provides configurable storage file size for capped collections, while all other collections do not have any limit for their size. However, it is up to the user to specify whether the data is stored to a single file or multiple files. The user can specify the maximum file size where the database expands up to the specified size after which a new file is created. In case a single file store is preferred, the database expands indefinitely until the storage medium is full.
2.1.7. Index

In order for any data storage to work efficiently, indexing is as mandatory as its underlying infrastructure. Indexing simplifies the search method of selecting the requested query results. For example, on the request for an update operation on a document, NosDB can go either of the two ways:

1. The primitive, brute force scanning of the whole dataset to find the document to update,
2. Finding the related index and its corresponding document.

Needless to say, option 1 is a waste of resources and performance. In the time while the collections are being scanned, the user might perform more operations which will keep on pending, resulting in an unnecessary lag. Thus, the documents are indexed as soon as they are cached in NosDB and follow a proper procedure to persist the indices for even faster access to the documents.

NosDB index is in the form of a B-tree data structure. It stores the Document ID as its value in a sequential fashion, which makes traversal even simpler.

There are three modules involved in NosDB indexing:

2.1.7.1. Index Cache

Just like the Journal and Oplog, the Index has an Index Cache which contains the metadata of the indices created, i.e., the document IDs of the indices maintained and logged. As soon as an operation is stored in-memory, its index is stored in the index cache. Once the operation is persisted to file storage, the information in the index cache is persisted to the index store.

On system restart, the indices are reloaded and the document IDs are stored in the index cache to enhance performance.

2.1.7.2. Index Store

The index store is a dedicated physical storage for indices. It is separate from the file storage of documents, allowing faster accessibility of indices and analogous documents. Once the document is persisted in the file storage, its corresponding index is persisted in the index store in the form of a B-tree. The indices are identified by the document ID.

2.1.7.3. Index Log

The index log (.NSI file) stores each index activity like creation, build progress and removal. Once the data is persisted in the file and index store, the indexing information is persisted in the index log as well. The index log is a permanent log, so there is no information loss when the system restarts because the indices’ build progress can be determined through the log.

The read/write response is not returned to the user until the indices are logged.
2.1.8. Communication between nodes

Communication between nodes can occur in the following scenarios:

1. Replication

The nodes communicate with each other on the basis of a pull-mechanism. All the operations are initially performed and logged on the primary node. This is where the Oplog comes in handy. The R-node periodically checks for any differences between the values of the primary Oplog and its local Oplog. The operations which have the latest operation ID or a difference in the value will get pulled by the R-node asynchronously. Refer to Replication for more in-depth detail on replication.

![Figure 2.1.7: Communication between nodes during replication](image)

In case the entries in the primary Oplog have been overwritten by the time the replica node rejoins, complete state transfer will take place between the nodes to ensure data consistency.

2. Primary node down

In the event of a primary node down, the nodes can conduct elections to elect a new primary. The nominating node broadcasts its information like election ID, priority and Oplog entry, and the voting nodes return a response in favor or against the node. For more detail on elections in NosDB, refer to Election Mechanism.
2.1.9. Client operation on a node

2.1.9.1. Client Operation Types

The client is liable to perform both read and write operations on NosDB, which behave accordingly to assist scalability and fault tolerance. An atomic operation is performed at a document-level.

- **Write operations**

Write operations essentially add or modify the documents in NosDB. Since write operations are persisted to storage, it is beneficial to persist them in bulk, preventing network choke. Thus, the write operations in NosDB are first cached in-memory and then persisted to storage after reaching a specific bulk size.

As persistence of any operation to the file store is asynchronous, the client has the option of specifying the write-concern of the operation. This means that the client can either receive the response:

- As soon as the operation is cached in-memory, or
- After the operation is actually persisted in the journal.

- **Read Operations**

Read operations in NosDB include fetching stored data through API calls and querying. Techniques like Indexing boost read performance as there is no need to traverse the whole data set.

As explained [P-nodes and R-nodes](#), all nodes allow reading of data but only the primary node can cater write operations. Hence, the read-preference for the node can be specified by the client – indicating whether they want to read data from primary or replica node.

For in-depth detail about client operations in NosDB, please refer to the chapter Client Operations.
2.1.9.2. The Process: From Client to File Storage

Let us consider that the client performs a write operation on NosDB. It is up to the client whether the operation should be persisted or not. The following events take place within the node upon the operation:

![Flow of user operation on a node](image)

**Figure 2.1.8: Flow of user operation on a node**

1. Operation moves in-memory - it is cached to be persisted later. At this stage, if the write-concern has been set to in-memory, the response is returned.
2. The index for the operation is cached in Index cache. It will be moved to the index store once the operation is persisted.
3. Operation is cached in Journal cache.
4. Operation is persisted in Journal log. At this stage, if the write-concern has been set to 'journal', the response is returned.
5. Operation is cached in Oplog cache. It will be moved to the Oplog once the operation is persisted.

The following events take place in the background on independently running threads, thus there is no deterministic sequence of when the activities are performed:

- **FS** - Operation is persisted to File Store.
- **OL** - Operation is persisted to Oplog. It will be replicated to the replica node through the Oplog.
- **IS** - Index is persisted in Index store. Any future operation requiring an index will be loaded to the index cache through the index store.
- **IN** - Index information and status is persisted in index log.

Once the operation has been persisted, the entry is removed from the journal log as it is temporary.
If the operation fails due to any reason, it is rolled back from all logs and the cache. The user will not be notified of the failed operation, but the failure and removal will be logged.

2.2. Shards

As the name indicates, a shard is a partition of NosDB which further consists of multiple nodes bound to each other in a replicated topology. A node may exist in different shards but it cannot exist more than once in the same shard.

Each shard contains only one P-node, and can contain a configurable number of R-nodes. Thus each shard can have a varying number of replicas in the topology as configured by the user. More than one backup of the data can be created by increasing the number of R-nodes in the shard. The number of backups is equal to the number of the nodes. This partitioning and replication ensures availability because there is no single point of failure and prevents data loss even if one shard or node is down.

There is no automatic node assignment and state transfer, so the P-node has to be specified through election mechanism.

Each shard receives a partition of the data from the client, so the data volume keeps on varying between the shards as NosDB scales out or on shard removal. Partitioning also ensures one-hop operations, as the document is accessed through that specific node. NosDB provides three data-distribution strategies for sharding:

1. **Hash based**
   Data is distributed amongst all shards on the basis of the hash code of NosDB’s hashing algorithm. A hash based distribution map is generated by Configuration Server, which then distributes the data buckets to all shards. This strategy auto balances the load on each shard. For more detail, please refer to the chapter **Hash based distribution strategy**.

2. **Range based**
   Data is distributed amongst the nodes according to the ranges specified for each shard by the user. The range determines the number of documents per collection in a shard. For example, Shard A will receive 1-50 documents of the Employee collection, Shard B will receive 50-78, and shard C will get the remaining, i.e., 78-100 documents. For more detail, please refer to the chapter **Range based distribution strategy**.

3. **Non-sharded**
   Data is directed towards just one shard, where additional shards are only recorded in the list of shards. In case the shard with the data is removed gracefully, state transfer will occur between this shard and the next shard in the list. In case the shard is removed forcefully, data loss will occur because all data has been stored in that shard. For more detail, please refer to the chapter **Non-sharded distribution strategy**.
2.2.1. Shard Behaviors

State Transfer on Shard Addition

When a new shard is added to the cluster, the distribution strategy is applied again to reallocate the data through state transfer. The shards are connected to each other through their respective P-nodes, which have the latest operations logged on their Oplog. This can trigger any of the following cases:

1. During state transfer, if the P-node changes at the target shard (the newly added shard in this case), state transfer will continue from where the last P-node left, eliminating the need for a complete state transfer.

2. However, if the P-node changes at the source shard, it will not impact the state transfer as the next P-node will be considered for state transfer instead.

Primary Node Down

As mentioned before, all read and write operations can be handled by the P-node, but the R-nodes are Read-only. So it is necessary that the P-node remains active at all times.

In case the P-node does go down, the next connected node will take over as the new primary using an election mechanism to ensure fault tolerance. Once the downed P-node becomes active, it will update itself by replicating the latest state of the acting primary and then conduct elections.

A shard is still considered a part of the cluster even if all shard nodes are down. This is because the shard holds specific values of the distribution map, and the read/write operations which are to be performed on those values are still being targeted on that shard, regardless of its nodes’ state. However, the operation would fail as the nodes would be unavailable to allow the respective modification or fetching of data.

Shard Removal

A shard is available until and unless forcefully or manually removed by the user. On shard remove, the data is redistributed depending on the removal mechanism.

NosDB offers two mechanisms to remove shards:

1. Graceful remove

Before the shard goes down, the data is redistributed among the remaining shards during its existence. This is made possible by triggering state transfer and once the data has been completely transferred to the other shards, the shard is removed wholly by configuration server.

   - In case of hash based, the distribution map will be regenerated and the shards will be assigned accordingly.
   - For range based, the user will explicitly specify which shard the range(s) should be moved to.
   - For non-sharded strategy, state transfer will occur between the shard being removed if it contains the data and the next shard in the list of available shards.

This mechanism guarantees no data loss during the whole transaction, and ensures smooth performance of operations.
There should be at least one shard other than the shard being removed otherwise user will not be allowed to remove shard.

2. **Forceful Remove**

A shard can also be removed forcefully by the user at any time. In such a case, the shard will be removed immediately without any state transfer, resulting in data loss for its operations.

- For hash based, there will be data loss as the new operations will be redirected to the new distribution, and the pending tasks of the removed shard will be lost.
- For range based, the ranges will be distributed to other shards using Round Robin strategy. Operations will be performed on the newly assigned shard immediately.
- For non-sharded strategy, data will be lost because the whole data has been stored in one shard.

2.2.2. **Communication between shards**

Multiple shards form a cluster. This creates the need for communication between the shards just like node-to-node communication within a shard. Hence, the shards are connected to each other through the primary nodes of each shard. In case the primary node of a shard goes down as explained above, the next elected P-node will connect with the other shards.

![Diagram showing communication between shards](image)

**Figure 2.2.2: Primary-to-primary communication between shards**

- Primary Shard A is connected to primary Shard B and vice versa.
- Primary Shard A is connected to primary Shard C and vice versa.
- Primary Shard B is connected to primary Shard C and vice versa.
2.3. Database Cluster

A NosDB database cluster is made up of three components: shards, configuration cluster and distributor. This is the component which actually communicates with the clients. As a whole, it demonstrates the Partition-of-Replica topology – partitioned shards and replicated nodes.

2.3.1. Shards

Multiple shards form a database cluster. In other words, the database cluster contains partitions where the data is distributed for faster access and higher performance. So when a client sends a request to the cluster, the shard distribution map has already been assigned and the operation is only directed towards the specific shard.

2.3.2. Configuration Cluster

The configuration cluster is a composition of two configuration servers (CS) representing mirror topology in an active-passive concept. Only the active configuration server is open to communication with the clients in the configuration cluster, while the passive CS creates synchronous backup of the information in the active CS. The “clients” for the configuration server include nodes/shards, user and management tools.

The configuration cluster acts as a layer between the shards and distributor. It holds the metadata about the distributions and other configurations of the database on cluster level i.e. the overall structure of shards, how collections are distributed amongst the shards, tools, client connections, node status, backup/restore
configurations and synchronization details. Therefore, the configuration server is a centralized store for all the configurations of the cluster.

All database nodes which are part of the database cluster have information about the configuration cluster. The client only has to provide the IP of the nodes to initiate automatic establishing of a connection with the configuration cluster.

NosDB communicates with the configuration server in the following cases:

- When a new instance of the cluster is started
- When configurations are hot applied
- On primary node down
- During election process of primary node
- When a shard is added or removed, changing the distribution map

The configuration cluster is automatically managed in NosDB. This means that the user does not have to worry about the impact of adding/removing nodes or selecting between active and passive servers.

- **Node Addition and Removal**

Whenever a database node or shard is added to the database cluster, it is checked whether the node can act as a configuration server (CS) node or not. If the configuration cluster does not have the required two CS nodes, and the newly added node is not already a part of the configuration cluster, the node/shard will be added as a configuration server node in the configuration cluster.

Similarly, on node/shard removal from the database cluster, it is first checked if the node is a part of the configuration cluster or not. If it is not a part of the configuration cluster, it is simply removed and a new node is added in place of the removed node.

- **Active-Passive Server Node Selection**

The active/passive CS nodes of the cluster are automatically selected by honoring the node priority and configuration state of the node. A node with priority "1" is a stronger candidate to take on the Active role unless it has stale or no data. The node with priority "1" eventually takes over the Active role after transferring the state from the existing active node, while the existing active node (with lesser priority) is demoted to the Passive role.

The priority of the CS node is auto-assigned. It can differ from the priority assigned by the user to the database node.
2.3.3. Distributor

The cluster’s distributor serves as a proxy between the configuration service and clients. Accordingly, the direct interaction of clients takes place with the distributor, providing abstraction from the sharded environment. Hence, the distributor needs to be active at all times so that client connection is not lost with the cluster. To ensure such reliability, the distributor is dedicated a separate process so that even if the service process does go down, the clients will not experience any connection failure and neither will the client connection information at the distributor get lost.

Client-Distributor Interaction

On the first interaction of the distributor with client, the distributor contacts the configuration cluster layer and obtains the distribution map of the collections between the shards. Using that distribution map, it then directs the client towards the specified shard.

On successive interactions, the distributor already has the distribution map so it does not need to contact the configuration server again in order to distribute client connections amongst the shards. The next contact of the distributor with the configuration server takes place when the distribution changes. This prevents configuration files from being changed manually at the client side.

For more detail on the distributor process, refer to the chapter Optimization Process.

2.4. State Transfer Mechanism

NosDB offers State Transfer - a collection-level data transfer mechanism which is triggered after distribution of data is changed across a shard. State transfer involves the following state transfer managers at varying levels to manage data transfer:

- Node-level manager manages state transfer at collection level,
- Database-level manager manages collection data transfer and speeds up the process by using multiple threads per collection if required.

State transfer can either occur between two shards or two nodes of a single shard:

1. Inter Shard

Inter-shard state transfer takes place between two shards, where data is transferred bucket by bucket. The requesting shard initiates state transfer by locking that specific bucket on the configuration server, which is used for bookkeeping each bucket being transferred. The bucket being transferred is logged on all nodes of the providing shard until the bucket is fully transferred, after which the logged operations are transferred to the requesting shard.

2. Intra Shard

Intra-shard state transfer occurs between two nodes of the same shard to synchronize the state. Initial sync copies the whole data from providing node to the requesting node. A node uses initial sync when it has no data, such as when the node is new, or when the node has data but is missing a history of the shard’s replication. Intra-shard state transfer differs from inter-shard state transfer because data is copied from the primary rather than being moved, and logging of the bucket being transferred takes place on the requesting node instead of the providing node. Logging is incremental as the state transfer progresses.
2.5. **Shard Distribution Strategies**

NosDB offers various strategies to distribute the data among the shards of a cluster to balance the data load. These strategies make handling of data more flexible for the user, and implement smooth transitions without any user intervention in case of shard addition or shard removal.

2.5.1. **Hash based Distribution Strategy**

Hash based Distribution Strategy ensures random and almost equal data distribution across all the shards.

By default, NosDB has a 1000 buckets and each bucket holds some data based on the hash code of the partition key. A hash based distribution map is generated by NosDB's hashing algorithm, which then distributes the buckets among the shards. The map has to be regenerated on shard addition or removal, as the buckets have to be distributed equally among the shards.

There are two cases for state transfer:

**Shard Addition**

1. Adding first shard: All the buckets (for now 1000) will be assigned to this shard.

2. Adding second shard: 500 buckets will be assigned to this shard based on the hashing distribution map, resulting in 500 buckets for the other shard. State transfer will be initiated because the data has to be transferred to the buckets of the second shard from the first shard. However, during state transfer, the operation of the second shard’s buckets will still be directed towards the first shard until their state transfer is complete (this is default behavior of state transfer for all distribution strategies).
3. Adding a third shard will distribute the buckets in groups of 334, 333, 333 and state transfer will be initiated.

**Shard Removal**

The buckets will equally be distributed amongst the remaining shards. Shard removal triggers the graceful and non-graceful cases:

1. **Graceful remove**: The distribution map will be regenerated and the shards will be assigned accordingly, initiating state transfer.
2. **Non-Graceful**: The shard will be removed immediately without any state transfer, resulting in data loss of all the operations.

**2.5.2. Range based Distribution Strategy**

Data is distributed amongst the nodes according to the ranges specified for each shard by the user. For example, Shard A will receive keys within the range 1-50 of the Employee collection, Shard B will receive keys 50-78, and Shard C will get the remaining, i.e., 78-100 keys. Load balancing is either manual or automatic.

![Figure 2.4.2: Range Based Distribution](image)
Shard Removal

- **Graceful removal:** The user has to explicitly specify which shard(s) the range(s) of the removed shard should be moved to. Operations will be targeted towards the shard to be removed until the state transfer is complete.
- **Non-graceful removal:** Ranges of this shard will be distributed to other shards using Round Robin strategy. Operations will be performed on the newly assigned shard immediately.

Range Configurations

The range has to be defined upon the following rules:

1. Should be continuous
2. Should be defined in a sequence
3. Must not overlap
4. Start of current range must be equal to end to previous range.

- **Valid** configuration example:

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Shard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>Shard A</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
<td>Shard C</td>
</tr>
<tr>
<td>400</td>
<td>600</td>
<td>Shard A</td>
</tr>
<tr>
<td>600</td>
<td>1000</td>
<td>Shard B</td>
</tr>
</tbody>
</table>

- **Invalid** configuration examples:

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Shard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>Shard A</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>Shard C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Shard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>Shard A</td>
</tr>
<tr>
<td>400</td>
<td>500</td>
<td>Shard B</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
<td>Shard C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Shard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>Shard A</td>
</tr>
<tr>
<td>70</td>
<td>100</td>
<td>Shard B</td>
</tr>
</tbody>
</table>

State Transfer

Range Based strategy allows state transfer, but the following behaviors should be kept in mind:

- No state transfer occurs if a new range is added to a new shard.
- State transfer occurs if a range's shard is updated.
- If state transfer is in progress and the user tries to update the range, updating will fail until state transfer is complete.

**2.5.3. Non-Sharded Distribution Strategy**

NosDB also offers non-sharded distribution where the whole data is directed towards a single shard. This strategy allows you to add more than one shard but the data still goes to the configured shard. The newly added shard is simply added to a list of shards so that in case of shard removal, there is a shard to transfer the data to.

Removing a shard involves three different cases:

1. **Shard being removed is not the target for data:** In this case, the shard will be simply removed from the list of shards because there is no data transaction with this shard.

2. **Non-graceful removal of target shard:** This shard will be removed immediately without any state transfer and data will be redirected to the next shard in the list.

3. **Graceful removal of target shard:** The operations will be targeted towards this shard until state transfer is complete. All future operations will then be performed on the next shard in the list and the first shard will get removed. Figures 2.4.3 and 2.4.4 illustrate the process.

There needs to be at least one shard other than the shard being removed otherwise the user will not be allowed to remove shard.

---

**Figure 2.4.3. Graceful Removal of Shard A**

**Figure 2.4.4. Connection resumed with the next shard in list after state transfer**
2.6. Replication

The nodes in NosDB communicate with each other on the basis of the standard Pull-model. All the operations are initially performed and logged on the primary node. The Replica node periodically checks for any differences between the values of the primary Oplog and its local Oplog. The operations which have the latest operation ID will get pulled by the replica node asynchronously.

Replication involves the following components:

1. **Pull Model Task**
The pull model task runs at the replica node and is responsible for “pulling” the non-replicated operations from the primary node.

2. **Corresponder**
The corresponder runs at the primary node for each replica node, and is responsible for returning the non-replicated operations to the replica node in chunks.

2.6.1. Replication Process

1. The Pull model at the replica node requests its last operation ID from the Oplog.
2. The Oplog returns the latest operation.
3. This operation ID is sent to the Corresponder, running on the primary node.
4. The Corresponder makes a chunk of operations, continuing the order and sends it back to replica.
5. The Pull Model Task receives the chunk.
6. The Pull Model Task iterates over the chunk and performs the operations individually.

![Figure 2.6: Replication process](image-url)
2.6.2. Behavior

There are multiple behaviors which are handled by NosDB accordingly:

1. **No Operations in Replica Node Oplog**  
   i. No roll back is required because it is a fresh shard with no invalid operations.  
   ii. The Pull Model Task will make a new operation carrying default values and send that to Corresponder.  
   iii. The Corresponder will check if the primary’s Oplog has overflown or not. If not, it will create the chunk from the first operation in it and send it back.

2. **Overflown Primary Oplog**  
The oldest operations are removed from the Oplog after it reaches its maximum size. In this case, the first ever operation in the Oplog has been removed, and thus cannot be retrieved. Hence, replication cannot be performed now, instead this replica node would go into the state ‘Within Shard State Transfer’.

3. **Simultaneous Replication and State Transfer**  
In such a case, the operations with pending buckets (for transfer to replica) are directly inserted into the operation log without being performed over the replica. The operations will be performed through state transfer once the bucket reaches the replica.

4. **Rollback Scenario**  
The operations will be rolled backed in the following use case:

Node-1 is the primary node but it goes down for a good amount of time such that Node-2 (a replica node in the shard) becomes primary. At the time when Node-1 went down, Node-2 was only 75% synchronized with Node-1.

Due to primary node down, the Distributor Database Engine starts routing the user operations to Node-2, which means Node-2 will eventually have the latest operations.

After sometime, Node-1 joins the cluster once again, containing the 25% operations that Node-2 was unable to sync. These operations are now invalid, and need to be removed from the Oplog before Node-1 officially joins the cluster. This is to make sure that once replication takes place, Node-1 should only contain the 75% operations which were synced so that both nodes are synchronized completely. Thus, the 25% operations are removed from the Oplog and roll backed.

4.1. **Special case**  
In case the operations are greater than 64 MB, they will not rolled back – instead, the node will go into an “Intermediate” state where it will neither become primary, nor have any operations performed. In such a case, the node will have to be handled manually by the user.

5. **Primary Node Change**  
Consider the shard in the aforementioned example in the following cases:

- Once Node-2 becomes primary, replication stops with that node.
- If Node-2 does not become primary, it requests the other primary node for the operations.
6. Distribution Change

There are a few different scenarios which may cause the distribution to change e.g. Add or Remove Shard.

Even if the distribution changes, all operations are still applied on the replica regardless of whether their buckets are a part of this shard or not. This ensures data consistency.

2.7. Node Election Mechanism

In order for a primary node to be declared, the nodes undergo an election mechanism. The election mechanism is conducted in two phases:

- Speculative Phase

Three factors determine the liability of a node in being elected as the primary:

1. **Majority:** It is mandatory that out of the total configured nodes in a shard, the nodes connected to a node are in majority. This involves two considerations:
   - If the number of nodes is even, majority refers to greater or equal to configured nodes/2. For example, if 4 nodes are configured, at least 2 nodes must be connected to a node at that time.
   - If the number of nodes is odd, majority is greater or equal to (configured nodes + 1)/2. For example, if 5 nodes are configured, at least 3 nodes must be connected to a node at that time.

2. **Node Priority:** The user specifies the priority of a node, which may exist between the range 1 to 3 - 1 being the highest. If it is high, it is assumed that the user wants the node to be the primary node.

3. **Latest Oplog Entry:** The node containing the latest Oplog entry has a higher chance of being elected as the primary because it has the most updated data. The primary node also notifies the configuration server of its Oplog entry continuously in case it ever goes down.

The node “speculates” if any of the factors are in its favor, so that the election mechanism can be initiated.

- Authoritative Phase

On initialization of a shard, when there is just one node added, the node is considered primary by default. However, there will be no backup of the operations performed on it. Therefore once more nodes are added, the nodes compete on the aforementioned factors during an authoritative phase to become the primary.

The election process is then carried out as follows:

1. The node obtains an election ID from the configuration server and then locks it so other nodes may not nominate themselves while the current node’s election is in progress.

2. The node then broadcasts its information including the Election ID, its Oplog entry and priority to all the majority nodes, which check if their Oplog is more updated or if their priority is higher than the nominated node.

3. If both conditions are false for the majority nodes, they will return a vote in favor of the node.
It is necessary that all nodes participating in the election mechanism are in favor for a node to be elected as the primary.

However, the case aforementioned is the basic mechanism. There are two variants to the election mechanism based on the scenario:

1. **Take Over Election Mechanism**

   In case where a node (N2) with priority 2 is primary, and a node with priority 1 (N1) is added to the shard, the speculative phase will conclude that N1 should be made the primary. For that purpose, N1 will begin replicating the operations from N2 which will allow a margin window so that the pending operations are performed completely on N1. During the margin window duration, N2 will go into a read-only state. Once all operations have been replicated and the margin window has passed, N2 demotes itself and N1 "takes over" as the primary node.

   Three scenarios can trigger the Takeover Mechanism:
   - A node with higher priority is added to the shard.
   - The priority of an existing node is updated to a higher value.
   - A node with higher priority becomes up.

2. **Waiting Phase Election Mechanism**

   In case the primary node (N1) goes down, one of the majority nodes (N2) gets the Oplog entry from the configuration server and compares it against its own Oplog entry. If the Oplog entry at the configuration server is more updated than the nominating node, N2 will wait for N1 to become active again during a waiting phase. The waiting phase also serves to prevent data loss as there may be operations which have not been replicated before the primary went down.

   Once the waiting phase is over and N1 has not been up again, N2 will initiate the takeover election mechanism.
2.8. Split-Brain Handling

Split-brain is a cluster condition analogous to the biological Split-brain syndrome. This occurs when a node’s connection breaks with the configuration service or a connected node due to network failure. Since each side of the “split-brain” starts believing the other does not exist, it is highly critical if the condition occurs on a primary node as all write operations will be halted and updates to the R-nodes will not remain consistent. Thus, NosDB tackles the split-brain scenario from various perspectives:

1. Split-brain between primary and configuration service:

In such a case, the configuration service will not know if any primary exists and will consider the whole shard as a read-only shard.

NosDB handles this situation in the following ways:

- For a shard with **two** nodes:

  The primary demotes itself as soon as it loses connection with the configuration service. Once the primary is demoted, the node in connection with the configuration service will be able to elect a new primary.

- For a shard with **three or more** nodes:

  If the primary loses connection with the configuration service, it will retain its status until it no longer has a majority of the nodes connected to it.

2. Split-brain between primary and connected node:

In this scenario, the R-nodes will not know that a primary node exists to replicate the data from and will move on to elect a new primary if majority of the nodes are not connected to the primary node anymore.

In such a case, NosDB offers split-brain handling in the following ways:

- For a shard with **two** nodes:

  The primary will retain its status as long as it has a majority and the configuration service is connected to it.

- For a shard with **three or more** nodes:

  The primary will retain its status as long as it has a majority connected to it. Once the majority is lost, the node will step down from primary and elections will be conducted for a new primary. Regardless of the higher priority or latest Oplog entry of the downed node, the node conducting the new election will still be elected as the primary if it receives votes in its favor.

Split-brain handling allows minimum read-only time which ensures that the incoming write operations are not affected by the split-brain, sustaining reliability and fault tolerance as a result.
Data models define the way your data is stored and accessed. NosDB offers flexibility through varying data models which can fit your application requirements.

3.1. Embedded Data Model

NosDB offers the embedded model which allows embedding all related documents within one document. This eradicates the need of referring data from various collections. So instead of writing complex queries to resolve and gather data from referrals, related documents can be embedded to accumulate associated data in one place which can be accessed even with a single query. NosDB also offers infinite embedded support so that minimum querying yields required data.

An example of an embedded data model with the Products collection and multi-level embedding:

```
ProductDetails
{
   "key": "567164",
   "ProductName": "Milk",
   "Category": "Dairy",
   "Order":
   {
      "OrderDate": "2016/1/16",
      "OrderID": 20390,
      "Shipment": "23A",
      "ShippingCompany":
      {
         "SCName": "XYZ Logistics",
         "Contact": xxx-xxx-xxx-xx
      }
   }
}
```

Figure 3.1: Embedded Data Model
3.2. Non-Embedded Data Model

If there are documents where embedding will result in redundancy, or the data volume is very large, it is best to keep the collections separate. NosDB offers non-embedded data model for such cases to prevent any negative impact on performance because of the aforementioned reasons.

Moreover, any many-to-many relationship is suited in a non-embedded data model. For example, multiple students are enrolled in multiple courses. Therefore, mentioning the student details in each Course registered will result in redundancy. Hence, in such a scenario, adopting the non-embedded data model is optimal as a separate collection Students-Courses can be created which stores IDs of students against the Course IDs.

Figure 3.2: Non-Embedded Data Model
3.3. Hybrid Data Model

In cases where creating a separate collection may contain sparse data as in a one-to-many relationship, it is optimal if the related information of multiple collections is stored in a multi-valued field instead. This data model acts as a hybrid of embedded and non-embedded model as not all information of the collection is embedded within the other.

For example, a student can be a part of one society at maximum, or none at all – a society can have multiple students. Since the size of the society is limited because of the one-to-many relationship, the collection Societies can contain an array of the student IDs of the society members.

```
Students

{
  "_key": "567164",
  "StudentID": 342,
  "Name": "Alice Drew",
  "Contact": "xxx-xxx-xxx-x",
  "Grade": "A"
}
```

```
Societies

{
  "_key": "Society12",
  "Name": "Arts Society",
  "Mentor": "Ms. Elizabeth",
  "Enrolled Students": [342, 156, 343]
}
```

Figure 3.3: Hybrid Data Model
4. Client Operations

In This Chapter:

Write Operations
Read Operations

The client is liable to perform both read and write operations on NosDB, which behave accordingly to assist scalability and fault tolerance. The behaviors exhibited by the client operations as a whole include:

**Document-Level Operations**

A single operation works on document-level, that is, an operation will be performed on a single document. For example, an UPDATE query will modify the contents of a single document, inclusive of embedded documents.

**Collection-specific Bulks**

Though the operations are individually document-level, each bulk is collection-specific. Therefore, all of the cached operations for a specific collection will be accumulated in a single bulk to be sent to the query distributor for persistence.

**Transparent Distribution**

The distributor communicates between the client and the shards, so operations are distributed transparently in a scaled environment. Hence, any operation will be redirected at the distributor towards the specific shard (containing the required data) based on its particular distribution strategy.

4.1. Write Operations

Write operations essentially add or modify the documents in NosDB. Since write operations are mostly persisted to storage, it is beneficial to persist them in bulk, preventing network choke. Thus, the write operations in NosDB are first cached in-memory and then persisted to storage after reaching a specific bulk size. The write operations exhibit additional behaviors:

**Atomicity**

Though the write operations are sent in bulk, the operations’ performance is atomic - the operation will either be performed completely or not at all. This means that if the bulk has some operations that fail, the whole bulk will not be discarded – instead, the failed operations are simply logged.

**Asynchronous Backup**

As explained in Components – P-Node and R-nodes, the write operations are received by the Primary node. The replica nodes are read-only, so the operation is destined towards the Primary node. However, the replica nodes eventually replicate the operations through the Primary’s Oplog, enabling consistency and back up.
Document-Level Journaling

NosDB offers document-level journaling to prevent data loss in the event of a system crash if there are cached operations which have not yet been persisted. The modifications to the in-memory data by write operations are logged in the journal immediately, which are reflected on file storage eventually.

Critical Care in Indexing

Creating indexes and writing queries involving indexes requires critical dealing. This is because while there is significant advantage in read operations, write operations take longer to persist as they are performed on each index.

![Diagram of write operations execution path]

Figure 4.1: Write Operations execution path

4.1.1. Write Concern

As persistence of any operation to the file store is asynchronous, the client has the option of specifying the write-concern of the operation. This means that the client can receive the response after either of the events:

- **In-Memory Write Concern**
  The response is returned as soon as the operation is cached in-memory.

- **Journal Write Concern**
  The response is returned after the operation is persisted in the journal.

By default, the write-concern is set to in-memory, so that the client may carry on with the operations smoothly without having to wait for each response. However, the option can be changed according to the user’s preference to journal option which will return the response as required.
4.2. Read Operations

Read operations in NosDB include fetching stored data through API calls and querying. Techniques like Indexing boost read performance as there is no need to traverse the whole data set.

4.2.1. Read Preference

As explained in Components, all nodes allow reading of data but only the primary node can cater write operations. Hence, the client is given the liberty to specify how the data is read from any node according to the preference. Read-Preference is specified while fetching items through API or querying with ExecuteReader() statements which require reading from the database.

- Primary Only Read Preference

The read requests are only sent to the primary node when the latest data is required and as the primary nodes always contain the most updated data.
- **Load Balanced Read Preference**

  The load of requests is balanced using Round Robin strategy, where the read requests are balanced on client side among primary and replicas accordingly.

  ![Load Balanced Read Preference Diagram](image1)

  *Figure 4.4: Load Balanced Read Preference*

- **Nearest Read Preference**

  The read request is sent to the node which is "nearest" to the client, irrespective of its status as Primary or Replica. The "nearness" is determined on the basis of the Round Trip Time (RTT) of each node. Hence, for two nodes with RTT 2ms and 5ms respectively, the read request will be sent to the node with RTT 2ms.

  ![Nearest Read Preference Diagram](image2)

  *Figure 4.5: Nearest Read Preference*
5. Storage

In This Chapter:
- In-memory Cache
- LMDB Storage Provider

NoDB's scalable nature requires that the data is stored such that constant transaction and distribution is catered. This is why it has to guarantee that the adding, updating and fetching of data into and from the storage is time and space efficient. Thus, NoDB persists the data to physical storage in a file storage system to avoid any data loss in the event of a system crash. However, frequently used data is stored in-memory to prevent accessing the disk for each read.

![Storage steps in persistence](image)

**Figure 5.1. Storage steps in persistence**

5.1. In-Memory Cache

Each NoDB node contains a dedicated in-memory cache to handle any transaction load, which is especially vital with ad-hoc querying and large amount of live, distributed data. Instead of making excessive trips to access this data from the underlying storage, it is efficient if frequently used data is stored in-memory, allowing faster operations to the client as the operations are asynchronously persisted in the background.

Hence, NoDB follows the sequence of caching the client operation first in memory, and then sending it to the physical file storage.
5.2. **LMDB Storage Provider**

The storage manager contains the storage provider, through which the actual file writing process takes place. The storage provider facilitates the manager in storing the data; hence NosDB utilizes **Lightning Memory-Mapped Database (LMDB)** as its third party storage provider.

LMDB by Symas is a high performance, embeddable, transactional database which stores data in the form of key-values. LMDB has been created for the OpenLDAP Project, which is a lightweight module of the Lightweight Directory Access Protocol (LDAP). It uses memory-mapped files, so it has the “read performance of a pure in-memory database while still offering the persistence of standard disk-based databases, and is only limited to the size of the virtual address space”.

Key features as detailed on the [Symas website](https://symas.com/lmdb/) include:

- **Ordered-map interface** (keys are always sorted, supports range lookups)
- **Fully transactional, full ACID semantics with MVCC.**
- **Reader/writer transactions**: readers don’t block writers and writers don’t block readers. Writers are fully serialized, so writes are always deadlock-free.
- **Read transactions are extremely cheap, and can be performed using no mallocs or any other blocking calls.**
- **Memory-mapped, allowing for zero-copy lookup and iteration.**
- **Maintenance-free, no external process or background cleanup/compaction required.**
- **Crash-proof, no logs or crash recovery procedures required.**
- **No application-level caching. LMDB fully exploits the operating system’s buffer cache.**
- **32KB of object code and 6KLOC of C.**
6. Backup and Restore

In This Chapter:

Backup
Restore
Roles of NosDB Components

In a distributed environment as that of NosDB, it is essential to provide backup and restore operations to ensure fault-tolerance and protection of data in case of unexpected system failure.

NosDB provides database-level backup and restore operations on running nodes, i.e., moment-in-time snapshots of the data will be taken for any operations being performed during the backup process. This will ensure data consistency while restoring the database. An atomic backup or restore operation is performed against a clustered/stand-alone database.

Backing up data should be made a routine process so that data is restored with least inconsistency.

6.1. Backup

Backup ensures that a copy of the data is preserved so that it can be restored in the event of a system failure. Moreover, backups can be made to store a copy of the source database to a new location on either configuration:

- **Distributed**: Creates backup of each shard to a specified location on the primary node of the respective shard. For example, D:\\database\\NosDB\backups.

  The location specified for backup must be valid for all primary nodes of each shard.

- **Consolidated**: Creates a backup of all the shards sequentially on a shared location. For example, //server1/backups.

  Make sure that NosDB Database service (NosDBSvc) and NosDB Configuration service (NosConfSvc) have write access on the location.

NosDB offers two levels of backup jobs:

6.1.1. Full backup

A full backup, as the name implies, will store the whole data and all components associated with it to either a local or shared location. A full backup copies the following:

- Configuration meta-info about the clustered or stand-alone database like collections, indexes, storage provider from the configuration server. The meta-info aids in creating a database at any location with same or a different name.
• The Oplog for the operations being performed during the backup process. Since NosDB backup can be performed without going offline, the Oplog backup enables a moment-in-time data snapshot of the operations to prevent any inconsistency in restoration.

If an operation which triggers state transfer (e.g., adding/removing shard) is performed during backup, the job would be cancelled. Thus make sure that no state transfer occurs while the backup is in progress.

• Actual data from the source databases against the shards. However, for server side deployments and attachments, only the configuration information will be backed up – not the actual data.

6.1.2. Differential backup

Since full backup is an expensive process for huge amounts of data, NosDB offers differential backup which only backs up the modifications in the database that have occurred since the last full backup. This is particularly useful if regular backups are made, as the data, time and network cost involved is much lesser compared to a full backup. The backup is performed in a cumulative manner - the changes are backed up as they keep on occurring at the database.

A differential backup copies the following:

• The Oplog of any operations being performed during the backup process. The operations are tracked in a dedicated log (DIFLog) which stores the Oplog operations being performed against each shard since the last full backup.

If a new shard is added to a configuration against which a DIFLog is being created, the tracking would be cancelled.

• Actual data from the shards that has been changed because of add, update and delete operations.

![Figure 6.1. Backup](image)
Referring to Figure 6.1, the differential backup made on T₁ will copy all modifications made since the full backup at T₁. The backup made on T₂ will copy all changes made since T₁, T₃ will store all changes made since T₁, and so on.

Once the next full backup is made at T₂, the whole data will be copied along with the configuration details and Oplog, which will include the changes from the differential backup.

6.2. Restore

Restoration is a multi-phase process that recreates a database from the provided backup. Just like backup, NosDB offers two levels of restoration jobs: Full and Differential. This implies that the restore type should correspond to the type of backup you have performed.

For example, a Differential Restore will only be performed on a database backed up through the Differential option. This is because the DIFLog is used during the Differential Restore, and a Full Backup does not create the DIFLog.

6.2.1. Full Restore

In a full restore, the entire database is reconstructed from the provided backup.

A full restore must be provided with a database that has been backed up through full backup.

During a full restore, NosDB:

i. Configures the destination database according to the configuration of the provided backup in the config server meta-info. The provided configuration must be consistent with the one existing during the backup phase. These configurations include:
   • Number of shards
   • Shard names
   However, number of nodes in those shards and participating nodes (IPs) can be changed.

Consistency in configuration is mandatory because the Oplog operations are applied to recreate a moment-in-time snapshot of the source data. Hence, in case of any inconsistent configuration, NosDB will wipe out the existing data and restore the database with the new configuration.

ii. Recreates data and indexes through the backed up database against all participating shards in the database.

iii. Performs any operations in the Oplog that were performed during the backup window, ensuring consistency.
6.2.2. Differential Restore

As full restoration can prove a heavily expensive operation if performed regularly, NosDB offers differential restoration which only restores the modifications that have occurred since the last full restoration.

A differential restore must be provided with a database that has been backed up through the differential backup option.

Restoration is a comparatively costly process because of storage limitations, and the reconstruction of an entire database and index creation. Thus, in case a database exists with restored data, NosDB will apply differential restoration of the same database to the existing one.

A database will not be restored with an existing name.

In case the database does not exist, NosDB performs a full backup then performs differential restore against the database.

6.3. Roles of NosDB Components

The following components are responsible for carrying out the backup and restore operations:

Configuration Server

- Responsible for book-keeping against an operation.
- Services all client requests
- Saves the following information:
  - Database configuration
  - Collection distribution strategy, bucket information
  - Index information

Shards

Performs all data and Oplog backups against a particular shard. The data is in the form of a JSON document.

For more detail on creating backup and restoring databases, please refer to DDL Statements in Programmers’ Guide and Backup and Restore in Administrators’ Guide.
A distributed environment means granting access to many different users, where any user can change the cluster configuration, perform management operations or access data without any restriction. The concern of cluster security arises when the data is critical or you do not want everyone to have administrative access rights to your server except for the list of authorized users.

NosDB provides a customizable security feature for securing distributed database access. Any unwanted or unauthorized access to the NosDB cluster and to the data that resides in the cluster can be avoided by simply configuring NosDB security. Moreover, the data being transmitted over network between servers and the application is secured by configuring NosDB data encryption mechanism.

Using a database system securely requires database administrators to take precautionary security measures on their distributed system. These measures involve two steps - authentication and authorization of the user who will be accessing the database system.

### 7.1. Authentication

Authentication includes the process of validating if a user has access to the system on the basis of an authentication mechanism. This is the first step in ensuring security in NosDB.

NosDB provides two types of authentication mechanisms:

**Windows Authentication**

This includes a challenge-response based model involving Microsoft Active Directory for the users of a local system, domain or workgroup. If a user belongs to the domain and a registered login of NosDB, he/she is authenticated to access NosDB. The following factors should be kept in mind:

- For a local machine account in **domain** environment, the client, configuration service and database service must be running on the same node.
- For a local machine account in **workgroup** environment, the account must be registered.

**NosDB Authentication**

This is similar to the SQL Server Authentication in SQL Server, which consists of a custom username-password authentication model. The credentials for NosDB authentication will be either passed through the connection string, or as parameters of the API and tools in an encrypted format. By default, the user **admin** is created, of which the password is specified during installation.

### 7.2. Authorization

Authorization includes the process of verifying if the authenticated user can perform a specific operation on any of the database system resources. For this purpose, the database administrator assigns some roles to the database
users. These roles further define if the user has rights to perform an operation on that specific resource; for example if "db_datareader" is assigned to a user for a database, that user can only read from that database.

To understand the concept of authorization in more detail, we will first divide our database system:

### 7.2.1. Resources

A resource refers to any single entity of the database system. For example, a database or a collection.

These resources are categorized further in NosDB as follows:

<table>
<thead>
<tr>
<th>Administrative Resources</th>
<th>Application Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>• User</td>
<td>• Collection</td>
</tr>
<tr>
<td>• Database</td>
<td>• Document</td>
</tr>
<tr>
<td>• Cluster</td>
<td>• Attachment</td>
</tr>
</tbody>
</table>

#### 7.2.2. Roles

A role is a set of operations that can be assigned to a user to define his/her responsibilities. In other words, a user is created once a role has been granted to a registered login in NosDB. An operation is the utilization of a resource to demonstrate certain behaviors by the user. NosDB provides a set of built-in roles with varying levels of granularity:

##### 7.2.2.1. Database Roles

<table>
<thead>
<tr>
<th>Database Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_datareader</td>
</tr>
<tr>
<td>A user with the <code>db_datareader</code> role can perform only read operations on the database. The <code>db_datareader</code> role is owned by <code>db_user</code>.</td>
</tr>
<tr>
<td>db_datawriter</td>
</tr>
<tr>
<td>A user with the <code>db_datawriter</code> role can perform only write (INSERT/UPDATE/DELETE) operations on the database. The <code>db_datawriter</code> role is owned by <code>db_user</code>.</td>
</tr>
<tr>
<td>db_user</td>
</tr>
<tr>
<td>A user with the <code>db_user</code> role can perform read and write operations on the database.</td>
</tr>
<tr>
<td>db_admin</td>
</tr>
<tr>
<td>A user with the <code>db_admin</code> role can CREATE, DROP or ALTER a collection/index/stored procedure/CLR function/CLR trigger in NosDB. In addition, a db_admin can also GRANT and REVOKE roles to/from a user on the database.</td>
</tr>
</tbody>
</table>
### 7.2.2.2. Cluster Roles

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_creator</td>
<td>A user with the <code>db_creator</code> role can perform any data definition (DDL) operation on the databases, like CREATE, DROP or ALTER databases over the cluster.</td>
</tr>
<tr>
<td>cluster_manager</td>
<td>A user with the <code>cluster_manager</code> role can perform start/stop operations on the shards and nodes of the cluster.</td>
</tr>
<tr>
<td>cluster_admin</td>
<td>A user with the <code>cluster_admin</code> role can GRANT and REVOKE roles over the cluster and can perform managerial operations on the cluster like adding and removing the shards and nodes of the cluster.</td>
</tr>
</tbody>
</table>

### 7.2.2.3. Server Roles

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security_admin</td>
<td>A user with the <code>security_admin</code> role can perform any data definition (DDL) operation on the users, like CREATE, DROP or ALTER users.</td>
</tr>
<tr>
<td>sys_admin</td>
<td>A user with the <code>sys_admin</code> role can perform any operation on the NosDB server, including cluster and database managerial operations.</td>
</tr>
<tr>
<td>distributor</td>
<td>The distributor is a special role, used only for the distributor service. A distributor is only authorized to distribute client operations.</td>
</tr>
</tbody>
</table>

### 7.2.2.4. Custom Roles

NosDB provides the flexibility of defining custom roles for users which grant a customized set of varying operation permissions to a user. Custom roles have database level scope, meaning the custom role can only be created on or can have permissions on databases only.

The process of creating and granting custom roles requires the following privileges:

1. The custom role is created by the `cluster_admin`.

2. Once created, permissions can now be granted to the role. The `db_security_admin`, `db_owner` and `db_admin` have the privilege to grant permissions to the role. These permissions can be selected from the following permission set:

<table>
<thead>
<tr>
<th>Permissions</th>
<th>DropDatabase</th>
<th>AlterDatabase</th>
<th>CreateCollection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DropCollection</td>
<td>AlterCollection</td>
<td>CreateIndex</td>
</tr>
<tr>
<td></td>
<td>DropIndex</td>
<td>CreateUserDefinedFunction</td>
<td>DropUserDefinedFunction</td>
</tr>
</tbody>
</table>
3. Once the role has been granted with the permissions, the `db_securityadmin`, `db_owner` and `db_admin` can grant the role to an existing user in NosDB. The user now has permissions to perform the operations on the specified database.

A custom role can contain an **empty** permission set and be granted to a user. Once any permissions are granted to the role, they will be granted to the specified user.

### 7.2.3. Permissions

Permissions authorize users to perform the actions allotted within their respective roles. The Cluster Admin creates the custom roles by granting permissions to certain actions.

### 7.2.4. Role-Action Chart

<table>
<thead>
<tr>
<th>Database Roles</th>
<th>Cluster Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>db_datareader</strong></td>
<td>dbcreator</td>
</tr>
<tr>
<td><strong>db_datareader</strong></td>
<td>dbcreator</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>1. [All db_user Actions]</td>
</tr>
<tr>
<td></td>
<td>2. CREATE, DROP, ALTER database</td>
</tr>
<tr>
<td></td>
<td>3. CREATE DRO...</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>4. CREATE, DROP, ALTER collection</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>5. CREATE, DROP, ALTER trigger</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>6. CREATE and DROP index</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>7. CREATE and DROP CLR functions</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>8. CREATE and DROP database roles</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>9. GRANT and REVOKE database roles</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td><strong>clustermanager</strong></td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>1. [All db_admin Actions]</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>2. DROP databases</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td><strong>clustermanager</strong></td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>1. [All db_creator Actions]</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>2. CREATE, DROP, ALTER databases</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>3. CREATE and DROP database roles</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>4. START and STOP shard</td>
</tr>
<tr>
<td><code>db_datareader</code></td>
<td>5. START and STOP node</td>
</tr>
<tr>
<td>Role</td>
<td>Permissions</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| clusteradmin | 1. [All clustermanager Actions]  
|            | 2. Create and remove cluster  
|            | 3. GRANT and REVOKE cluster roles  
|            | 4. Add and remove shard  
|            | 5. Add and remove node |
| securityadmin | 1. [All clusteradmin Actions]  
|             | 2. CREATE, DROP, ALTER cluster users |
| sysadmin    | All actions                                                               |
| distributor | Distribute client operations.                                              |
8. Encryption

In This Chapter:

Encryption Process

Encryption makes data undecipherable using a key and password to render it useless without that key. Hence NosDB provides encryption to enhance the security of data; any data added to the database collection will be encrypted, and will be of no use unless the key is available to decrypt.

NosDB encryption implements the Data Encryption Key (DEK) and Master Key model. The DEK is determined through the encryption provider specified by the user. The encryption provider is an encryption standard, either 3DES or AES. This DEK is used to encrypt the data before it is written to the store.

Moreover, further security is established by encrypting the Data Encryption Keys through the master key. To encrypt/decrypt the DEKs, the user creates a master key for NosDB by specifying a password for the master key. Using that password, NosDB gets a master key from DPAPI (Data Protection Application Programming Interface) and stores it with the DEK in encrypted form in the configuration store to be later used to encrypt and decrypt the Data Encryption Keys. The master key is encrypted through DPAPI as well.

NosDB provides the flexibility to choose from any of the built-in encryption providers to configure encryption at database level:

- 3DES_128
- 3DES_192
- AES_128
- AES_192
- AES_256

Encryption can be configured and enabled at runtime. However, the encryption provider cannot be reconfigured once it is specified. Note that encryption is configured at the database level and enabled at a collection level. This means while the provider is configured on the database, encryption will need to be enabled individually on each desired collection.

Any existing data in the collection will not be encrypted unless it is updated after encryption is enabled. Any data added after encryption is enabled will be encrypted by default.

NosDB supports document level encryption - complete document is encrypted.

8.1. Encryption Process

1. Using DDL query, the master key is created from a user specified password.
2. A Database Encryption Key is automatically generated when the encryption provider is configured using NosDB Management Studio or DDL query. This DEK will be used to encrypt the data, while the master key encrypts the key.
3. Once the encryption provider is configured, encryption can be enabled and disabled on the collection.
9. Data Expiration

In This Chapter:
Expiration Properties

Enormous datasets of rapidly increasing volume often contain data that can become void or stale after a period of time. Such nature of data requires that it is expired on a regular basis to enhance transactions of the database with relevant data. For example, expiration aids in maintaining sessions. A session can be expired by setting expiration on a DateTime-specific field like sessionCreated. If the difference between current time and sessionCreated is greater than the session duration, the session can be expired.

Hence, incorporating data expiration in a large scale NoSQL database like NosDB results in optimized performance on account of the following factors:

- Network traffic is lowered as transaction load is specific to a certain time duration. For example, let’s suppose new logs are generated with the start of any activity. There is no need to fetch the logs from the previous activity, thus they can be expired.
- Memory consumption is regulated as the stale data is removed periodically. For example, expiration can be enabled for a system set to record video for a 5 hours duration. NosDB will then automatically remove the data which was recorded more than 5 hours ago, freeing memory for the next batch.
- Eliminates need to write code for carrying out expiration, as NosDB offers simple GUI and DDL querying based expiration management.

Expiration Interval vs. Cleanup Interval

Note that there is difference in expiration of data and actual deletion (“clean up”) of the data from the database. NosDB provides expiration interval at collection level and cleanup interval at database level. An index is maintained internally on the specified field to keep track of the documents that need to be expired after every expiration interval. Once expired, a background thread executes after a configurable interval (default is 60 seconds) that deletes expired documents. This is the cleanup interval. There may be a delay between when a document is expired and when it is actually removed from the database.

Cleanup Interval > Expiration Interval

While a document might be expired, there is possibility that it has not been removed from the database (cleanup interval is greater than expiration interval). For example, expiration interval is set to 30 seconds, while the cleanup interval is 60 seconds. Hence, 30 seconds after insertion of the document, it gets expired from the collection but still hasn’t been cleaned up from the database. Therefore, it can still entertain retrieval operations till cleanup interval is completed. Note that this also means that the lifespan of the document is actually [Expiration Interval + Remaining Cleanup Interval] in the database.

Expiration Interval > Cleanup Interval

Similarly, if expiration interval is 60 seconds and cleanup interval 30 seconds, the deletion of data from the database will only take place after the cleanup interval which succeeds the expiration interval. This can also result in an overlap, as shown in Figure 1.
9.1. **Expiration Properties**

NosDB expiration has the following properties:

9.1.1. **Expiration Types**
   - *Interval Based:* A time duration in minutes/seconds is specified while enabling expiration. The document will automatically expire once the specified time has elapsed since the indexed field value.
   - *Absolute Time:* Upon specifying expiration interval as 0, the expiration strategy compares field value with the current time and expires the document accordingly.

9.1.2. **Document Specific**
   - If a document does not contain the specified field, the document will never expire.
   - If a specified field is an array of date field, the item will be expired based on the lowest value in array.

9.1.3. **Execution Preference**
   - Expiration of documents will only take place on the primary node of a shard.
   - Expiration is performed by a background thread.

9.1.4. **Monitoring**
   Number of documents expired can be monitored from a PerfMon counter "Expiration Count".

9.1.5. **Limitations**
   - Expiration strategy cannot be configured on capped collections.
   - If the indexed field is not a date field, the document will never expire.
   - Expiration strategy can only be created on date fields (value of these fields must match a certain DateTime format). These are single field indexes.
## Supported DateTime Formats

<table>
<thead>
<tr>
<th>Supported Format</th>
<th>Equivalent Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>M/d/yyyy h:mm:ss tt</td>
<td>M/dd/yyyy hh:mm</td>
</tr>
<tr>
<td>M/d/yyyy h:mm tt</td>
<td>dd-MM-yy</td>
</tr>
<tr>
<td>MM/dd/yyyy hh:mm:ss</td>
<td>dd/mm/yyyy</td>
</tr>
<tr>
<td>M/d/yyyy h:mm:ss</td>
<td>dd/MM/yyyy HH:mm:ss</td>
</tr>
<tr>
<td>M/d/yyyy hh:mm tt</td>
<td>yyyy-MM-dd</td>
</tr>
<tr>
<td>M/d/yyyy hh tt</td>
<td>yyyy-MM-dd HH:mm:ss</td>
</tr>
<tr>
<td>M/d/yyyy h:mm</td>
<td>yyyy-MM-ddTHH:mm:ssK</td>
</tr>
<tr>
<td>MM/dd/yyyy hh:mm</td>
<td>yyyy-MM-ddTHH:mm:sszzz</td>
</tr>
<tr>
<td>dd-MM-yyyy</td>
<td></td>
</tr>
</tbody>
</table>
10. Indexing

In This Chapter:

Index Creation
Index Types
Index Behaviors

Indexing is a universal method to enhance search and retrieval of any item within a collection of items. This can be illustrated by the example where a database contains terabytes of items, and the client requests to update the department of all employees of the Engineering department to Technology. If the data has not been indexed, the system steps into the dangerous terrain of traversing the whole dataset, which is a huge blow to the performance. Moreover, since most systems are distributed nowadays, there is an additional overhead of traveling over the network to search each distributed data collection for the respective Employee records. In the worst case, there may be just one record of the Engineering department, that too lying at the last location of the traversal – an unnecessary wastage of resources.

However, if the data has been indexed on Department, it takes just one query on the respective field to extract the values from their particular locations in one hop. There is no need to traverse the whole dataset, and scalability boosts the purpose of indexing as it further partitions the data.

10.1. Index Creation

A NosDB index is created in the form of a B tree, which contains the document keys of the indexed documents against the index key in an ordered fashion. This makes resolving queries with comparisons like “Age > 20” very simple, as all records stored after Age = 20 are understandably greater than 20. A single index can cater all of the following data structures together in itself:

- String
- Boolean
- Null
- Number
- DateTime
- Array
- Object of a custom class

Because of the various data structures being catered, indices in NosDB are very fine grained. All formats are supported in comparisons. However, the custom objects only support equality comparison.

NosDB indexes multiple data types on a single index by following a type comparison rule. What this implies is that a type is considered greater or lesser than the other. So if the index is queried as follows:

```java
int value <attribute> string value
```

The operation will be valid and performed.
The indexes are initialized upon an index creation request from the user but remain non-functional until the index is completely populated. The index population occurs in background.

10.2. Index Types

10.2.1. Single Attribute Indices

NosDB allows the user to index any field in a document. This is useful if you have certain frequent queries and fields to apply the operations on.

For example, if a telecommunication company requires the total incoming call duration of each user at the end of each day for analysis, indexing on the field *IncomingCallDuration* will ease access to the respective data because unnecessary traversal is not required.

```json
{
    "UserID": "12345",
    "FirstName": "Mary",
    "LastName": "Evans",
    "Subscription": "Post-Paid",
    "IncomingCallDuration": "20:34:04",
    ...
}
```

10.2.1.1. Embedded Indices

If the data is in the form of JSON documents, indices can be defined within the embedded documents as well. The fields of the embedded documents will be internally accessed through dot notation.

For example, if *Customer* documents are stored with an embedded document of *Address* for each employee, and operations are to be performed on the records of customers of a specific state, the index will be created on *Address.State*.

```json
{
    "UserID": "12345",
    "FirstName": "Mary",
    "LastName": "Evans",
    "Age": 26,
    "MaritalStatus": "Single",
    "Transport": "None",
    "Address": {
        "Street": "999 St",
        "Town": "Houston",
        "State": "TX"
    }
}
```
10.2.2. Compound Indices

In order to further make the indices more specific, NosDB provides the feature of compound indexing. This means that you can create composite indices of multiple fields of the document.

For example, in the following sample document, if a firm wants to analyze the records of married customers above the age of 40 who travel by car, an index can be defined on the attributes MaritalStatus, Transport and Age. Additionally, the sequence of the attributes in the index can be switched according to the user’s preference.

```
{  
  "CustID": "cust_123",  
  "FirstName": "Joe",  
  "LastName": "Drew",  
  "MaritalStatus": "Married",  
  "Transport": "Car",  
  "Age": 45,  
  "Address": {...}  
}
```

Note that only a single array field is allowed in Compound index.

The index also provides the ability to query on prefixes. For example, the MaritalStatus, Transport and Age index created above can also be utilized for querying on following combinations of attributes:

- MaritalStatus
- MaritalStatus and Transport
- MaritalStatus, Transport and Age

However, the prefixes follow the ‘previous must’ pattern, i.e. the index cannot be used to query on attributes if their preceding attributes (prefixes) are not included. So querying on the following combinations will be delegated to full store scan:

- Transport
- Age
- Transport and Age
- MaritalStatus and Age

![Figure 8.1: Permissible and non-permissible prefixes in indexes.](image)
10.2.2.1. Embedded Indices

Compound indices can also be defined with embedded fields. For example, if a firm wants to analyze the records of customers of Texas older than 25 years, an index can be defined on the attributes Age and Address.State.

```
{
    "CustID": "cust_345",
    "FirstName": "Mary",
    "LastName": "Evans",
    "MaritalStatus": "Married",
    "Transport": "None",
    "Age": 26,
    "Address": {
        "Street": "999 St",
        "Town": "Houston",
        "State": "TX",
        "ZipCode": "77002"
    }
}
```

10.3. Index Behaviors

10.3.1. Sparse Indices

Since NosDB is schema-less, each document contains varying fields and formats. Thus, all of the aforementioned index types are sparse by default - they will only store those documents which contain the indexed field and will overlook the rest of the documents. This ensures the index is compact and precise; there is no point in directing a key towards an empty field if it does not exist.

For example, if one document contains the indexed field Age and the next document does not, the next document will not be indexed.

A null value is considered a value and will be indexed.

10.3.2. Caching

As explained in Chapter 2: Components, the index contains a cache to store the frequently used indices instead of fetching them from the file storage. This makes searching of the relevant items even faster as the index is in-memory. NosDB provides three configurable modes for caching the indices:

1. **All**: All of the index data stored on disk will be cached in memory. This impacts performance if majority of the indices are not even being used.

2. **Recent**: Stores all of the recently used data from the index. On the execution of the query, the index first checks the index cache for the required data. If it does not exist there, it will be loaded in-memory from
the file storage and reside in the cache unless internal eviction takes place according to LRU policy within
the cache.

3. **None**: None of the data in the index will be cached, and every time a query is executed the data in the
index will have to be accessed through the file storage. It is useful if there is memory limitation or such
queries are rare which require index resolving.

### 10.3.3. Journaling

Each index logs its operations in a journal which are persisted in the commit cycle. If journal option is enabled, the
log will be persisted on disk, else it will be stored in system memory. The journal logs the index activity like
insertion, updation and removal.
11. CLR Triggers

_In This Chapter:_

Registering and Configuring Triggers

Trigger Behavior

Triggers are functions registered against a database operation and are executed in response as soon as the operation is performed, hence _triggering_ the function. NosDB supports collection-level triggers to perform client-defined actions prior to or after an operation has been performed on the collection. Each collection can only have a single trigger, however the trigger can contain multiple actions.

These actions include the operations which essentially modify the collection content, in other words, triggers can be registered for any DML (data manipulation language) action through NosDB. The actions for which a trigger can be fired in NosDB include:

<table>
<thead>
<tr>
<th>Pre-Operation</th>
<th>Post-Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Insert</td>
<td>Post-Insert</td>
</tr>
<tr>
<td>Pre-Update</td>
<td>Post-Update</td>
</tr>
<tr>
<td>Pre-Delete</td>
<td>Post-Delete</td>
</tr>
</tbody>
</table>

All _Pre_ triggers are executed before the document is cached. This means that the operation will only be performed once the trigger returns true. This helps to validate the document before an action, saving precious CPU cycles. _Post_ triggers are performed once the operation has been journaled.

Triggers are useful for applications which depend on certain data items and their modification needs to be responded to.

For example, the count of _GoldCustomers_ has to be updated once a customer is added to the _CustomerDetails_ collection of a banking system. The client can create a trigger on the action _PostInsert_ which will increment the count – confirming that insertion has taken place.

Another scenario can include the _PreInsert_ operation where before a customer’s document is inserted, the format of _DateOfBirth_ is verified to avoid any exception occurring during insertion of the document.

Due to the modifying nature of the actions, the modification to the collection takes place at document level, except for _PreUpdate_, where only the attributes to be updated within a document are specified. Apart from _PreUpdate_, the rest of the actions are performed against the whole document.
11.1. Registering and Configuring Triggers

To configure triggers on NosDB, IDatabaseTrigger interface provided by NosDB needs to be implemented and then deployed through NosDB Management Studio. NosDB will use that custom provider to register triggers against the database. Using the interface, the client provides the trigger action and its corresponding JSON document in an instance of the TriggerContext class to the PreTrigger() or PostTrigger() methods of IDatabaseTrigger.

For more detail on registering triggers, please refer to the section on CLR Triggers in Programmer's Guide.

11.2. Trigger Behavior

11.2.1. Failure conditions

Triggers may fail to execute if:

- An exception is thrown during execution of the client’s implementation against the trigger action.
- PreTrigger() returns false.

The failed triggers are logged in DatabaseHost.log located at %INSTALL_DIR%\logs.

11.2.2. Updating Triggers

- Updating trigger implementation

Since the assembly for the implementation is deployed through NosDB Management Studio, changing the implementation is not possible unless the trigger is dropped and a new trigger is created with the updated implementation.

- Updating enabled actions

Trigger actions can be updated through NosDB Management Studio by checking/unchecking the checkboxes for the actions. Note that if implementation against an enabled action does not exist, the trigger will not be fired.
12. Querying in NosDB

In This Chapter:
- Extended SQL Properties
- Functions
- Query Syntax

In any large scale, schema-less data store, retrieving data based on a primary key is not feasible because each data item contains its own schema. This elicits the need for indexing on various other fields of the database to ease the searching of data. Retrieving these items requires effective querying of the data, such that the search is relative to the attribute. For example, a company requires to fetch the items which have UnitsInStock lesser than 20. Hence, NosDB allows searching and modifying the result set according to the query criteria. This means that the database can be updated through queries at runtime when needed. NosDB requires all searchable attributes to be indexed before using those indexes.

Since NosDB accommodates unstructured data and supports JSON documents and arrays as field values, the columns of the database can be multi-valued. Compared to the relational database where a single value is stored against a single column, NosDB fully supports the embedded data model. Thus, NosDB extends the standard SQL in order to cater the additional flexibility it provides.

12.1. Extended SQL Properties

The extended SQL operators have been created while keeping in mind the scalability and structure of NosDB. Thus, they have the following assisting properties:

12.1.1. Infinite Embedded Support

As NosDB supports embedded JSON documents to an infinite number, it can fetch the attributes from any level of the embedded document with dot notation, just as it supports indexing on the embedded documents.

For example,
{
    "ProductID": "DA123",
    "ProductName": "milk",
    "Order":
    {
        "OrderDate": "2016/1/16",
        "OrderID": 20390,
        "Shipment": "23A",
        "ShippingCompany":
        {
            "SCName": "XYZ Logistics", "Contact": "xxx-xxx-xxx-x"
        }
    }
}
Querying on Products.Order.ShippingCompany.Contact is permissible. However, the greater the level of embedded documents, the more it will impact performance because of increased complexity. It is advisable if the level of documents is kept up to 3-4 levels in order to achieve optimum performance.

12.1.2. Infinite Array Support

Just like embedded documents, each index of an array can further contain arrays up to an infinite level. A simplified sample document containing arrays within arrays would be as follows:

```json
{
   "ProductID": "DA123",
   "ProductName": "Milk",
   "PackageSize": [S, M, L],
   "Types": ["Skimmed", "Full Fat"],
   "TypeBySizeAvailable": [[12, 5], [4, 17], [21, 21]]
}
```

12.1.3. Wild Card Searching

Since the fields can also contain string values, NosDB provides the flexibility of indexing fields with specific patterns on the string itself, i.e., the query includes wildcard searching. A wildcard allows you to search for strings starting with/ending with/containing a specified pattern of character(s).Wildcard searching is particularly useful in ad-hoc queries and unstructured data, because the data is not very definitive.

For example, if a social networking site wants to sort the comments on posts of FIFA according to relevance, it can create a query to return the Comment fields containing the wildcards "%foot%", "GER%", "%FA".

```json
{
   "CommentID": "Comm_123", "UserID": "footie123",
   "CommentSection":
   {
      "Comment": "GERMANY ONCE AGAIN!",
      "Time": "21:09", "Likes": 9
   }
}
{
   "CommentID": "Comm_367", "UserID": "rayisaac",
   "CommentSection":
   {
      "Comment": "FIFA epidemic breaks out. Hide yo' wives",
      "Time": "21:14", "Likes": 14
   }
}
{
   "CommentID": "Comm_454", "UserID": "imaspammer",
   "CommentSection":
   {
      "Comment": "Meanwhile, a wild rabbit appears. . .",
      "Time": "22:37", "Likes": 0
   }
}
```
12.1.4. Parametrized Queries

In any distributed system, sending queries with attributes and values over the network is risky, as it can be easily accessed. Thus, the values and attributes are parameterized just like SQL to avoid decryption of values.

Parameterizing also aids in generalizing the stored procedures and user defined functions, as the values are provided against the parameters on runtime as constant values.

The parameters are defined in a key-value manner, as @name = ‘Joe’. There is no syntax change in a parametrized query; except that it contains the placeholder instead of the attribute, “WHERE FirstName = @name” which makes the query useless lest it is sniffed over the network.

12.2. Functions

NosDB supports integration of functions in querying to further enhance querying in the database. Two categories of functions are provided by NosDB:

12.2.1. Built-in Functions

NosDB offers a variety of built-in functions, including the standard SQL Aggregate and Scalar functions.

- Aggregate functions return an aggregated single result after executing the query on the filtered data set. These include COUNT, SUM, MAX, MIN, and AVG.

- Scalar functions return a single value calculated against each record. These include NOW, UCASE, LCASE, LEN, ROUND, MID and FORMAT. However, SQL is relational-database-friendly so it does not cater to multi-value fields like arrays and JSON documents. Thus, additional querying operators are provided by NosDB like CONTAINS, ANY, CONTAINS ALL, ARRAY SIZE and EXISTS.

For the whole list of built-in functions provided by NosDB, please refer to Query Syntax in NosDB.

12.2.2. CLR Functions

Apart from the vast range of built-in functions, users are also provided with the flexibility to provide their own custom logic to perform calculations which may be beyond the scope of built-in functions. These user-defined functions (UDFs) are collection-level routines which can be used in SELECT queries just like built-in functions, and are particularly useful in cases where a certain calculation is frequently required to be performed.

12.2.2.1. Supported Query Types

The UDFs can be used in two scenarios for querying – projection and comparison.

Consider the following sample of the collection Customers:

```json
{    "_key": "Cust_123",    "CustomerID": 1234,    "Name": "Mary Evans",    "Address": {"Street": "999 St", "State": "TX"} }
```
• Projection

For example, if a firm needs to analyze the churn rate of its customers of Texas to predict the performance of the company, calculateChurn() can be defined which performs the calculations according to the formula after filtering the customers of Texas. Thus, the resulting query will be executed using calculateChurn():

```
SELECT calculateChurn(CustomerDocument) FROM Customers WHERE Address.State == 'TX'
```

• Comparison

For example, a company may like to send out a token of gratitude to the customers who have been subscribed to a certain service for more than 40 months. GoldSubscribedMonths() can be defined to calculate the duration through the current month and date of subscription. Thus, the resulting query will be executed by comparing the result of GoldSubscribedMonths() against 40.

```
SELECT Name, Address FROM Customers WHERE GoldSubscribedMonths(CustomerID) >= 40
```

12.2.2.2. Parameters

UDFs in NosDB accept multiple parameters. Two types of parameters can be passed to User-defined functions:

1. Constant

```
SELECT GoldSubscribedMonths(23143) FROM Customers
```

2. JSON document attribute

```
SELECT CheckIfGoldCustomer("Name") FROM Customers
```

3. JSON document instance

```
SELECT CheckIfGoldCustomer(@customerDocument) FROM Customers
```

This query will iterate each document in the collection to extract the value of Name and pass as parameter to GoldSubscribedMonths().

4. Nested UDF

If the company wishes to calculate churn of only the customers who have subscribed to the Gold package, nested UDFs can be defined. Care has to be taken in keeping the return value(s) of the nested UDF consistent with the parameter(s) of the calling UDF.

```
SELECT GoldSubscribedMonths(GetGoldCustomerID(@customerDocument)) FROM Customers
```
12.3. SQL Reference

Consider the following sample document of collection *Products* in the following examples:

```json
{
    "ProductID": 10,
    "ProductName": "milk",
    "Category": "Dairy",
    "UnitsInStock": 23,
    "PackageSize": ["S", "M", "L"],
    "Types": ["Skimmed", "Full Fat"],
    "TypeBySizeAvailable": [[12, 5], [4, 17], [21, 21]],
    "Order": {
        "OrderDate": "2016/1/16",
        "OrderID": 20390,
        "OrderSize": ["S", "L"]
    }
}
```

12.3.1. Reserved Keywords

NosDB has the following reserved keywords which are not to be specified as identifiers – as variable or method names. However, they can be specified using the delimited identifiers ‘’’ (double quotes) or ‘$’ (Dollar sign).

<table>
<thead>
<tr>
<th>NULL</th>
<th>ANY</th>
<th>INSERT</th>
<th>BY</th>
<th>GRANT</th>
<th>INDEX</th>
<th>EXECUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>CONTAINS</td>
<td>REPLACE</td>
<td>ORDER</td>
<td>ON</td>
<td>COLLECTION</td>
<td>EXEC</td>
</tr>
<tr>
<td>FALSE</td>
<td>ARRAY</td>
<td>REMOVE</td>
<td>HAVING</td>
<td>REVOKE</td>
<td>NEXT</td>
<td>SELECT</td>
</tr>
<tr>
<td>ASC</td>
<td>SIZE</td>
<td>RENAME</td>
<td>TOP</td>
<td>DATABASE</td>
<td>INTO</td>
<td>DELETE</td>
</tr>
<tr>
<td>DESC</td>
<td>IN</td>
<td>FROM</td>
<td>ADD</td>
<td>AS</td>
<td>VALUES</td>
<td>DROP</td>
</tr>
<tr>
<td>WHERE</td>
<td>BETWEEN</td>
<td>GROUP</td>
<td>FUNCTION</td>
<td>ROLE</td>
<td>SET</td>
<td>TRIGGER</td>
</tr>
<tr>
<td>OR</td>
<td>EXISTS</td>
<td>LOGIN</td>
<td>ALTER</td>
<td>CREATE</td>
<td>DISTINCT</td>
<td>MASTER</td>
</tr>
<tr>
<td>AND</td>
<td>IS</td>
<td>KEY</td>
<td>LIKE</td>
<td>TRUNCATE</td>
<td>TO</td>
<td>USER</td>
</tr>
<tr>
<td>NOT</td>
<td>ROWS</td>
<td>OFFSET</td>
<td>FETCH</td>
<td>ONLY</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NosDB caters using reserved keywords in queries as identifiers with the delimited identifiers:

```
SELECT "Order" FROM Products WHERE "Order".OrderDate == DateTime("2016, 1, 20")
```

```
SELECT $Order$ FROM Products WHERE $Order$.OrderDate == DateTime("2016, 1, 20")
```
12.3.2. Binary Operators

NosDB supports unary as well as binary expressions:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>10 + 2</td>
</tr>
<tr>
<td></td>
<td>'App'+'le'</td>
</tr>
<tr>
<td>-</td>
<td>10 - 2</td>
</tr>
<tr>
<td>*</td>
<td>10 * 2</td>
</tr>
<tr>
<td>/</td>
<td>10 / 2</td>
</tr>
<tr>
<td>%</td>
<td>10 % 2</td>
</tr>
</tbody>
</table>

12.3.3. Logical Query Operators

NosDB provides support for the standard logical operators – AND, OR, NOT.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Use (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Retrieves merged result set for two or more conditions.</td>
<td>&quot;SELECT * FROM Products WHERE ProductName LIKE '%m' AND Category LIKE '%D'&quot;</td>
</tr>
<tr>
<td>OR</td>
<td>Retrieves result set which fulfills any one condition.</td>
<td>&quot;SELECT * FROM Products WHERE ProductName LIKE '%m' OR Category LIKE '%D'&quot;</td>
</tr>
<tr>
<td>NOT</td>
<td>Retrieves result set which doesn’t match the given criteria.</td>
<td>&quot;SELECT * FROM Products WHERE Category NOT LIKE '%D'&quot;</td>
</tr>
</tbody>
</table>

Note that NosDB resolves the query according to precedence. For example, the query A AND (B OR C) will first execute (B OR C) and then apply the AND operation on the resulting data set.

12.3.4. Basic Query Operators

The basic query operators are used for comparison in a query. Apart from the operators provided by SQL, NosDB offers additional operators which have been marked with an asterisk (*) in the following table:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Use (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>= or ==</td>
<td>Equals to</td>
<td>&quot;SELECT ProductName FROM Products WHERE ProductID == 10&quot;</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>SQL Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equals to</td>
<td>&quot;SELECT * FROM Products WHERE ProductID != 10&quot;</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td>&quot;SELECT * FROM Products WHERE Orders.OrderID &lt; 20100&quot;</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td>&quot;SELECT * FROM Products WHERE Orders.OrderID &gt; 20100&quot;</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than equal to</td>
<td>&quot;SELECT * FROM Products WHERE Orders.OrderID &lt;= 20100&quot;</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than equal to</td>
<td>&quot;SELECT * FROM Products WHERE Orders.OrderID &gt;= 20100&quot;</td>
</tr>
<tr>
<td><code>IN</code></td>
<td>Exists within the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE UnitsInStock IN (20,15,25)&quot;</td>
</tr>
<tr>
<td><code>NOT IN</code></td>
<td>Not within the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE UnitsInStock NOT IN (20,15,25)&quot;</td>
</tr>
<tr>
<td><code>BETWEEN</code></td>
<td>Lies within the specified range.</td>
<td>&quot;SELECT * FROM Products WHERE UnitsInStock BETWEEN 0 AND 10&quot;</td>
</tr>
<tr>
<td><code>NOT BETWEEN</code></td>
<td>Not within the specified range.</td>
<td>&quot;SELECT * FROM Products WHERE UnitsInStock NOT BETWEEN 20 AND 50&quot;</td>
</tr>
<tr>
<td><code>LIKE</code></td>
<td>Pattern like. Mostly used for wildcard base searching.</td>
<td>&quot;SELECT * FROM Products WHERE ProductName LIKE '%m' AND Category LIKE '%D'&quot;</td>
</tr>
<tr>
<td><code>NOT LIKE</code></td>
<td>Not like pattern.</td>
<td>&quot;SELECT * FROM Products WHERE ProductName NOT LIKE '%m'&quot;</td>
</tr>
<tr>
<td><code>EXISTS*</code></td>
<td>Checks if the attribute exists in the document (schema-less).</td>
<td>&quot;SELECT ProductName FROM Products WHERE $OrderDetails$ EXISTS&quot;</td>
</tr>
<tr>
<td><code>NOT EXISTS*</code></td>
<td>Checks if the attribute does not exist in the document.</td>
<td>&quot;SELECT ProductName FROM Products WHERE $Order$ NOT EXISTS&quot;</td>
</tr>
<tr>
<td><code>IS NULL*</code></td>
<td>Checks if the value against the attribute is null.</td>
<td>&quot;SELECT ProductID FROM Products WHERE ProductName IS NULL&quot;</td>
</tr>
<tr>
<td><code>IS NOT NULL*</code></td>
<td>Checks if the value against the attribute is not null.</td>
<td>&quot;SELECT ProductID FROM Products WHERE ProductName IS NOT NULL&quot;</td>
</tr>
</tbody>
</table>
12.3.5. **Aggregate Functions**

Aggregate functions return an aggregated single result after executing the query on the filtered data set.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Use (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM</td>
<td>Calculates sum of result set of specified criteria.</td>
<td>&quot;SELECT SUM(UnitsInStock) FROM Products WHERE ProductID &lt;= 10&quot;</td>
</tr>
<tr>
<td>COUNT</td>
<td>Calculates count of the result set for a specified criteria.</td>
<td>&quot;SELECT COUNT(*) FROM Products WHERE ProductID &lt; 10&quot;</td>
</tr>
<tr>
<td>AVG</td>
<td>Calculates average of the result set for specified criteria.</td>
<td>&quot;SELECT AVG(UnitsInStock) FROM Products WHERE Category = 'Dairy'&quot;</td>
</tr>
<tr>
<td>MIN</td>
<td>Returns minimum of the result set for specified criteria.</td>
<td>&quot;SELECT MIN(UnitsInStock) FROM Products WHERE Category = 'Dairy'&quot;</td>
</tr>
<tr>
<td>MAX</td>
<td>Returns maximum of the result set for specified criteria.</td>
<td>&quot;SELECT MAX(UnitsInStock) FROM Products WHERE Category = 'Dairy'&quot;</td>
</tr>
</tbody>
</table>

12.3.6. **Scalar Functions**

Scalar functions return a single value calculated against each record.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Use (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCASE</td>
<td>Converts the attribute value to upper case characters.</td>
<td>&quot;SELECT ProductID FROM Products WHERE UCASE(Category) = 'DAIRY'&quot;</td>
</tr>
<tr>
<td>LCASE</td>
<td>Converts the specified attribute to lower case.</td>
<td>&quot;SELECT ProductID FROM Products WHERE LCASE(Category) = 'meat'&quot;</td>
</tr>
<tr>
<td>LEN</td>
<td>Returns length/character count of the attribute value.</td>
<td>&quot;SELECT ProductID FROM Products WHERE LEN(Category) &gt; 5&quot;</td>
</tr>
<tr>
<td>ROUND</td>
<td>Rounds the attribute's value to the specified precision.</td>
<td>&quot;SELECT ProductID FROM Products WHERE ROUND(Price,0) &lt; 5&quot;</td>
</tr>
<tr>
<td>MID</td>
<td>Returns a substring from the attribute's value containing specified number of characters starting from the specified position of the string.</td>
<td>&quot;SELECT ProductID, MID(ProductName,2,2) FROM Products&quot;</td>
</tr>
<tr>
<td>FORMAT</td>
<td>Formats the attribute's value according to the</td>
<td>&quot;SELECT ProductID, FORMAT(NOW(),'YYYY-MM-DD') FROM Products&quot;</td>
</tr>
</tbody>
</table>
12.3.7. Special Operators for Arrays

Due to its schema-less nature and support for arrays and JSON documents, NosDB also offers a set of extended operators to accommodate querying on multivalve fields (arrays), which have SQL-like format. NosDB’s additional operators which have been marked with an asterisk (*) in the following table:

1. Querying Arrays

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Use (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTAINS ANY*</td>
<td>Checks if the array contains any of the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize CONTAINS ANY ('S', 'M')&quot;</td>
</tr>
<tr>
<td>NOT CONTAINS ANY</td>
<td>Checks if the array does not contain any of the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize NOT CONTAINS ANY ('S', 'M')&quot;</td>
</tr>
<tr>
<td>CONTAINS ALL*</td>
<td>Checks if the array contains all of the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize CONTAINS ALL ('S', 'M')&quot;</td>
</tr>
<tr>
<td>NOT CONTAINS ALL</td>
<td>Checks if the array does not contain all of the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize NOT CONTAINS ALL ('S', 'M')&quot;</td>
</tr>
<tr>
<td>ARRAY SIZE*</td>
<td>Returns records containing the array of the specified size.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize ARRAY SIZE 3&quot;</td>
</tr>
<tr>
<td>NOT ARRAY SIZE</td>
<td>Returns records which do not contain the array of the specified size.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize NOT ARRAY SIZE 3&quot;</td>
</tr>
<tr>
<td>IN</td>
<td>Exists within the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize IN (['S', 'M'], ['L', 'XL'])&quot;</td>
</tr>
<tr>
<td>NOT IN</td>
<td>Not within the specified values.</td>
<td>&quot;SELECT * FROM Products WHERE PackageSize NOT IN (['S', 'M', 'L'])&quot;</td>
</tr>
<tr>
<td>SLICE*</td>
<td>Returns a subarray (slice) of the array with specified start and end values for the slice.</td>
<td>&quot;SELECT (PackageSize) SLICE(1,2) FROM Products&quot;</td>
</tr>
<tr>
<td>SLICE MATCH*</td>
<td>Returns records with arrays which match the values for the slice.</td>
<td>&quot;SELECT (PackageSize) SLICE MATCH ('S', 'L') FROM Products&quot;</td>
</tr>
</tbody>
</table>
2. Updating Arrays

NosDB provides extended support for updating arrays. Note that the value against the attribute must exist and be of JSON array type, else the selected document will not be updated.

Consider “PackageSize”: ['S', 'M', 'L'] in the following examples:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Use (Example)</th>
</tr>
</thead>
</table>
| ADD      | Appends the specified values to the array. | "UPDATE Products SET PackageSize ADD ('XL')"  
Result: {PackageSize: ['S', 'M', 'L', 'XL']} |
| INSERT   | Checks if the array does not contain any of the specified values. | "UPDATE Products SET PackageSize INSERT ('S', 'M', 'XL')"  
Result: {PackageSize: ['S', 'M', 'L', 'XL']} |
| REPLACE  | Replaces existing value with the specified value. | "UPDATE Products SET PackageSize REPLACE ('S'='Small', 'L'='Large')"  
Result: {PackageSize: ['Small', 'M', 'Large']} |
| REMOVE   | Removes attribute from array if it exists. | "UPDATE Products SET PackageSize REMOVE ('S')"  
Result: {PackageSize: ['M', 'L']} |

12.3.8. Miscellaneous

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Use (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP BY</td>
<td>Groups data based on an aggregate function.</td>
<td>&quot;SELECT Category, COUNT(*) FROM Products WHERE ProductID &gt; 10 GROUP BY Category&quot;</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>Sorts the result set based on criteria in descending or ascending order. In case the query is applied on a non-existent attribute, it will return an empty result.</td>
<td>&quot;SELECT Product FROM Products WHERE Category = 10 ORDER BY Category&quot;</td>
</tr>
<tr>
<td>NOW()</td>
<td>Returns the current DateTime.</td>
<td>&quot;SELECT ProductName FROM Products WHERE Orders.OrderDate = NOW()&quot;</td>
</tr>
<tr>
<td>DateTime(&quot;any date time compatible string&quot;)</td>
<td>Retrieves result set with respect to specified date time.</td>
<td>&quot;SELECT Order FROM Products WHERE Orders.OrderDate = DateTime('2017-1-20')&quot;</td>
</tr>
<tr>
<td>TOP</td>
<td>Returns the first specified number of records and executes query on the specified records.</td>
<td>&quot;SELECT TOP 10 ProductName FROM Products WHERE ProductID = 10&quot;</td>
</tr>
</tbody>
</table>
OFFSET | Skips the specified number of resultant records. | "SELECT ProductName FROM Products WHERE ProductID = 10 OFFSET 10 ROWS"

OFFSET FETCH | An extension of the OFFSET keyword, skips the specified rows and fetches a limited number of records from the result set. | "SELECT ProductName FROM Products WHERE ProductID = 10 OFFSET 20 ROWS FETCH NEXT 500 ROWS ONLY"

DISTINCT | Returns only the distinct values in the result set. | "SELECT DISTINCT ProductName FROM Products"

RENAME TO | Renames the attribute. | "UPDATE Products SET RENAME PackageSize TO 'PackagingSize'"

12.3.9. Special Considerations for WHERE Clause

NosDB’s schema-less nature accommodates arrays and JSON documents which can be embedded; for example, an array of JSON documents, or JSON documents containing arrays.

The WHERE clause can be used in the following ways to access these arrays and documents:

- WHERE Products.PackageSize[0] = 'S'
- WHERE Products.TypeBySizeAvailable[0][1] < 15
- WHERE Products.$Order$.OrderID > 25000
- WHERE Products."Order".OrderSize[0] = 'S'
13. Query Optimization

In This Chapter:
Query Execution Path
Query Optimization in NosDB

13.1. Query Execution Path

A query executed against a set of documents in NosDB passes through the following components to reach the client:

1. **Parser**
   - Parses the query and verifies whether the grammar and syntax of the query is correct or not. For example, the query "SELECT ProductName WHERE UnitsInStock < 10 FROM Products" will be marked as incorrect because it does not follow the standard pattern, and the client will be notified accordingly.
   - Extracts the meaningful objects from the query - both the objects to project and filter. For example, in the query "SELECT ProductName FROM Products WHERE UnitsInStock < 10", the objects ProductName, Products, and UnitsInStock < 10 will be extracted to be searched in storage or indices.

2. **Optimizer**

   Selects the most optimized execution path for the query by applying cost-based techniques on the WHERE clause. This cost is determined by the number of I/O operations required in each path. It also transforms the query in some cases in order to yield more optimization.

   For example, the query "SELECT ProductName FROM Products WHERE UnitsInStock < 10 AND CATEGORY = 'Dairy'" can follow various paths, like scanning the whole dataset or searching single-attribute indexes like Category or UnitsInStock, or a compound index [Category + UnitsInStock]. The most optimized path for
this query will be the compound index. Please refer to Query Optimization for in-depth detail about optimization in NosDB.

3. Executor

Executes the query according to the optimized path selected by the optimizer. Continuing with the aforementioned example in optimizer, the executor will access the [Category + UnitsInStock] index and extract the product names from the documents against each matching key. The results will be sent to the result manager for managing query execution efficiently.

4. Result Manager

The result manager is located at the database level and acts as a bookkeeper for the queries and their respective result sets. Each query is assigned a unique reader ID, which is sent along with its corresponding query result set to the result manager. This helps in keeping track of query execution and provides fault tolerance in case communication between the client and distributor breaks.

5. Distributor

The distributor acts accordingly for each query type.

1. Scalar Queries

Since scalar queries return a single result against each document, the result set can be considerably large which is not feasible to send over the network. Hence, the result is sent in chunks to the distributor, through which the client reads the items transparently. This reduces the network burden and grants faster response time to the client.

Once the number of items fetched by client reaches a certain threshold, the distributor requests for more chunks from the result manager after checking availability of chunks at the shard. This request contains the reader ID of the query and the last chunk ID, so the next chunks are sent through the reader ID’s corresponding result set at the result manager. Hence, it is ensured that the distributor does not exhaust its items and data transfer to the client remains smooth.

For example, after the query "SELECT ProductName FROM Products WHERE Category = 'Dairy'" is executed, Shard A has 500 results and Shard B has 400 results. However, the result manager will send chunks of let’s suppose, 100 results each to the distributor, so the distributor would have 200 result items. The client will iteratively fetch the result from among these 200 results. The shards then send chunks of additional 100 items each upon the distributor's request when the chunk is about to exhaust.
II. Aggregate Queries

All aggregate functions are combined at the distributor, because the environment is distributed and the client requires a single result.

For example, the MAX() function is individually executed on each shard of the server, which returns its respective maximum values to the distributor. The distributor then applies MAX() over the values received from the particular shards and finally sends the maximum among these values to the client.
13.2. Query Optimization in NosDB

For a database with humungous volume of unstructured data, it is necessary that the queries which access/modify data are exceptionally optimized for high performance. Hence, NosDB determines an efficient execution method for each query, such that network overhead, I/O, storage are all taken into account. The optimizer also contains statistics like required resources, number of records to be returned, and indexing information. These statistics aid in calculating the I/O rate.

The optimizer undertakes the cost-based technique for optimization. In relational databases, the cost is usually defined by the number of bytes to skip in order to reach the specified element in storage. However, since NosDB integrates LMDB in its storage module, the number of bytes reserved for each item are not disclosed. Therefore, NosDB also considers cost as the number of I/O operations required to access the item.

Thus, the optimizer generates all possible optimal query plans and selects the one with lowest cost (IOs) based on the resources and statistics available in the environment.

The following activities are considered as I/O operations:
- Reading from file store
- Accessing an index

13.2.1. Factors Affecting Optimization

13.2.1.1. Statistics

The optimizer keeps track of various statistics to calculate cost upon which the optimization decision relies:

1. Indexing statistics
   - Total keys
   - Unique keys
   - Minimum key (numeric)
   - Maximum key (numeric)
   - Depth of the index (B+ Tree)

2. Storage statistics
   - Total number of documents
   - Expected number of I/O operations
   - Expected number of records to be returned

3. System statistics
   - Resources
   - Performance

8.2.2.2. Indexing

Indexing the items boosts query performance as a whole. Since the query optimizer checks for indices built on the required field, it is efficient if frequently used queries have the indexes built beforehand. This saves cost as accessing an index also accounts for an I/O operation.
There are two types of queries on indices:

I. **Equality comparisons**

Equality comparisons (=, !=, <>) on indices will return with accurate cardinalities regardless of the type of the values stored in an index.

II. **Inequality comparisons**

Inequalities (<, >, <=, >= ...) will return accurate cardinalities only if the queried value’s data type is numeric.

The predicates which have no indices to entertain their variables assume selection cardinality as all the docs in the collection.

For example, consider the query "SELECT * FROM Customers WHERE Age > 20 AND State = 'TX'" on a database with the following statistics:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Selection Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>State = 'TX'</td>
<td>70,000</td>
</tr>
<tr>
<td>Age &gt; 20</td>
<td>120,000</td>
</tr>
<tr>
<td>Age &gt; 20 AND State = 'TX'</td>
<td>250</td>
</tr>
</tbody>
</table>

- In case of no index, the query is executed against the whole data set:
  - Search for all documents with State = TX (70,000 records returned)
  - Search for all documents with Age > 20 (120,000 records returned)
  - Apply the AND operation over these records (250 records returned)

  Hence, it proves as a complete waste of performance and cost, enhancing the need to build indexes for this frequently used query.

- In case of an index built on either of the fields, the cost of traversing the whole data set is reduced as the location of the required records is known through the index in one hop.

- In case of a compound index on [Age, State], the exact records (250) of the query are retrieved and there is no need to apply the AND operation over thousands of records.
8.2.2.3. Transformation Rules

NosDB’s optimizer transforms the query wherever the attributes of a query can be accommodated by existing compound indices. Transformation refers to simplification of the predicate into various combinations by incorporating the standard relational algebra rules.

The cost incurred on each transformed query is then calculated before selecting the least cost (most optimal) path. One of the most simplistic transformations take place by applying the distributive law over the predicate for cost efficiency. For example, a predicate \( A \ AND \ (B \ OR \ C) \) is transformed into \( (A \ AND \ B) \ OR \ (A \ AND \ C) \), if the indices \([A \ AND \ B]\) and \([A \ AND \ C]\) exist.

Invalid ranges in queries:

- \( ID < 2 \) and \( ID < 3 \)
- \( ID < 1 \) and \( ID > 3 \)
- \( ID < 2 \) and \( ID < \text{“John”} \)
- \( 1 = 1 \)

Invalid predicates return an empty value and no transformation takes place because zero records are returned. However, in case of \( ID < 2 \) and \( ID < 3 \), the records with \( ID < 3 \) will be considered.

13.2.2. Optimization Process

After the query is parsed, the optimizer:

1. Checks for any relevant compound indices that could entertain the attributes for projection or the \textit{WHERE} predicates. Compound indices dramatically decrease cost as compared to accessing single-attribute indices and then merging their results.

There can be multiple indices built with various combinations that could fit the predicates.

In case any suitable compound index exists, the optimizer tries to fit the predicates in an indexed-execution-predicate with the index as its data-source.

2. Transforms the query according to transformation rules to accommodate the predicates within the indices and make the search limited.

3. Calculates selection cardinalities against each transformed option available. Selection cardinality of a predicate is accurately calculated only if an index could feed its attribute(s). Similarly, selection cardinality of multiple predicates can be calculated by means of single compound index, if it exists.

4. Calculates cost (I/O operations required) for each index of the predicates based on the indexing statistics and selection cardinality.

5. Selects the least cost execution path from among all expected query execution paths. The executor will execute all future queries of a similar pattern based on this optimized path, where the compound indexes will be accessed directly. In case the collection changes, the execution path is regenerated.
14. Joins Support in NosDB

In This Chapter:

- Joins Architecture in NosDB
- Join Behavior in NosDB
- Supported Join Types

This feature is only available in NosDB Enterprise Edition.

NosDB offers Joins in NoSQL, which are implemented using MapReduce framework on the base of 1-Bucket Theta Join algorithm.

```sql
SELECT cust.ContactName, cust.Phone, ord.OrderDate, ord.ShippedDate, ord.Freight
FROM Customers cust
JOIN Orders ord
ON cust.CustomerID = ord.CustomerID
WHERE cust.City = "NY"
AND ord.Status = "Pending"
```

14.1. Join Architecture in NosDB

14.1.1. Map Phase

- Each mapper runs per node and contains the Reducer matrix. For the collections Customers containing $C_m$ documents, and Orders containing $O_n$ documents, a Reducer matrix has the dimensions $(C_m \times O_n)$.
- Based on the cardinality of both collections, NosDB computes the number of reducers internally.
- The mapping begins with the Mapper iterating over the Customers collection, and assigning a random row number of the reducer matrix to each document.
- Similarly, it iterates over the Orders collection, and assigns a row number of the reducer matrix to the document.
- Using the row number and reducer matrix, the mapper emits multiple key-value pairs for a single document where every key is sent to its assigned reducer.

14.1.2. Reduce Phase

- The reducers receive their respective documents for Customers and Orders.
- Each reducer performs the join over the documents based on the joining criteria.
- The primary keys of all documents being joined are sent as the output of the reducer to a Join Query Store, which contains a union of the outputs of all reducers.
- The Join matrix contains keys of the Customers and Orders documents which had successful join results.
- Projection occurs when a chunk of data is fetched through the reader.

Figure 1 illustrates the concept:

1. The Map phase shows a record is assigned a row and each cell of the row refers to a reducer.
2. For simplicity, we depict two of the records from the Reducer matrix. \((C_1, O_1)\) points to reducer 1, while \((C_1, O_3)\) points to reducer 2.
3. In the Reduce phase, the corresponding Customers and Orders documents are joined based on the joining criteria.
4. The primary keys of all document pairs which have a true output for the join query are sent to the Join Query Store in the form of a join matrix.

**Figure 1: Joins Architecture**

### 14.2. Join Behavior in NosDB

Presently, NosDB supports maximum 2 collections in a single Join query.

#### 14.2.1. Correlation Names

Correlation names must exist in the Join queries. This helps in distinguishing collections and attributes as the same attribute can exist in multiple collections. For example, CustomerID can exist in Customers and Orders.

#### 14.2.2. Projection

When using "SELECT *" in a join query, there is a chance that both collections contain the same attribute, hence the projection after the Join would result in duplicate attributes. Since JSON does not allow duplicate attributes, NosDB internally appends the correlation with the attribute name to differentiate the attributes. For example, ord_CustomerID and cust_CustomerID.

#### 14.2.3. Nulls

For Outer Joins, the Joiner is not filtered so the resultant can contain null data. For example,
SELECT * FROM Customers cust LEFT JOIN Orders ord ON cust.CustomerID = ord.CustomerID;

There can be customers who have not placed any orders, thus the joining matrix contains all matched records of the Customers and Orders where the joining attribute is true, and since it is an outer left join, all unmatched records of Customers are also included in the resultant.

<table>
<thead>
<tr>
<th>ContactName</th>
<th>Address</th>
<th>Phone</th>
<th>CustomerID</th>
<th>ord_CustomerID</th>
<th>ord_OrderID</th>
<th>ord_OrderDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Simon</td>
<td>Obere Str. 57</td>
<td>030-0074321</td>
<td>REGGC</td>
<td>REGGC</td>
<td>24453</td>
<td>2016-07-08T00:00:00</td>
</tr>
<tr>
<td>Lara Trujillo</td>
<td>Avda. de la Constitución 2222</td>
<td>(5) 555-4729</td>
<td>OCEAN</td>
<td>OCEAN</td>
<td>87444</td>
<td>2016-02-03T00:00:00</td>
</tr>
<tr>
<td>Victoria Ashworth</td>
<td>Fauntleroy Circus</td>
<td>(171) 555-1212</td>
<td>MAISD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diego Roel</td>
<td>Moralzarzal, 86</td>
<td>(91) 555-94 44</td>
<td>FRANS</td>
<td>FRANS</td>
<td>98356</td>
<td>2016-02-05T00:00:00</td>
</tr>
<tr>
<td>Howard Snyder</td>
<td>2732 Baker Blvd.</td>
<td>(503) 555-7555</td>
<td>GREAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

However, these null Orders records are not made part of the projection; only the Customers documents are projected to the client side.

Currently, Joins do not cater Aggregation and Grouping in NosDB.
15. MapReduce

In This Chapter:
- MapReduce Architecture
- How does MapReduce work?
- Example MapReduce Execution

The MapReduce framework in NosDB allows robust analytical processing on voluminous data in a scalable environment. It takes unstructured, large sets of data to process them into a meaningful and concise set of results. To distribute input data and analyze it simultaneously, MapReduce operates in parallel fashion on all shards in a cluster of any size.

15.1. MapReduce Architecture

Once the client sends a request to perform a MapReduce task, it first communicates with the configuration server by requesting for a master shard which could act as a coordinator shard for auto balancing of data and distribution of tasks over all the connected shards. The client then deploys its MapReduce assemblies to the configuration server and the MapReduce task is initiated with the deployed assemblies specified.

As soon as the task is submitted via the distributor, the master shard ensures that all shards start the task execution in parallel. The master shard keeps track of each shard’s task status and reports it to the configuration server in case the client requests for the status.

MapReduce is executed at collection level, that is, you can perform the MapReduce tasks over multiple collections.

Note that because of its document-oriented nature, the input and final output of MapReduce has to be in JSON format as it is persisted to a collection. The output can either be persisted to an existing collection, a user-defined collection, or NosDB’s default output collection.

15.2. How does MapReduce Work?

NosDB MapReduce has three phases: Map, Combine, and Reduce. Only the Mapper is necessary to implement – the Reducer and Combiner implementation is optional.

The Mapper, Combiner and Reducer are executed simultaneously during a NosDB MapReduce task on the NosDB cluster. A typical MapReduce task includes the following components:

**Mapper:** Processes collection-specific (JSON format) input to generate a set of intermediate key-value pairs which are discretely sent to either Combiner or Reducer for further refining and extraction of the data.

**Combiner Factory:** Assigns a unique Combiner to each document provided to it. User can implement it to provide different combiners for different documents.

**Combiner:** Works as a local reducer to the node where Mapper’s output is combined to minimize traffic between Mapper and Reducer. The tasks are processed and stored in bulk before being sent to the Reducer, i.e., the data from Mapper is processed in chunks - the size of which is configurable. Once the combined results reach the specified chunk size, the elements are forwarded to the Reducer.
**Reducer Factory:** Assigns a unique Reducer to each document provided to it. User can implement it to provide different reducers for different documents.

**Reducer:** Processes all those intermediate key-value pairs generated by Mapper or combined by Combiner to aggregate, perform calculations or apply operations to produce the reduced output which can be persisted to a collection.

**Query Filter:** Optionally filters collection data based on the query executed by client before being sent to the Mapper. The Mapper will be executed on the result of this query.

**Document Filter:** Optionally filters collection data based on the specified document ID before being provided to the Mapper. If it returns true, the Map will be executed on the specified documents returned.

**Tracker Task:** Tracks progress of the task and its status as the task is executed. It also lets you fetch the output of the task and enumerate over it.

**Output:** The output from Reducer is persisted in a collection in the physical file storage. Hence, it has to be in JSON format necessarily.

### 15.3. Example MapReduce Execution

For example, an airline company wishes to analyze the top 3 destinations of flights originating from Seattle. Considering the humungous amount of data stored for the flight details, querying for the top three destinations at database level over WAN is extremely expensive and degrades performance significantly. Thus, it is useful if such heavy analyses are made at the server end, where only the result is sent over the network, preventing any choking.

![Figure 12.1: MapReduce Execution Components](image-url)
**Query Filter:** The MapReduce functionality is further made optimal by querying for documents which have the origin as Seattle. The result set includes all those documents which have the field “Origin: Seattle”.

**Mapper:** Extracts the destination city from each document and maps it against the “City” field.

**Combiner:** Counts the occurrence of each city and stores it in a key-value format in a dictionary to send to the reducer.

**Reducer:** Combines the values from all Combiners into a sorted dictionary and extracts the top 3 cities to be sent over the network to the client.