# Table of Contents

1. Introduction to NosDB ........................................................................................................... 2
2. Setting Up Java Development Environment ........................................................................ 3
3. Database Connection ............................................................................................................. 11
   3.1. Connect to Database ......................................................................................................... 11
4. Basic Operations through Document API ........................................................................... 13
   4.1. Using Database Collection ............................................................................................... 13
   4.2. Insert Documents ............................................................................................................ 14
   4.3. Get Documents ............................................................................................................... 17
   4.4. Update Documents .......................................................................................................... 19
   4.5. Delete Documents ........................................................................................................... 21
5. Querying in NosDB ................................................................................................................ 22
   5.1. Data Manipulation Language (DML) Support ................................................................. 22
   5.2. SQL Reference ............................................................................................................... 27
6. Basic Operations through SQL API ..................................................................................... 34
   6.1. ExecuteReader() Method .............................................................................................. 35
   6.2. ExecuteNonQuery() Method ............................................................................................ 37
   6.3. ExecuteScalar() Method ............................................................................................... 38
7. Using NCache with NosDB ................................................................................................... 39
   7.1. NCache Supported Features ............................................................................................ 40
   7.2. Integrating NCache Provider .......................................................................................... 41
   7.3. Using Configured NCache Provider ................................................................................. 43
   7.4. Caching Behaviors ......................................................................................................... 45
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

NosDB 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><code>courier</code></td>
<td>Specifies inline code snippets, file, class, interface names.</td>
</tr>
<tr>
<td><code>monospace</code></td>
<td>Specifies any JSON document examples within the text.</td>
</tr>
<tr>
<td><code>monospace</code></td>
<td>Specifies any SQL examples within the text.</td>
</tr>
<tr>
<td>📝</td>
<td>Specifies additional and vital information for the user.</td>
</tr>
<tr>
<td>🔄</td>
<td>Specifies any significant step to be taken in your application.</td>
</tr>
</tbody>
</table>
2. Setting Up Java Development Environment

The following section has been implemented using NetBeans IDE 8.0 and NosDB version 2.0.

- Open File -> New Project...

- Select Java Application in the Category Java.
- Click Next to create a new project.
- Specify the **Project Name** and **Project Location**.

- Check the box if you want to dedicate a folder for any libraries, or if you want to create the **Main Class**.

- Click **Finish**.
To this solution, you need to add the NosDB client library and resources in order to utilize its features.

- Right-click on the **Projects** tab of the left pane and select **Properties**.
- **Project Properties** window opens up.
- Select the **Libraries** option in the **Categories** pane.
- Click on **Add JAR/Folder**.
- **Add JAR/Folder** window appears.

- **Browse** to `InstallDir\java\lib`

- **Select** `nosdb-client.jar` and **click Open**.
• Click Add JAR/Folder again.
• **Browse to [InstallDir]\java\lib\resources**
• **Select all of the jar files:**
• **Click Open.**
The added JARs are displayed.

Click **OK**.
You can now create applications utilizing NosDB’s features.
3. Database Connection

To utilize the following APIs, import the following package in your application:

```java
import com.nosdb.client.*;
```

3.1. Connect to Database

A NosDB database instance is initiated by connecting it to the configured database, which can be either clustered or stand-alone. The Java client for NosDB works through NosDB Distributor Service (NosDistributorSvc) which has to be connected to.

NosDB Java Client uses NosDB Authentication as its authentication model. NosDB Authentication is similar to the SQL Server Authentication in SQL Server, which consists of a custom username-password authentication model.

The `getDatabase()` method is provided with the connection string (and optional parameters) for the configured database. A connection string in NosDB requires the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Source</td>
<td>Server name/IP</td>
</tr>
<tr>
<td>Port</td>
<td>Configuration server port</td>
</tr>
<tr>
<td>Database</td>
<td>Database name</td>
</tr>
<tr>
<td>Local Instance</td>
<td>Boolean value to specify clustered database (false) or standalone (true)</td>
</tr>
<tr>
<td>Distributor Address</td>
<td>IP of the machine running the NosDB Distributor Service (NosDistributorSvc)</td>
</tr>
<tr>
<td>Distributor port</td>
<td>Port of Distributor</td>
</tr>
<tr>
<td>User Id</td>
<td>User Id for NosDB Authentication</td>
</tr>
<tr>
<td>password</td>
<td>Password for NosDB Authentication</td>
</tr>
</tbody>
</table>

The NosDB Distributor Service (NosDistributorSvc) **must be started manually** before attempting to establish a connection.
• Initializing Clustered Database

```java
//connection string
String conn = "Data Source=127.0.0.1; Port=9950; Database=northwind; Local Instance=false; Distributor Address=127.0.0.1; Distributor port=9970; User Id=admin; password=nosdb123;"

//initialize database
Database db = NosDBClient.getDatabase(conn);
```

• Initializing Stand-alone Database

A stand-alone database requires the tag `Local Instance` in the connection string to be set to `true`:

```java
//connection string
String conn = "Data Source=127.0.0.1; Port=9950; Database=northwind; Local Instance=true; Distributor Address=127.0.0.1; Distributor port=9970; User Id=admin; password=nosdb123;"

//initialize database
Database db = NosDBClient.getDatabase(conn);
```
4. Basic Operations through Document API

*In This Chapter:*
- Using NosDB Collection
- Insert Documents
- Get Documents
- Update Documents
- Delete Documents

4.1. Using Database Collection

An instance of NosDB's `DBCollection` class is created and initiated through the `getCollection()` method which returns the configured collection. All operations performed on this instance will be reflected to the configured collection. This collection can take in either a JSON document instance or custom .NET objects. Each document and object has a unique identifier, the document key (`_key`) which acts as a primary key for the collection.

> To utilize the following APIs, import the following package in your application:

```java
import com.nosdb.client.*;
```

4.1.1. With JSON Documents

```java
//create collection instance
DBCollection<JSONDocument> collection = db.getCollection("Products");
```

4.1.2. With Custom Objects

```java
//create collection instance of Product class type
DBCollection<Product> collection = db.getCollection("Products", Product.class);
//pre-requisite: "Products" exists
```
4.2. Insert Documents

NosDB supports two types of data objects – JSON Documents and Custom Class Objects.

JSON Documents

NosDB offers the JSONDocument class to create JSON document instances and perform operations like adding and removing attributes.

To utilize the following APIs, import the following package in your application:

```java
import com.nosdb.client.*;
```

```java
//create JSON document
JSONDocument product = new JSONDocument();
```

Custom Objects

Apart from adding JSON documents, NosDB also allows adding custom .NET objects to the collection. The key for the custom objects is added as a property with the annotation @SerializedName("_key") before the attribute you want to use as a key. In the following example, the value of ProductID will be the value of _key in NosDB. However, if no key is specified, the _key attribute will be initialized with a default GUID automatically.

To specify the key, import the following package in your application:

```java
import com.google.gson.annotations.SerializedName;
```

```java
public class Product {
    @SerializedName("_key") //key attribute will have value of ProductID
    public String ProductID;
    public String Name;
    public Double UnitPrice;
    public Integer UnitsInStock;
    public Boolean Discontinued;

    public Category Category;

    //Getters setters
}
```

```java
public class Category {
    public Integer ID;
    public String Name;
    public String Description;

    //Getters setters
}
```
4.2.1. Single Document

The `insertDocument()` method inserts a single document to the database by taking in an instance of the document along with an optional `WriteConcern` value. By default, the `WriteConcern` is set to `InMemory`.

- **JSON Document**

```java
JSONDocument category = new JSONDocument();
category.add("ID", 1);
category.add("Name", "Beverages");

JSONDocument product = new JSONDocument();

product.add("ProductID", "12");
product.add("Name", "Chai");
product.add("Discontinued", false);
product.add("UnitPrice", 18.00);
product.add("UnitsInStock", 38);
product.add("Category", category);

//with default (InMemory) WriteConcern
collection.insertDocument(product);

//with Journal WriteConcern
collection.insertDocument(product, WriteConcern.Journal);
```

- **Custom Object**

```java
Product product = new Product();

product.setProductID("12");
product.setName("Chai");
product.setDiscontinued(false);
product.setUnitPrice(18.00);
product.setUnitsInStock(38);

//embedded attributes in Category class
Category category = new Category();
category.setID(1);
category.setName("Beverages");

product.setCategory(category);

collection.insertDocument(product);
```
4.2.2. Bulk Documents

The `insertDocuments()` method inserts documents in bulk to the database by taking in a `Collection` of documents along with an optional `WriteConcern` value. By default, the `WriteConcern` is set to `InMemory`.

- **JSON Document**

```java
JSONDocument category = new JSONDocument();
category.add("ID", 1);
category.add("Name", "Beverages");

product1.add("ProductID", "12");
product1.add("Name", "Chai");
product1.add("UnitPrice", 18.00);
product1.add("UnitsInStock", 38);
product1.add("Category", category);

product2.add("ProductID", "34");
product2.add("Name", "Aniseed Syrup");
product2.add("UnitPrice", 10.00);
product2.add("UnitsInStock", 13);

java.util.Collection<JSONDocument> docs = new java.util.ArrayList<>();
docs.add(product1);
docs.add(product2);
java.util.ArrayList<FailedDocument> failedDocs = collection.insertDocuments(docs);
```
### Custom Objects

```java
product1.setProductID("12");
product1.setName("Chai");
product1.setUnitPrice(18.00);
product1.setUnitsInStock(38);
product1.setDiscontinued(false);

Category category1 = new Category();
category1.setID(1);
category1.setName("Beverages");
product1.setCategory(category1);

product2.setProductID("34");
product2.setName("Aniseed Syrup");
product2.setUnitPrice(10.00);
product2.setUnitsInStock(13);
product2.setDiscontinued(false);

Category category2 = new Category();
category2.setID(2);
category2.setName("Condiments");
product2.setCategory(category2);
```

```java
java.util.Collection<Product> products = new java.util.ArrayList<>();
products.add(product1);
products.add(product2);
ArrayList<FailedDocument> failedDocs = collection.insertDocuments(products);
```

### 4.3. Get Documents

#### 4.3.1. Single Document

The `getDocument()` method fetches the corresponding document from the database by taking in a `DocumentKey` along with an optional `ReadPreference` value. The document is returned to an instance of the object type added. By default, the `ReadPreference` is set to `PrimaryOnly`.

- **JSON Document**

  ```java
  //with default (PrimaryOnly) ReadPreference
  JSONDocument result = collection.getDocument("key1");
  
  //with Load Balanced ReadPreference
  JSONDocument result = collection.getDocument("key1", ReadPreference.LoadBalanced);
  //perform operations
  ```
4.3.2. Bulk Documents

The `getDocuments()` method fetches the corresponding documents from the database specified through a list of document keys along with an optional `ReadPreference` value and the option to use caching. The documents are returned to an instance of `DBCollectionReader`, which can be enumerated to read the results. By default, the `ReadPreference` is set to `PrimaryOnly`. For more detail on Caching, please refer to the chapter Using NCache with NosDB.

```java
// Custom Objects
// with default (PrimaryOnly) ReadPreference
Product result = collection.getDocument("12");
// perform operations

// JSON Documents
java.util.Collection<DocumentKey> docKeys = new ArrayList<>(); // collection of document keys

docKeys.add("key1");
docKeys.add("key2");

// with default (PrimaryOnly) ReadPreference
DBCollectionReader reader = collection.getDocuments(docKeys);

while (reader.readNext())
{
    JSONDocument doc = reader.getDocument();
    // perform operations
}

String criteria = "SELECT * FROM Products WHERE ProductName = 'Chai';

// with caching enabled and LoadBalanced ReadPreference
DBCollectionReader reader = collection.getDocuments(criteria, true, ReadPreference.LoadBalanced);

// with caching disabled and default (PrimaryOnly) ReadPreference
DBCollectionReader reader = collection.getDocuments(criteria, false);

while (reader.readNext())
{
    JSONDocument doc = reader.getDocument();
    // perform operations
}
Custom Objects

```java
java.util.Collection<DocumentKey> docKeys = new java.util.ArrayList<>();
docKeys.add("12");
docKeys.add("34");

//with default (PrimaryOnly) ReadPreference
DBCollectionReader<Product> reader = collection.getDocuments(docKeys);
while (reader.readNext())
{
    Product result = reader.getObject();
    //perform operations
}
```

4.4. Update Documents

4.4.1. Single Document

The `updateDocument()` method updates an existing document by taking in an instance of the updated document (against the same key) along with an optional `WriteConcern` value. By default, the `WriteConcern` is set to `InMemory`.

JSON Document

```java
JSONDocument category = new JSONDocument();
category.add("ID", 1);
category.add("Name", "Drinks"); //CategoryName changed from Beverages to Drinks

JSONDocument product = new JSONDocument();
product.add("ProductID", "12");
product.add("ProductName", "Chai"); //Name changed to ProductName
product.add("Discontinued", false);
product.add("UnitPrice", 19.50); //UnitPrice changed from 18.00 to 19.50
product.add("UnitsInStock", 38);
product.add("Category", category);

//with default (InMemory) WriteConcern
collection.updateDocument(product);
```
### Custom Objects

```java
Product product = new Product();

product.setProductID("12");
product.setName("Chai");
product.setUnitPrice(19.50);  //UnitPrice changed from 18.00 to 19.50
product.setUnitsInStock(38);

//embedded attributes in Category class
Category category = new Category();
category.setID(1);
category.setName("Drinks");  //CategoryName changed from Beverages to Drinks

product.setCategory(category);
collection.updateDocument(product);
```

### 4.4.2. Bulk Documents

The `updateDocuments()` method updates a bulk of existing documents, specified by a `List<JSONDocument>` of updated documents (against the unchanged keys) along with an optional `WriteConcern` value. By default, the `WriteConcern` is set to `InMemory`.

#### JSON Documents

```java
JSONDocument category = new JSONDocument();
category.add("ID", 1);
category.add("Name", "Beverages");

product1.add("ProductID", "12");
product1.add("ProductName", "Chai");
product1.add("UnitPrice", 19.50);
product1.add("Category", category);

product2.add("ProductID", "34");
product2.add("Name", "Aniseed Syrup");
product2.add("UnitPrice", 10.00);

java.util.Collection<JSONDocument> docs = new java.util.ArrayList<>();  // collection of documents
docs.add(product1);
docs.add(product2);

java.util.ArrayList<FailedDocument> failedDocs = collection.updateDocuments(docs);
```
### Custom Objects

```java
product1.setProductID("12");
product1.setName("Chai");
product1.setUnitPrice(19.50); // updated price
product1.setDiscontinued(false);

Category category1 = new Category();
category1.setId(1);
category1.setName("Beverages");
product1.setCategory(category1);

product2.setProductID("34");
product2.setName("Aniseed Syrup");
product2.setUnitsInStock(4); // updated unitsInStock
product2.setDiscontinued(false);

Category category2 = new Category();
category2.setId(2);
category2.setName("Condiments");
product2.setCategory(category2);
```

```java
java.util.Collection<Product> products = new java.util.ArrayList<>();
products.add(product1);
products.add(product2);
ArrayList<FailedDocument> failedDocs = collection.updateDocuments(products);
```

### 4.5. Delete Documents

#### 4.5.1. Single Document

The `deleteDocument()` method deletes the document specified through a String document key along with an optional `WriteConcern` value. By default, the `WriteConcern` is set to `InMemory`.

```java
collection.deleteDocument("key1"); // with default (InMemory) WriteConcern
collection.deleteDocument("12", WriteConcern.JOURNAL); // with Journal WriteConcern
```

#### 4.5.2. Bulk Documents

The `deleteDocuments()` method deletes a bulk of existing documents specified by a `List<String>` of document keys along with an optional `WriteConcern` value. By default, the `WriteConcern` is set to `InMemory`.

```java
java.util.Collection<String> docKeys = new ArrayList<>();
docKeys.add("key1");
docKeys.add("key2");
ArrayList<FailedDocument> failedDocs = collection.deleteDocuments(docKeys);
```
5. Querying in NosDB

In This Chapter:

Data Manipulation Language (DML) Support
SQL Reference

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.

5.1. Data Manipulation Language (DML) Support

NosDB supports DML to exploit the data for CRUD operations like SELECT, DELETE, INSERT and UPDATE.

The following convention has been used for Query notation:

<table>
<thead>
<tr>
<th>monospace</th>
<th>Document Query Language keywords</th>
</tr>
</thead>
</table>
| [ ]        | Clauses within "[ ]" are **optional**; e.g. `WHERE <Predicate>`.
| [ ]*       | Arguments in `[ ]*` can be **repeated for 0 or more number of times**; e.g. `[<BinaryExpression>]*`.
| | Represents a **choice** between arguments of a query; e.g. `[ASC | DESC]`.

Universal Syntax:

```
<BinaryExpression> = [<JSONValue> | <Attribute> | <Function> | '(' <BinaryExpression> ')'] [[+-|*|%] [<JSONValue> | <Attribute> | <Function> | '(' <BinaryExpression> ')']]*
```

```
<Identifier> = Identifier rule
<JSONValue> = JSON Value
<Attribute> = Document attribute
<Function> = Query specifiable function
<Predicate> = Filtration predicate
```
Consider the following sample documents for collections *Products* and *Orders* in all succeeding examples:

<table>
<thead>
<tr>
<th>Products</th>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>&quot;key&quot;: &quot;DA123&quot;,</td>
<td>{</td>
</tr>
<tr>
<td>&quot;ProductID&quot;: 10,</td>
<td>&quot;key&quot;: &quot;TE1368&quot;,</td>
</tr>
<tr>
<td>&quot;Name&quot;: &quot;Chai&quot;,</td>
<td>&quot;OrderID&quot;: 10248,</td>
</tr>
<tr>
<td>&quot;UnitPrice&quot;: &quot;18.00&quot;,</td>
<td>&quot;CustomerID&quot;: &quot;VINET&quot;,</td>
</tr>
<tr>
<td>&quot;UnitsInStock&quot;: 23,</td>
<td>&quot;OrderDate&quot;: &quot;2015-07-04&quot;,</td>
</tr>
<tr>
<td>&quot;Discontinued&quot;: &quot;False&quot;,</td>
<td>&quot;Freight&quot;: 32.38,</td>
</tr>
<tr>
<td>&quot;Category&quot;: {</td>
<td>&quot;ShipName&quot;: &quot;Vins et alcools Chevalier&quot;</td>
</tr>
<tr>
<td>&quot;ID&quot;: 1,</td>
<td>&quot;OrderDetails&quot;: [</td>
</tr>
<tr>
<td>&quot;Name&quot;: &quot;Beverages&quot;,</td>
<td>&quot;ProductID&quot;: &quot;1&quot;,</td>
</tr>
<tr>
<td>&quot;Description&quot;: &quot;Soft drinks,</td>
<td>&quot;UnitPrice&quot;: 14.0,</td>
</tr>
<tr>
<td>coffees, teas, beers and ale&quot;</td>
<td>&quot;Quantity&quot;: 12,</td>
</tr>
<tr>
<td>}</td>
<td>&quot;Discount&quot;: 0.05</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

5.1.1. SELECT Statement

Selects JSON document(s) from a particular collection. Optional clauses can be used to filter the results, along with aliasing and specific querying using UDFs and stored procedures.

```
SELECT
[DISTINCT]
[TOP <IntegerValue> | TOP (<IntegerValue>)]
([[<BinaryExpression> [AS <Identifier>]]|'*'] |,[<BinaryExpression> [AS <Identifier>]]|'*']*)
FROM <Identifier>
[[INNER] | LEFT | RIGHT] JOIN <Identifier> ON <JoinExpression>]*
[WHERE <Predicate>]
[ORDER BY (<BinaryExpression> [ASC|DESC][, <BinaryExpression> [ASC|DESC]]* )]
[GROUP BY (<BinaryExpression> [ASC|DESC][, <BinaryExpression> [ASC|DESC]]* )]
[OFFSET <IntegerValue> ROWS | OFFSET <IntegerValue> ROWS FETCH NEXT <IntegerValue> ROWS ONLY] [;]
```

Examples:

- **Normalized Data Model**

  "SELECT ProductID, UnitPrice AS Price FROM Products"

  "SELECT * FROM Orders OFFSET 20 ROWS FETCH NEXT 50 ROWS ONLY"

  "SELECT SUM(UnitsInStock) FROM Products WHERE Discontinued = 'false'"
5.1.2. INSERT Statement

Inserts the specified document into a collection. As the collections in NosDB are schema-less, any number of attributes provided in the values sections will be part of the document being inserted into the collection. The document to insert is of JSON format. All the attributes will be translated as specified into NosDB documents. The attributes can be parameterized. In case of embedded documents, sub-attributes can be specified.

\[
\text{INSERT INTO } \langle\text{Collection}\rangle (\langle\text{Attribute}\rangle [, \langle\text{Attribute}\rangle]*) \\
\text{VALUES } (\langle\text{ConstantBinaryExpression}\rangle [, \langle\text{ConstantBinaryExpression}\rangle]*) \\
[;] \\
\langle\text{ConstantBinaryExpression}\rangle = \text{The binary expression which involves constants only.}
\]

Examples:

- **Normalized Data Model**

  "\text{INSERT INTO } \text{Products} (\text{Name, UnitsInStock, UnitPrice, Discontinued}) \text{VALUES } (\text{'}Eggs\text{'}', 23, 10.50, '\text{True}'\text{')})"

  “\text{INSERT INTO } \text{Orders} (\text{OrderID, CustomerID, OrderDate}) \text{VALUES } (10248, @customer, \text{'2015-07-04 00:00:00.000'})"
### Embedded Data Model

**Insert into Products**

```sql
INSERT INTO Products (ProductID, ProductName, Category) VALUES ('key1', 'Chai', {
  "Name": "Beverages", 
  "Description": "Soft Drinks"
})
```

**Insert into Orders**

```sql
INSERT INTO Orders (OrderID, CustomerID, OrderDate, Freight, OrderDetails) VALUES ('10248','VINET', '1996-07-04 00:00:00.000', 32.38, [{
  "ProductID": 11, 
  "UnitPrice": 14.0, 
  "Quantity": 12, 
  "Discount": "0"
}])
```

#### 5.1.3. UPDATE Statement

Updates those specific attribute(s) of (a) document(s) provided in the JSON format which meet the conditions provided in the WHERE clause. Attributes of arbitrary depths can also be updated by specifying them through the dot notation. In case the attribute does not exist, it will be added to the collection as a new attribute where the conditions of the WHERE clause are satisfied.

```sql
UPDATE <Collection> SET ['(' Attribute = <BinaryExpression> [ , <Attribute> = <BinaryExpression> ])* ',']
[WHERE <Predicate>]
[]
```

**Examples:**

- **Normalized Data Model**

  ```sql
  "UPDATE Products SET (UnitPrice = 19.50) WHERE UnitPrice = 18.00"
  
  "UPDATE Orders SET OrderDate = DateTime('2017-03-03') WHERE OrderDate = DateTime('2000-01-01')"
  
  "UPDATE Orders SET CustomerID = @customer, ShipName = 'Hanari Carnes' WHERE ID = 10248"
  ```

- **Embedded Data Model**

  ```sql
  "UPDATE Products SET (Category.Name = 'Farm Fresh') WHERE Category.Name = 'Drinks'"
  
  "UPDATE Orders SET Order.OrderDetails[0] = @orderdetails WHERE Order.OrderDetails ARRAY SIZE 2"
  
  "UPDATE Orders SET Order.OrderDetails.Discount = 0.75 WHERE Order.OrderDetails.Quantity > 100"
  
  "UPDATE Products SET Category.Name = 'Drinks', UnitPrice = 99 WHERE Discontinued = 'False' AND Category.Name='Beverages'"
  ```
5.1.4. DELETE Statement

Deletes document(s) from a particular collection. The WHERE clause can be used to specify condition of deletion(s).

DELETE
FROM <Collection>
[WHERE <Predicate>]
[;]

Examples:

- **Normalized Data Model**

  "DELETE FROM Orders"
  "DELETE FROM Products WHERE ProductID < 500"
  "DELETE FROM Products WHERE ProductID IN (10, 20, 30)"
  “DELETE FROM attachmentcollection”

- **Embedded Data Model**

  "DELETE FROM Products WHERE Category.Name = 'Beverages'"
  "DELETE FROM Orders WHERE Order.OrderDetails.Discount = 0"
  "DELETE FROM Orders WHERE Order.OrderDetails[0].Quantity = 100"
5.2. SQL Reference

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

```json
{
   "_key": "DA123",
   "ProductID": 10,
   "Name": "milk",
   "Category": "Dairy",
   "UnitsAvailable": 23,
   "PackageSize": ['S', 'M', 'L'],
   "Types": ['Skimmed', 'Full Fat'],
   "TypeBySizeAvailable": [[12, 5], [4, 17], [21, 21]],
   "Orders": {
      "OrderDate": "2017/1/16",
      "OrderID": 20390,
      "OrderSize": ['S', 'L']
   }
}
```

5.2.1. 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>UPDATE</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:

```sql
SELECT "Order" FROM Products WHERE "Order".OrderDate == DateTime('2017, 1, 20')
```

```sql
SELECT $Order$ FROM Products WHERE $Order$.OrderDate == DateTime('2017, 1, 20')
```
5.2.2. 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'+'1e' (string concatenation)</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>

5.2.3. 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 Name 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 Name 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.

5.2.4. 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 Name FROM Products WHERE ProductID == 10&quot;</td>
</tr>
<tr>
<td>!=</td>
<td>Not equals to</td>
<td>&quot;SELECT * FROM Products WHERE ProductID != 10&quot;</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>SQL Example</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>-------------</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 wild card base searching.</td>
<td>&quot;SELECT * FROM Products WHERE Name 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 Name 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 Name 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 Name FROM Products WHERE Orders 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 Name 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 Name IS NOT NULL&quot;</td>
</tr>
</tbody>
</table>
5.2.5. 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>

5.2.6. 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(Name,2,2) FROM Products&quot;</td>
</tr>
</tbody>
</table>
5.2.7. 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:

<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'])&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 through. 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>
<tbody>
<tr>
<td>ADD</td>
<td>Appends the specified values to the array.</td>
<td>&quot;UPDATE Products SET PackageSize ADD ('XL')&quot;</td>
</tr>
<tr>
<td></td>
<td>Checks if the array does not contain any of the</td>
<td>Result: {PackageSize: ['S', 'M', 'L', 'XL']}</td>
</tr>
<tr>
<td>INSERT</td>
<td>specified values.</td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td>Updates existing value with the specified value.</td>
<td>&quot;UPDATE Products SET PackageSize UPDATE ('S'='Small', 'L'='Large')&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result: {PackageSize: ['Small', 'M', 'Large']}</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Removes attribute from array if it exists.</td>
<td>&quot;UPDATE Products SET PackageSize REMOVE ('S')&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result: {PackageSize: ['M', 'L']}</td>
</tr>
</tbody>
</table>

5.2.8. 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</td>
<td>&quot;SELECT Product FROM Products WHERE Category = 10 ORDER BY Category&quot;</td>
</tr>
<tr>
<td></td>
<td>descending or ascending order. In case the query</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is applied on a non-existent attribute, it will</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return an empty result.</td>
<td></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</td>
<td>Retrieves result set with respect to specified</td>
<td>&quot;SELECT Order FROM Products WHERE Orders.OrderDate = DateTime('2017, 1, 20')&quot;</td>
</tr>
<tr>
<td></td>
<td>date time compatible string&quot;</td>
<td></td>
</tr>
<tr>
<td>TOP</td>
<td>Returns the first specified number of records and</td>
<td>&quot;SELECT TOP 10 ProductName FROM Products WHERE ProductID = 10&quot;</td>
</tr>
<tr>
<td></td>
<td>executes query on the specified records.</td>
<td></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'"

### 5.2.9. 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'"
6. Basic Operations through SQL API

In This Chapter:
- ExecuteReader() Method
- ExecuteNonQuery() Method
- ExecuteScalar() Method

NosDB’s flagship mechanism of retrieving and modifying data in a scalable environment is through querying. Thus, it further extends the existing SQL execution methods to query on schema-less JSON documents effectively.

To utilize the following APIs, import the following package in your application:

```
import com.nosdb.client.*;
```

The `Database` class contains the following methods to allow execution of the query:

```
long executeNonQuery(String queryText)
long executeNonQuery(String queryText, Collection<ParameterImpl> parameters)

DBCollectionReader executeReader(String queryText)
DBCollectionReader executeReader(String queryText, Collection<ParameterImpl> parameters)

DBCollectionReader executeReader(String queryText, Collection<ParameterImpl> parameters, ReadPreference readPreference)

Object executeScalar(String queryText)
Object executeScalar(String queryText, Collection<ParameterImpl> parameters)
```

### Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>executeReader</td>
<td>DBCollectionReader</td>
<td>Returns list of key-value pairs in result set which fulfills the query criteria. This key value pair has document key and its respective value. Used in SELECT queries.</td>
</tr>
<tr>
<td>executeNonQuery</td>
<td>long</td>
<td>Used to execute queries containing DELETE and INSERT operations. Returns number of affected rows after query is executed.</td>
</tr>
<tr>
<td>executeScalar</td>
<td>Object</td>
<td>Returns a single value against the query. Especially used for aggregate queries, however, NosDB extends the functionality to allow fetching the first value of the result set in scalar queries as well.</td>
</tr>
</tbody>
</table>
## Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>String</td>
<td>Text of the query to be executed through either of the methods.</td>
</tr>
<tr>
<td>parameters</td>
<td>Collection&lt;ParameterImpl&gt;</td>
<td>Used in case of parametrized queries.</td>
</tr>
<tr>
<td>readPreference</td>
<td>ReadPreference</td>
<td>Determines the user preference on which node the data should be read from. Can be PrimaryOnly, LoadBalanced or Nearest.</td>
</tr>
</tbody>
</table>

### 6.1. ExecuteReader() Method

Used for SELECT queries where the result is returned in a reader to be traversed accordingly. For example, the reader allows methods like `getDocument()` where the documents of the result set are returned to the user.

- **JSON Document**

```java
// precondition: database has been initialized
String query = "SELECT Category.Name, Category.Description FROM Products WHERE Name = 'Chai'";

DBCollectionReader reader = db.executeReader(query);
while (reader.readNext())
{
    JSONDocument data = reader.getDocument(); //return result as document
    //perform operations
}
```

```java
String query = "SELECT Category.Name, Category.Description FROM Products WHERE Name = @name";

java.util.Collection<ParameterImpl> param = new java.util.ArrayList<>();
param.add(new ParameterImpl("name", "Chai"));

DBCollectionReader reader = db.executeReader(query, param, ReadPreference.PrimaryOnly, true);
```
### Custom Objects

```java
// precondition: database has been initialized
String query = "SELECT Category.Name, Category.Description FROM Products WHERE Name = 'Chai';"

DBCollectionReader reader = db.executeReader(query);

while (reader.readNext())
{
    String categoryName = reader.getString("Name");
    String categoryDescription = reader.getString("Description");

    // perform operations
}
```

```java
// precondition: database has been initialized
String query = "SELECT * FROM Products WHERE Name = 'Chai';"

DBCollectionReader<Product> reader = db.executeReader(query);

while (reader.readNext())
{
    Product product = reader.getObject(); // object deserialized into Product class
    // perform operations
}
```

```java
// precondition: database has been initialized
String query = "SELECT Order.OrderDetails FROM Orders";

DBCollectionReader<OrderDetail[]> reader = db.executeReader(query);

while (reader.readNext())
{
    OrderDetail[] orderDetails = reader.getArray("OrderDetails");
    // perform operations
}
```
6.2. **ExecuteNonQuery() Method**

Used for queries which modify the collection, and do not require any database data in return. These include the **INSERT**, **UPDATE** and **DELETE** queries. `executeNonQuery()` returns the number of rows affected by the operation, in case the user wishes to track the success of the operation.

1. **INSERT Query**

```java
//precondition: database has been initialized
String query = "INSERT INTO Products (ProductID, ProductName, Category) VALUES ('key1', 'Chai', {"Name":"Beverages", "Description":"Soft Drinks"})";
long rowsAffected = db.executeNonQuery(query);
```

2. **DELETE Query**

```java
//precondition: database has been initialized
String query = "DELETE FROM Products WHERE Category.Name = 'Beverages'";
long rowsAffected = db.executeNonQuery(query);
```

3. **UPDATE Query**

```java
//precondition: database has been initialized
String query = "UPDATE Products SET(Category.Name = 'Farm Fresh') WHERE Category.Name = 'Dairy'";
long rowsAffected = db.executeNonQuery(query);
```
6.3. **ExecuteScalar() Method**

While `executeScalar()` is used for Aggregate Querying, NosDB also extends the functionality by allowing it for scalar queries, where only the first record of the result set is required. This helps in avoiding the complexity of `executeReader()` altogether.

1. **Aggregate Querying**

```java
//precondition: database has been initialized
String query = "SELECT Count(*) FROM Products WHERE Category.Name = 'Beverages'";
Object count = db.executeScalar(query);
//perform operations
```

2. **Scalar Querying**

```java
//precondition: database has been initialized
String query = "SELECT Name FROM Products WHERE Category.Name = 'Beverages'";
Object name = db.executeScalar(query);
//perform operations
```
7. Using NCache with NosDB

In This Chapter:

NCache Supported Features
Integrating NCache Provider
Cache Queries Using Configured NCache Provider

This feature is only available in NosDB Enterprise Edition.

NCache by Alachisoft is a flexible and feature rich .NET caching solution that provides high performance and scalability to handle any transaction load. NCache features and APIs are designed to cache data for applications of any size – from small to enterprise-wide global installations.

NCache is a trusted clustered caching solution that makes sharing and managing data in a cluster as simple as on a single server. It accomplishes this by coordinating updates to the data using cluster wide concurrency control and replicating and distributing data modifications across the cluster using the highest performing clustered protocol available. The primary purpose of NCache is to help improve performance of .NET applications that would otherwise make expensive trips across the network.

For more detail about NCache, please refer to its online documentation.

NosDB supports integration of NCache into client applications where frequently used data is cached along with being persisted to the file storage.

- **Installation**

NCache for local caching is bundled with NosDB for free. In case you wish to use clustered caching, you need to purchase NCache or use its trial version.

Before installing NCache Client, make sure that NosDB has already been installed. The installation steps for NCache are documented here.
7.1. **NCache Supported Features**

- **Caching**
  
  The NCache client can be used as either a local in-proc (in-process) or local out-proc (out-of-process) cache, which means that the cache resides on a single node. Caching is disabled by default. It will be enabled through configurations as explained later in the chapter. Documents are inserted/fetched to/from the cache with atomic and bulk APIs and querying if caching is enabled.

- **NosDB Database Dependency**
  
  The user can enable database dependency on the documents being cached to NCache through configurations explained later in the chapter. Documents are cached with database dependency - any change in the database store will remove the document from the cache. Once an item is added to the database, it checks if the SQLDependency flag is enabled or not. If the flag was set to true, and the item has been added to NosDB database successfully, the item is added to the cache as well. NCache then handles registering database dependency with NosDB without any user intervention, where the item is added internally with an object of SQLDependency to the cache.

  In case any change is made to the data in the database with dependency registered, the corresponding item in the cache will be invalidated and removed. This ensures consistency of data in cache and database. The next time a client tries to access that item from the cache, it will not find any item with that key in the cache and consequently fetch it from the database. This way the most updated item will be stored in the local cache.

- **Expiration**
  
  NCache supports time-based data expiration with user-specified time interval to expire the data from the cache. Expiration is used if changes to the data occur in a deterministic time fashion or frequency. For example, customer information or region information may not be frequently updated, so expiration can be used to invalidate such data.

  The query result set can be added to the cache with expiration, which can also be specified by the API. There are two expiration strategies: [Sliding and Absolute](#). If the expiration type is not specified, NosDB will add the item with Absolute expiration of 5 minutes by default.

- **Querying**
  
  A specific query result set from NosDB will be cached in NCache based on the optional API parameter with `ExecuteReader()` API. The result set will be cached as a single item in the cache, with the query as its key.

  Any change in the result set, either through querying or APIs, will not affect the items in the cache. The item will only be removed from the cache once expiration takes place. Refer to [Cache Queries](#) to programmatically cache queries in NCache.
7.2. Integrating NCache Provider

NosDB’s custom NCache provider can be integrated in the application without any code changes. All it requires is a modification in the Distributor Service’s configuration file, along with a reference to the provider assembly.

Prerequisites

The following prerequisites will enable integration of NCache Provider in your application:

- Any .NET (4.0/4.5/4.6) application with reference to the provider assembly:
  
  `[InstallDir]\integrations\NCache Provider\NosDB.NCacheProvider.dll`

- You can also use the sample application for using NCache located in:
  
  `[InstallDir]\samples\applications\CachingQueries`

- Microsoft Visual Studio (any version)

- Database (NosDB)

Modifying DistributorService.exe.config

You can specify the cache information and enable SQLDependency and expiration through NosDB.DistributorService.exe.config located in `[InstallDir]\bin\service`. Add the following configuration tags within the `<configuration>` tag in the file:

```xml
<configuration>
  
  <configSections>
    <section name="cacheConfig"
      type="NosDB.Common.Caching.CacheConfigSection,NosDB.Common, Version=2.0.0.0, Culture=neutral, PublicKeyToken=8a1e00327893b9ef" allowLocation="true" allowDefinition="Everywhere" />
  </configSections>

  <cache cache-id="demoCache" cache-behavior="cache-all | cache-none | cache-doc [default] | cache-query" expiration-type="absolute" expiration-interval="10sec" enable-sqldependency="true">
    
    <collections>
      <collection name="products" cache-behavior="cache-all | cache-none | cache-doc | cache-query" expiration-type="absolute" expiration-interval="10sec" enable-sqldependency="true"/>
      <collection name="customers" cache-behavior="cache-all | cache-none | cache-doc | cache-query" expiration-type="absolute" expiration-interval="10sec" enable-sqldependency="true"/>
    </collections>
  </cache>

</configuration>
```
• `<configSections>`

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The human-understandable name of the connection string which will be passed through the application while connecting to the database.</td>
</tr>
<tr>
<td>type</td>
<td>The fully qualified name of the class which implements the <code>cacheConfig</code> section.</td>
</tr>
</tbody>
</table>

• `<cacheConfig>`

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache-id</td>
<td>The name for the local cache configured in NCache. The entry for the same cache should exist in <code>client.ncconf</code> of the client machine.</td>
</tr>
</tbody>
</table>
| cache-behavior | Specifies what should be cached:  
  • `cache-none`: data will not be cached.  
  • `cache-doc`: caches only the documents that are fetched or inserted from Document API.  
  • `cache-query`: caches the result sets generated in response to `SELECT` statements executed on ExecuteReader API.  
  • `cache-all`: cache both documents and result set. |
| expiration-type | The type of expiration of data from the cache, which can be Absolute or Sliding. By default, Absolute Expiration is considered. |
| expiration-interval | Time in seconds after which the item is expired from cache. |
| enable-sqldependency | Boolean value to enable/disable SQLDependency of cache with NosDB. |
| name     | Name of the collection being cached. |
7.3. Using Configured NCache Provider

To utilize NosDB’s NCache Provider, import the following package in your application:

```java
import com.nosdb.client.*;
```

7.3.1. Caching Queries

Once NCache Provider has been integrated, the query result from NosDB can be cached into the local cache instance configured in NCache. The `DBCollectionReader` `executeReader` method contains overloads that allow specifying whether the result should be cached into NCache or not, along with providing the expiration specifications.

```
DBCollectionReader executeReader(String queryText, Collection<ParameterImpl> parameters, ReadPreference readPreference, boolean cacheResult)

DBCollectionReader executeReader(String queryText, Collection<ParameterImpl> parameters, ReadPreference readPreference, boolean cacheResult, ExpirationType cacheExpiration, TimeSpan expiryTime)
```

### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queryText</td>
<td>String</td>
<td>Text of the query to be executed and cached.</td>
</tr>
<tr>
<td>parameters</td>
<td>Collection&lt;ParameterImpl&gt;</td>
<td>Used in case of parametrized queries.</td>
</tr>
<tr>
<td>readPreference</td>
<td>ReadPreference</td>
<td>Determines the user preference on which node the data should be read from. Can be PrimaryOnly, LoadBalanced or Nearest.</td>
</tr>
<tr>
<td>cacheResult</td>
<td>boolean</td>
<td>To determine whether the data should be cached in NCache.</td>
</tr>
<tr>
<td>cacheExpiration</td>
<td>ExpirationType</td>
<td>The type of expiration of data from the cache, which can be Absolute or Sliding. By default, Absolute Expiration is considered.</td>
</tr>
<tr>
<td>expiryTime</td>
<td>TimeSpan</td>
<td>The time duration for expiration. In Absolute, the item will be cached after the expiryTime is reached. In Sliding, the time will be extended if the item is accessed in the cache before the expiryTime is over.</td>
</tr>
</tbody>
</table>
- **Sample Implementation of Caching Query without Expiration**

```java
String query = "SELECT Category.Name, Category.Description FROM Products WHERE Name = @name";
java.util.Collection<ParameterImpl> param = new java.util.ArrayList<>();
param.add(new ParameterImpl("name", "Chai"));

DBCollectionReader reader = db.executeReader(query, param, ReadPreference.PrimaryOnly, true);
```

- **Sample Implementation of Caching Query with Sliding Expiration**

```java
String query = "SELECT Category.Name, Category.Description FROM Products WHERE Name = @name";
java.util.Collection<ParameterImpl> param = new java.util.ArrayList<>();
param.add(new ParameterImpl("name", "Chai"));

//Cache the resultset into NCache with Sliding Expiration of 5 minutes
DBCollectionReader reader = db.executeReader(query, param, ReadPreference.PrimaryOnly, true, ExpirationType SlidingExpiration, new TimeSpan(0, 5, 0));
```
7.4. **Caching Behaviors**

There are multiple scenarios where documents and query results are either inserted, fetched or deleted from NCache.

7.4.1. **Cache Fetches**

Documents and query results are first searched for in the cache and then served by the database in case no result is found. The data is fetched from the cache in the following scenarios:

1. When either of the overloaded methods for `getDocuments()` are called with value of 'useCache = true', a bulk GET request on cache is called for all the required documents. Only the documents that are not found in the cache are then called on the database.
2. When a `SELECT` statement is executed on `ExecuteReader` API, the complete result set is first looked up in the cache. If found, it is served as `DBCollectionReader`, otherwise, it is served from the database.

7.4.2. **Cache Inserts**

Documents and query results are inserted into the cache in the following scenarios:

1. When a document is inserted into the database through Document API.
2. When document is inserted into the database through SQL interface.
3. When a document is fetched from the database through Document API. This is a cache miss scenario.
4. When multiple documents are inserted into the database through Document API, each document is individually inserted into the cache.
5. When multiple documents are fetched from the database through Document API, each document is individually inserted into the cache. This is a cache miss scenario.
6. When a `SELECT *` statement with search criteria on PK only is executed on database which is expected to return at most one document, the document from result set is cached in form of complete document instead of the result set.
7. When a `SELECT` statement is executed and the Data Reader is exhausted by client API, a complete result set is inserted into the cache as a single object.

7.4.3. **Cache Deletes**

The specified keys are deleted from the cache in the following scenarios:

1. When the expiry time for cached item is reached.
2. When the cache is full, cached items are removed from the cache based on configured cache eviction policy.
3. When following methods are called, the specified keys are removed from cache:
   - `DeleteDocument()`
   - `DeleteDocuments()`