# Table of Contents

1. Introduction to NosDB ................................................................. 4  
   1.1. Edition Comparison .............................................................. 5  

## Installing NosDB ........................................................................ 7  
2. Database Server Requirements .................................................... 8  
   2.1. Hardware Requirements ......................................................... 8  
   2.2. Software Requirements .......................................................... 8  
3. Install NosDB ........................................................................... 9  
   3.1. Interactive Windows Installation ............................................. 9  
   3.2. Scripted Installation .............................................................. 20  

## Getting Started with NosDB .......................................................... 21  
4. Open Firewall TCP Port for Clients ............................................. 22  
   4.1. Configuration Server Port ....................................................... 22  
   4.2. Management Port ............................................................... 22  
5. Accessing Database Cluster across Domains .................................. 23  
   5.1. Using Runas Command ......................................................... 23  
6. Configuring and Using PowerShell in NosDB .............................. 24  
   6.1. Load NosDB Assembly ......................................................... 24  
   6.2. To Run Scripts ................................................................... 24  
   6.3. Navigating in PowerShell Console ....................................... 25  
7. Using NosDB Stress Test Tool ..................................................... 28  

## Configuring Database Clusters and Shards ..................................... 29  
8. Database Clusters ..................................................................... 30  
   8.1. Create Database Cluster ....................................................... 30  
   8.2. Connect to Existing Database Cluster .................................... 31  
   8.3. Disconnect from Cluster ...................................................... 32  
   8.4. Remove Database Cluster ................................................... 32  
9. Shards ...................................................................................... 33  
   9.1. Add Shard ........................................................................ 33  
   9.2. Remove Shard .................................................................... 34  
   9.3. Start Shard ........................................................................ 35  
   9.4. Stop Shard ........................................................................ 35  

## Configuring Databases ................................................................ 36  
10. Databases .............................................................................. 37  
   10.1. Create New Database ......................................................... 37  
   10.2. Configure Database Properties ........................................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.3. Drop Database</td>
<td>38</td>
</tr>
<tr>
<td>11. Collections</td>
<td>39</td>
</tr>
<tr>
<td>11.1. Create Collection</td>
<td>39</td>
</tr>
<tr>
<td>11.2. Drop Collection</td>
<td>39</td>
</tr>
<tr>
<td>12. Indexes</td>
<td>40</td>
</tr>
<tr>
<td>12.1. Create Index</td>
<td>40</td>
</tr>
<tr>
<td>12.2. Drop Index</td>
<td>40</td>
</tr>
<tr>
<td><strong>Configuring Security</strong></td>
<td>41</td>
</tr>
<tr>
<td>13. Security Concepts</td>
<td>42</td>
</tr>
<tr>
<td>13.1. Logins</td>
<td>42</td>
</tr>
<tr>
<td>13.2. Roles and Users</td>
<td>42</td>
</tr>
<tr>
<td>14. Logins</td>
<td>44</td>
</tr>
<tr>
<td>14.1. Add Login</td>
<td>44</td>
</tr>
<tr>
<td>14.2. Drop Login</td>
<td>45</td>
</tr>
<tr>
<td>15. Roles and Users</td>
<td>46</td>
</tr>
<tr>
<td>15.1. Cluster Roles</td>
<td>46</td>
</tr>
<tr>
<td>15.2. Database Roles</td>
<td>46</td>
</tr>
<tr>
<td><strong>Database Administration</strong></td>
<td>48</td>
</tr>
<tr>
<td>16. Backup Database</td>
<td>49</td>
</tr>
<tr>
<td>16.1. Distributed vs. Consolidated Storage</td>
<td>49</td>
</tr>
<tr>
<td>16.2. Backup Database</td>
<td>49</td>
</tr>
<tr>
<td>17. Restore Database</td>
<td>50</td>
</tr>
<tr>
<td>18. Get Task Information</td>
<td>50</td>
</tr>
<tr>
<td>19. Import Data</td>
<td>51</td>
</tr>
<tr>
<td>19.1. Import CSV Data</td>
<td>51</td>
</tr>
<tr>
<td>19.2. Import JSON Data</td>
<td>53</td>
</tr>
<tr>
<td>20. Export Data</td>
<td>54</td>
</tr>
<tr>
<td>20.1. Export CSV Data</td>
<td>54</td>
</tr>
<tr>
<td>20.2. Export JSON Data</td>
<td>55</td>
</tr>
<tr>
<td>21. Export Cluster Configuration</td>
<td>56</td>
</tr>
<tr>
<td>22. Using SQL Queries</td>
<td>57</td>
</tr>
<tr>
<td>22.1. SQL Queries in PowerShell</td>
<td>57</td>
</tr>
<tr>
<td>22.2. Invoking SQL Scripts</td>
<td>58</td>
</tr>
<tr>
<td>23. Move Collection among Shards</td>
<td>59</td>
</tr>
<tr>
<td>24. Get NosDB Version Information</td>
<td>59</td>
</tr>
<tr>
<td><strong>Database Monitoring</strong></td>
<td>60</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>25.</td>
<td>NosDB Counters</td>
</tr>
<tr>
<td>27.</td>
<td>Windows Event Logs</td>
</tr>
<tr>
<td>28.</td>
<td>NosDB Logs</td>
</tr>
<tr>
<td>29.</td>
<td><strong>Database Programming with NosDB .NET API</strong></td>
</tr>
<tr>
<td>29.1.</td>
<td>Setting Up Visual Studio Environment</td>
</tr>
<tr>
<td>29.2.</td>
<td>Reference NosDB Assemblies</td>
</tr>
<tr>
<td>29.3.</td>
<td>Specify Namespaces</td>
</tr>
<tr>
<td>30.</td>
<td>Using NosDB NuGet Package</td>
</tr>
<tr>
<td>30.1.</td>
<td>Database Connection</td>
</tr>
<tr>
<td>31.</td>
<td>Basic Operations through Document API</td>
</tr>
<tr>
<td>31.1.</td>
<td>Using NosDB Collection</td>
</tr>
<tr>
<td>31.2.</td>
<td>Insert Documents</td>
</tr>
<tr>
<td>31.3.</td>
<td>Get Documents</td>
</tr>
<tr>
<td>31.4.</td>
<td>Update Documents</td>
</tr>
<tr>
<td>31.5.</td>
<td>Delete Documents</td>
</tr>
<tr>
<td>32.</td>
<td>Querying in NosDB</td>
</tr>
<tr>
<td>32.1.</td>
<td>Data Definition Language (DDL) Support</td>
</tr>
<tr>
<td>32.2.</td>
<td>Data Control Language (DCL) Support</td>
</tr>
<tr>
<td>32.3.</td>
<td>Data Manipulation Language (DML) Support</td>
</tr>
<tr>
<td>32.4.</td>
<td>Query Syntax in NosDB</td>
</tr>
<tr>
<td>33.</td>
<td>Basic Operations through SQL API</td>
</tr>
<tr>
<td>33.1.</td>
<td>ExecuteReader() Method</td>
</tr>
<tr>
<td>33.2.</td>
<td>ExecuteNonQuery() Method</td>
</tr>
<tr>
<td>33.3.</td>
<td>ExecuteScalar() Method</td>
</tr>
<tr>
<td>34.</td>
<td><strong>Database Programming using NosDB ADO.NET Provider</strong></td>
</tr>
<tr>
<td>34.1.</td>
<td>Using ADO.NET with NosDB</td>
</tr>
<tr>
<td>34.2.</td>
<td>Setup Visual Studio Environment</td>
</tr>
<tr>
<td>34.3.</td>
<td>Connect to Database</td>
</tr>
<tr>
<td>34.4.</td>
<td>Basic Operations through SQL API</td>
</tr>
<tr>
<td></td>
<td>Binding UI Controls with NosDB through ADO.NET</td>
</tr>
</tbody>
</table>
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>monospace</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>
## 1.1. Edition Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>Open Source</th>
<th>Professional</th>
<th>Enterprise</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client API</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- .NET API</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Java API</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- REST API</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Data Model</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- JSON Documents</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Normalized Data Model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Embedded Data Model</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Attachments</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- CLR Triggers</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- CLR User Defined Functions (UDF)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Clustered (Sharded) Databases</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Stand-alone Database</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Sharded Database (no replication)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Sharded Database (with replication)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Stand-alone to Sharded DB Migration</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Hash Based Sharding</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Range Based Sharding</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Collections</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Single-Shard Collections</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Capped Single-Shard Collections</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Multi-Sharded Collections</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Move Single-Shard Collection Tool</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>High Availability</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Self-Healing Dynamic Cluster</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Configuration Manager (stand-alone)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Configuration Manager (clustered)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Shard Replicas (dedicated &amp; shared)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- State Transfer Across Shards</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Journaling (for data consistency)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Multi-Data Center Support</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- NosDB Upgrade without Downtime</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Performance &amp; Scalability</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Multiple Shards</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- DB Server In-Memory Caching</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- DB Client In-Memory Caching (NCache)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>(with InProc Client Cache)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Reads also Run on Replicas</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Async Operations (with Fire &amp; Forget)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Bulk Operations</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>SQL Support</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Basic SQL (SELECT, INSERT, UPDATE, DELETE)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Feature</td>
<td>Open Source</td>
<td>Professional</td>
<td>Enterprise</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------</td>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>- Advanced SQL (with JSON)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- LINQ</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Database Management (DDL)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Indexing (Basic)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Indexing (Compound)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>- Sql Dependency (DB Notifications on Updates)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Big Data Processing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- MapReduce</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>- Aggregator</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Admin &amp; Monitoring</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- NosDB Management Studio (with SQL Studio)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- NosDB Monitor (GUI)</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>- PerfMon Counters</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Windows Event Logs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- PowerShell Integration</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Server Mgmt API (.NET, Java, REST)</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>- Full Backup/Restore</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Differential Backup/Restore</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>- Import/Export</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Windows Authentication</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- NosDB Authentication</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>- Authorization</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- SSL Support</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Third-Party Integrations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Visual Studio 2013/2015</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>- Microsoft Power BI (Analytics)</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>- ADO.NET Provider</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- NuGet Package</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Cloud Platforms Supported</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Microsoft Azure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Amazon Web Services (AWS)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Installation Package</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Windows Installer Client &amp; Server (.msi)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Java Client Installer (.msi, .tar.gz)</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Installing NosDB

NosDB OpenSource Guide
## 2. Database Server Requirements

NosDB provides a Windows Installer Package (.MSI) to let you easily install NosDB. However, before you install NosDB, make sure the following system requirements have been fulfilled:

### 2.1. Hardware Requirements

- **Dual-CPU, Quad-Core or higher**
  NosDB is a highly multi-threaded software and takes full advantage of extra cores and CPUs. The most common configuration for NosDB is a dual-CPU quad-core machine. You may need stronger processing power if you have higher transaction loads and/or larger amount of data being stored in NosDB.

### 2.2. Software Requirements

- **Windows Server 2008 R2, Windows Server 2012 R2 (64-bit)**
- **.NET Framework 4.0/4.5/4.6**
- **Windows 7/8.1/10** (These versions of Windows are not highly recommended for production.)
- **Minimum 8 GB RAM in Servers**
  NosDB works efficiently with 8 GB RAM, however 16 GB RAM servers are highly recommended for optimum performance.
- **Minimum 160 GB Disk Space**
  For installation, the minimum recommended disk space required is 160 GB. The disk space for usage, however, depends on the scale on which NosDB is being used.

In case you wish to get specific recommendations for your needs, please consult [Alachisoft’s technical support](#).
3. Install NosDB

*In This Chapter:*

Interactive Windows Installation

Scripted Installation

3.1. Interactive Windows Installation

NosDB can also be installed via an interactive Windows installer.

We recommend you to run your evaluation using the NosDB Enterprise Edition. You can later decide whether you want to downgrade to NosDB Professional or not, based on your needs.

- Right-click on the `.MSI` file `NosDB_1.3_OpenSource_DotNet_x64.msi` and select *Run as Administrator*.

- The following *Welcome to the NosDB Installation Wizard* screen appears:

![Welcome to the NosDB Installation Wizard](image)

- Click *Next*.

- The next screen asks you to accept the license agreement. By clicking *I accept the terms in the License Agreement*, you are making a legal commitment to abide by the Alachisoft Software License Agreement for NosDB. You cannot proceed without accepting the license agreement.
• **Evaluation Information** dialog appears which asks for the installation key. This is the key which you have received from Alachisoft.
If you do not have an installation key, please click on **Get Installation Key** button which takes you to Alachisoft website to register and obtain an installation key.

Even when you have purchased NosDB and have a license key, you must provide the INSTALL KEY here (and not the purchased license key). You will get an opportunity later to specify your purchased license key when you activate your licenses.

If you are already registered, just enter your email address and an installation key will be emailed to you. Otherwise, register by specifying information to receive the installation key via email.
In **User Information** dialog, enter your name, company name, and email address. This information will be displayed as part of NosDB to indicate your identity.

- Click Next.
By default, NosDB is installed in "C:\Program Files\NosDB" folder. However, if you want to change this location, you can specify it in the next window.

Click Next.
In the **Select features** window, you can see all the components that will be installed automatically. You can unselect some of the components if you wish. However, it is strongly recommended that you keep everything as default.
• If you wish to change the default installation features, click on the drop down menu next to each feature and select the desired options.
- To view the disk space in your system, click on **Disk Cost**.

- **Disk Space Requirements** appears, displaying the available and required disk space, and the difference between the two. In case the disk space is not enough, the volumes will be highlighted.

- Click **OK** to go back to the **Select Features** window.

- After selection of the features, click **Next**.

- **NosDB Configuration** screen appears. Provide the directory where you wish to store the data files, index files, configuration files and journals.
• You will then be prompted to start installation process by clicking the **Install** button.
• Installation process will begin.
• After successful installation, the following success screen appears.

![NosDB OpenSource .NET Edition Setup](image)

NosDB has been successfully installed
Click the Finish button to exit this installation.

• Click **Finish** to exit the installation setup.

• The **Getting Started with NosDB** page opens up locally in your browser, which guides on getting started with NosDB in your applications.
3.2. Scripted Installation

NosDB provides a command line installation that you can either run from Command Prompt or add to your script files (.bat files). To install NosDB from Command Prompt, use the `msiexec.exe` utility.

- Open **Command Prompt** with administrative privileges.
- Enter the following command after altering it according to your requirements:

```
msiexec.exe /I "<Installation Package Path>" KEY="INSTALL_KEY_EMAILED_TO_YOU"
USERFIRSTNAME="FIRSTNAME" USERLASTNAME="LASTNAME" COMPANYNAME="COMPANYNAME"
EMAILADDRESS="EMAILADDRESS" DATADIR="<Target Dir>" IPADDRESS="NODE_IP" /qn
```

- Press **Enter**.
- The installation process starts. Once the installation process is complete, NosDB will be successfully installed in the default directory (`C:\Program Files\NosDB`).

<table>
<thead>
<tr>
<th>Switch/Properties</th>
<th>Purpose</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/I &lt;.msi Package Path&gt;</td>
<td>Location where the <code>NosDB_1.3_OpenSource_DotNet_x64.msi</code> file has been downloaded from the Alachisoft website.</td>
<td>Empty</td>
</tr>
<tr>
<td>KEY (Required)</td>
<td>Installation Key: You will have received this via email from Alachisoft when you downloaded NosDB. If not, then visit <a href="#">Alachisoft website</a> to register and obtain one.</td>
<td>Empty</td>
</tr>
<tr>
<td>USERFIRSTNAME</td>
<td>User’s first name</td>
<td>**</td>
</tr>
<tr>
<td>USERLASTNAME</td>
<td>User’s last name</td>
<td>**</td>
</tr>
<tr>
<td>COMPANYNAME</td>
<td>User’s company name</td>
<td>**</td>
</tr>
<tr>
<td>EMAILADDRESS (Required)</td>
<td>User’s email address</td>
<td>**</td>
</tr>
<tr>
<td>PASSWORD (Required)</td>
<td>Password for the admin account (admin).</td>
<td>-</td>
</tr>
<tr>
<td>IPADDRESS</td>
<td>IP address of the node to host server.</td>
<td>Local IP</td>
</tr>
<tr>
<td>/qn</td>
<td>Quiet mode: Installs NosDB without any user intervention.</td>
<td>Empty</td>
</tr>
</tbody>
</table>
Getting Started with NosDB

NosDB Open Source Guide
4. Open Firewall TCP Port for Clients

4.1. Configuration Server Port

If a firewall is enabled between the clients and the configuration server, then you must open a TCP port (default 9950) in the firewall for the client to communicate with the configuration server through the connection string.

"Data Source=20.200.20.44; Port=9950; Database=northwind; Local Instance=false;"

If you want to manually configure the Config Server Port, changes must be made on all servers running the Configuration Service. The port can be modified in the "Alachisoft.NosDB.ConfigurationService.exe.config" file located in "%INSTALL_DIR%/bin/service":

```xml
<appSettings>
  ...  
  <add key="ConfigServerPort" value="9950"/>
</appSettings>
```

4.2. Management Port

The configuration server further communicates with the Database Service using the Management Port. The default Management Port for the Database Service is 9960. If you want to manually configure the Management Server Port, changes must be made on all servers running the Configuration Service. The port can be modified in the "Alachisoft.NosDB.ConfigurationService.exe.config" file located in "%INSTALL_DIR%/bin/service":

```xml
<appSettings>
  ...  
  <add key="ManagementServerPort" value="9960"/>
</appSettings>
```

However, this port is being used by the database service so it also has to be modified accordingly in the "Alachisoft.NosDB.DatabaseService.exe.config" file located in "%INSTALL_DIR%/bin/service":

```xml
<appSettings>
  ...  
  <add key="ManagementServerPort" value="9960"/>
</appSettings>
```

Every time the configuration files (of Configuration Service and Database Service) are modified, the corresponding services must be restarted.
5. Accessing Database Cluster across Domains

To access a cluster across a domain, different permissions are required respective to the domain. This can also result in restricted access because of Windows Authentication, along with a similar limitation to view the PerfMon counters across a different domain.

5.1. Using Runas Command

Keeping in mind the aforementioned limitations, NosDB has incorporated the Runas command-line tool by Microsoft into its management modules so to access the cluster across domains, the management module specified with a “(Runas)” can be launched and used with ease:

- Launch `psmgmt.runas.cmd` from `%INSTALL_DIR%\bin\tools`.
- Enter the User ID of the login you want to access in the format `user@domain` or `domain\user`.
- Enter the password for the ID.
- You can now run the commands specific to the domain with permissions granted to the user you have logged on as.

Monitoring PerfMon Counters across Domains

- Follow the steps to access a database cluster as mentioned above.
- Run the following command through Command Prompt to launch PerfMon:

```
runas /env /netonly /user:user@domain perfmon.exe
```

- You can now monitor PerfMon Counters for a cluster in the domain you are logged in to.
6. Configuring and Using PowerShell in NosDB

NosDB provides integration with Windows PowerShell to easily automate NosDB processes on your network. You can manage NosDB remote tasks on your network through a single computer using PowerShell scripts.

NosDB PowerShell Provider is only compatible with **PowerShell 4.0 and above**.

6.1. Load NosDB Assembly

- Search for **NosDB PowerShell Management** and Run as Administrator.
- You will directed to NosDB's PowerShell module, indicated by the `PS` at the start of the command line.
- In order to work in NosDB environment through PowerShell, enable PowerShell script execution by setting the **ExecutionPolicy** to RemoteSigned.

```
Set-ExecutionPolicy RemoteSigned
```

Once the NosDB environment has been set up, you can now create a new database cluster or connect to an existing one to carry out NosDB’s tasks through PowerShell.

6.2. To Run Scripts

Once the cluster has been created, you can load an existing Northwind database provided in NosDB samples at `%INSTALL_DIR%\samples\data\json\northwind`. The sample contains JSON format files against specific collections which can be imported into the configured database to get you started with NosDB features.

The provided "**NorthWind.ps1**" script in the same location contains all steps - creating a database, collections and importing the sample northwind data into them. The script is ready for execution:

- Make sure you are connected to the cluster and within the context "NosDB:\$cluster\".
- Execute either of the following commands in PowerShell, based on your accessibility:
  - If you are accessing NosDB cluster from a remote client:
    ```
    Connect-DATABASECLUSTER -Server [cluster-IP] -Port [Port]
    ```
  - If you are accessing NosDB cluster from the local machine:
    ```
    Connect-DATABASECLUSTER
    ```

Execute the following command in the context, starting with the `&` sign:
6.3. Navigating in PowerShell Console

6.3.1. Databases

To view configured database information, switch to the context `databases` in NosDB PSDrive.

```powershell
PS NosDB: \cluster\databases>
```

Typing `dir` will display all configured databases and the storage provider selected against it.

```powershell
PS NosDB: \cluster\databases> dir
```

<table>
<thead>
<tr>
<th>Database name</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>alachisoft</td>
<td>LMDB</td>
</tr>
<tr>
<td>database1</td>
<td>LMDB</td>
</tr>
<tr>
<td>northwind</td>
<td>LMDB</td>
</tr>
</tbody>
</table>

6.3.2. Collections

To view configured collections of a specific database, change context from `databases` to `[database-name].`

```powershell
PS NosDB: \cluster\databases\[database-name]>
```

- Switch context from `[database-name]` to `collections`.

```powershell
PS NosDB: \cluster\databases\[database-name]\collections>
```

- Typing `dir` shows the configured collections and their types in the database.

```powershell
PS NosDB: \cluster1\databases\northwind\collections> dir
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>categories</td>
<td>Normal</td>
<td>HashBased</td>
</tr>
<tr>
<td>customers</td>
<td>Normal</td>
<td>HashBased</td>
</tr>
<tr>
<td>employees</td>
<td>Normal</td>
<td>HashBased</td>
</tr>
<tr>
<td>employee territories</td>
<td>Normal</td>
<td>HashBased</td>
</tr>
<tr>
<td>order-details</td>
<td>Normal</td>
<td>HashBased</td>
</tr>
<tr>
<td>orders</td>
<td>Normal</td>
<td>HashBased</td>
</tr>
<tr>
<td>products</td>
<td>Normal</td>
<td>HashBased</td>
</tr>
</tbody>
</table>
### 6.3.3. Indexes

To view configured indices of a specific database, switch context from collections to indices.

```
PS NosDB: \cluster \databases \[database-name] \collections \[collection-name] \indices
```

- Typing `dir` displays the configured indices in the collection.

```
PS NosDB: \cluster1 \databases \northwind \collections \customers \indices> dir
Context: NosDB: \cluster1 \databases \northwind \collections \customers \indices
Indices
----
customerid
customerid_country
```

### 6.3.4. Logins

To view all added logins, switch context from cluster -> security -> logins.

```
PS NosDB: \cluster \security \logins
```

- Typing `dir` displays all of the configured logins with roles assigned to them.

```
PS NosDB: \cluster1 \security \logins> dir
Context: NosDB: \cluster1 \security \logins
login Roles
---- ---
admin Server Roles (sysadmin)
        Cluster Roles (clusteradmin, clustermanager, dbcreator)
        Database Roles:
        northwind (db_owner, db_admin, db_user)
alachisoft\john_smith Server Roles (sysadmin)
        Cluster Roles (clusteradmin, clustermanager, dbcreator)
        Database Roles:
        northwind (db_owner, db_admin, db_user)
NT SERVICE\NosConfSvc Server Roles (sysadmin)
        Cluster Roles (clusteradmin, clustermanager, dbcreator)
NT SERVICE\NosDBSvc Server Roles (distributor)
```

### 6.3.5. Users

- **Cluster Users**

To view cluster users, switch context from clusters -> security -> users.

```
PS NosDB: \cluster \databases \[database-name] \collections \[collection-name] \indices
```
Typing `dir` displays the configured users and their respective cluster roles.

```
PS NosDB: \cluster1> cd security
PS NosDB: \cluster1\security> cd users
PS NosDB: \cluster1\security\users> dir
```

- **Database Users**

To view database users, switch context from `clusters -> databases -> $database-name$ -> users`.

```
PS NosDB: \cluster\databases\[database-name]\users>
```

Typing `dir` displays the configured users and their respective database roles.

```
PS NosDB: \cluster1> cd databases
PS NosDB: \cluster1\databases> cd northwind
PS NosDB: \cluster1\databases\northwind> cd users
PS NosDB: \cluster1\databases\northwind\users> dir
```
7. Using NosDB Stress Test Tool

Once the database has been configured, NosDB PowerShell allows testing the stress performance of the connected database. Test-Stress creates a test collection named "<shardName>_nosdb_test_coll" and performs multiple Insert, Update, Delete and Get operations on it from single or multiple clients. This command only works in the context of the specified database.

PS NosDB:\cluster\databases\[database-name]>

- To kill the Test-Stress program, enter the keyboard combination Ctrl+C.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-ThreadCount]</td>
<td>Number of client threads</td>
</tr>
<tr>
<td>[-TestCaseIterations]</td>
<td>Number of iterations in a test case</td>
</tr>
<tr>
<td>[-TestCaseIterationDelay]</td>
<td>Delay between each iteration of the test case</td>
</tr>
<tr>
<td>[-GetsPerIteration]</td>
<td>Number of Get operations in one iteration of the test case</td>
</tr>
<tr>
<td>[-UpdatesPerIteration]</td>
<td>Number of Update operations in one iteration of the test case</td>
</tr>
<tr>
<td>[-MaxDocuments]</td>
<td>Maximum number of documents to perform test on. By default, the maximum number is 10,000 and it cannot be exceeded.</td>
</tr>
<tr>
<td>[-DeletesPerIteration]</td>
<td>Number of Delete operations in one iteration of the test case</td>
</tr>
<tr>
<td>[-ReportingInterval]</td>
<td>To specify the interval after which the stats are reported</td>
</tr>
<tr>
<td>[-TotalIteration]</td>
<td>The number of times the test case is to be executed</td>
</tr>
<tr>
<td>[-DropCollection]</td>
<td>Deletes the test collection from the database</td>
</tr>
</tbody>
</table>

Examples:

- This command performs stress test on northwind database with default configurations and total iterations are infinite.

Test-Stress

- This command performs stress test on northwind database with 3 client threads and reports status after 1000 iterations. The rest of the settings are all defaults. Total iterations are infinite.

Test-Stress -ThreadCount 3 -ReportingInterval 1000

- Database is northwind, reporting interval is 2000, number of iterations in each test case is 2. Delay between iterations is 1 sec. Gets per iteration are 10; updates per iteration are 10; deletes per iteration are 10; Total iterations are infinite.

Test-Stress -TestCaseIterations 2 -TestCaseIterationDelay 1 -GetsPerIteration 10 -UpdatesPerIteration 10 -DeletesPerIteration 10 -ReportingInterval 2000 -TotalIteration 200000
Alachisoft

Configuring Database Clusters and Shards

NosDB Open Source Guide
8. Database Clusters

8.1. Create Database Cluster

New-DatabaseCluster creates a database cluster. However, a database cluster must contain a shard with at least one node.

Before creating a database cluster, the following pre-conditions must be fulfilled:

- All machines specified as shard nodes must have both the database and configuration services running.
- These machines must **not** be part of any other database cluster.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Name]*</td>
<td>Name of database cluster to be created</td>
<td>-</td>
</tr>
<tr>
<td>[-Shard]*</td>
<td>Name of shard to be added to the cluster</td>
<td>-</td>
</tr>
<tr>
<td>[-Port]*</td>
<td>Port for the shard in the cluster</td>
<td>-</td>
</tr>
<tr>
<td>[-Server]</td>
<td>Node IP address</td>
<td>First node IP resolved by DNS</td>
</tr>
</tbody>
</table>


Examples:

- This command creates a cluster named "Pleiades" with the specified shard, its corresponding port, and local machine with first address resolved by DNS as node IP.

  New-DatabaseCluster -Name Pleiades -Shard shard1 -Port 2101

- This command creates a cluster named "Pleiades" with the specified shard, its corresponding port, and remote server IP.

  New-DatabaseCluster -Name Pleiades -Shard shard1 -Port 2101 -Server 192.168.0.15
**Troubleshooting**

**Unable to create the cluster.**
This occurs if the environment variables have not been refreshed after the installation and might require a reboot of the system.
However, if you do not want to reboot, you can import the NosDB PowerShell module (`NosDBPS.dll`) found in the location: `%INSTALL_DIR%\bin\nosdbps`.
Execute the following command to import the module:

```
Import-Module NosDBPS
```

---

### 8.2. Connect to Existing Database Cluster

`Connect-DatabaseCluster` establishes a connection with the database cluster which will manage shards, view database details and manage database cluster nodes using the cluster.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Server</code></td>
<td>IP of the node connecting to the database cluster</td>
<td>First node IP resolved by DNS</td>
</tr>
<tr>
<td><code>-Port</code></td>
<td>Port of NosDB configuration service for the cluster</td>
<td>9950</td>
</tr>
<tr>
<td><code>-StandAlone</code></td>
<td>To specify if the database is standalone</td>
<td>False</td>
</tr>
</tbody>
</table>

**Examples:**

- This command connects to a cluster created on the local machine.

```
Connect-DatabaseCluster
```

- This command connects to a cluster created on a remote machine.

```
Connect-DatabaseCluster -Server 192.168.1.187 -Port 2147
```

- This command connects to standalone database created on a remote location.

```
Connect-DatabaseCluster -Server 192.168.1.187 -Port 2147 -StandAlone
```
### Troubleshooting

**Unable to connect to the cluster.**
This occurs if the environment variables have not been refreshed after the installation and might require a reboot of the system.
However, if you do not want to reboot, you can import the NosDB PowerShell module (NosDBPS.dll) found in the location: `%INSTALL_DIR%\bin\nosdbps\`. Execute the following command to import the module:

```powershell
Import-Module NosDBPS
```

Once connected, the following scripts will be executed from the drive `PS NosDB:`. This drive communicates with the underlying database and all commands will be executed through it.

#### 8.3. Disconnect from Cluster
The `Disconnect-DatabaseCluster` command disconnects PowerShell from the currently connected database cluster.

```powershell
Disconnect-DatabaseCluster
```

#### 8.4. Remove Database Cluster
Remove-DatabaseCluster removes the currently connected cluster, including all shards and databases created within it.

```powershell
Remove-DatabaseCluster
```
9. Shards

In This Chapter:

Add Shard
Remove Shard
Start Shard
Stop Shard

To manage shards, switch context from cluster to shards in NosDB PSDrive.

PS NosDB:\cluster\shards>

9.1. Add Shard

Add Shards adds a new shard to the database cluster.

All shards added to the database cluster are auto-started by default.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Name]*</td>
<td>Unique shard name</td>
<td>-</td>
</tr>
<tr>
<td>[-Port]*</td>
<td>Port for the shard</td>
<td>-</td>
</tr>
<tr>
<td>[-Server]</td>
<td>Node IP address</td>
<td>First node IP resolved by DNS</td>
</tr>
<tr>
<td>[-HeartBeat]</td>
<td>Heartbeat interval for the shard nodes in seconds</td>
<td>5</td>
</tr>
</tbody>
</table>

Add-Shard [-Name] [-Port] [-Server] [-HeartBeat]

Examples:

- This command adds a shard named “shard1” with specified port 2101 and default heartbeat interval i.e. 5 seconds in the cluster, where “shard1” contains the current machine as node with first address resolved by DNS.

Add-Shard -Name shard1 -Port 2101
This command adds a shard named *shard1* with specified port 2101 and default heartbeat interval i.e. 5 seconds in the database cluster, where *shard1* contains a remote server 192.168.0.15.

```
Add-Shard -Name shard1 -Port 2101 -Server 192.168.0.15
```

This command adds a shard named *shard1* with specified port 2101 and specified heartbeat interval i.e. 8 seconds in the database cluster, where *shard1* contains a remote server 192.168.0.15.

```
Add-Shard -Name shard1 -Port 2101 -Server 192.168.0.15 -HeartBeat 8
```

### 9.2. Remove Shard

*Remove-Shard* removes the specified shard from the database cluster.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Name]*</td>
<td>Name of shard to be removed</td>
<td>-</td>
</tr>
<tr>
<td>[-Quiet]</td>
<td>To execute the operation in quiet</td>
<td>-</td>
</tr>
<tr>
<td>[-Forced]</td>
<td>To specify if the removal is forceful</td>
<td>False</td>
</tr>
</tbody>
</table>

```
Remove-Shard [-Name] [-Quiet] [-Forced]
```

**Examples:**

- This command removes "*shard1*" from the database cluster after prompting confirmation. However, state transfer takes place before removal of the shard.

```
Remove-Shard shard1
```

- This command removes "*shard1*" from the database cluster after prompting confirmation. However, data from the shard is lost as removal is forceful.

```
Remove-Shard shard1 -Forced
```

- This command removes "*shard1*" from the database cluster without prompting confirmation. However, state transfer takes place before removal of the shard.

```
Remove-Shard shard1 -Quiet
```
This command removes "shard1" from the database cluster without prompting confirmation. However, data from the shard is lost as removal is forceful.

Remove-Shard shard1 -Quiet -Forced

### 9.3. Start Shard

Start-Shard starts any stopped shard of the database cluster. This command works in the context of shards and the specific shard as well.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Name]</td>
<td>Name of shard to be started</td>
</tr>
</tbody>
</table>

This command starts the specified shard from the database cluster.

PS NosDB:cluster\shards> Start-Shard [-Name]

This command starts the shard in the context of the shard to be started.

PS NosDB:cluster\shards\shard1> Start-Shard

### 9.4. Stop Shard

Stop-Shard stops the specified shard of the database cluster. This command works in the context of "shards" and the specific shard as well.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Name]</td>
<td>Name of shard to be stopped</td>
</tr>
</tbody>
</table>

This command stops the specified shard from the database cluster.

PS NosDB:cluster\shards> Stop-Shard [-Name]

This command stops the shard in the context of the shard to be stopped.

PS NosDB:cluster\shards\shard1> Stop-Shard
Configuring Databases
10. Databases

In This Chapter:
Create New Database
Configure Database Properties
Drop Database

10.1. Create New Database

The CREATE statement in DDL allows creating a new database in NosDB through PowerShell with the Invoke-SQL-Query command.

```
Invoke-SQL -Query 'CREATE DATABASE northwind'
```

This command creates a database northwind with default configurations. If you wish to configure the properties for the database, refer to Configure Database Properties.

10.2. Configure Database Properties

Database properties can be configured through JSON configuration pairs according to the following DDL syntax:

```
CREATE DATABASE <Database> ({<JSONConfigurationPair> [, <JSONConfigurationPair>]})
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiFile</td>
<td>Specifies whether the data should be kept in multiple files or single files</td>
<td>true</td>
</tr>
<tr>
<td>MaxFileSize</td>
<td>File size in case of multi-file store in MB</td>
<td>1000</td>
</tr>
<tr>
<td>MaxCollections</td>
<td>Maximum number of collections that can be created</td>
<td>1000</td>
</tr>
<tr>
<td>CacheSize</td>
<td>Size of cache in MB</td>
<td>512</td>
</tr>
<tr>
<td>ChecksumEnabled</td>
<td>Specifies whether checksum should be enabled in journal</td>
<td>true</td>
</tr>
</tbody>
</table>
### CleanupInterval

Periodic interval in seconds after which journal operations should be cleaned

<table>
<thead>
<tr>
<th>CleanupInterval</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Periodic interval in seconds after which journal operations should be cleaned</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FileSizeLimit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum file size for Journal in MB</td>
<td>5</td>
</tr>
</tbody>
</table>

**Examples:**

- This command creates a database "northwind" with single-file store, cache size 2117 MB, 100 collections at maximum and default journaling options.

```bash
Invoke-SQL -Query 'CREATE DATABASE northwind {"MultiFile": false, "CacheSize":2117, "MaxCollections": 100}'
```

- This command creates a database "northwind" with 100 collections at maximum, default checksum enabled, and cleanup interval of 50 seconds with a journal file size of 10 MB.

```bash
Invoke-SQL -Query 'CREATE DATABASE northwind {"MaxCollections": 100, "Journal": {"CleanupInterval": 50, "FileSizeLimit": 10}}'
```

### 10.3. Drop Database

The DROP statement in DDL allows dropping an existing database in NosDB through PowerShell with the Invoke-SQL –Query command.

```bash
Invoke-SQL -Query 'DROP DATABASE northwind'
```

This command drops the database named northwind along with all collections in it.
11. Collections

In This Chapter:
Create Collection
Drop Collection

11.1. Create Collection

The CREATE statement in DDL allows creating a new collection in NosDB through PowerShell with the Invoke-SQL –Query command:

```
Invoke-SQL -Query 'CREATE COLLECTION Products {"Database": "northwind"}''
```

This command creates a collection Products in the database northwind with default configurations.

11.2. Drop Collection

The DROP statement in DDL allows dropping an existing collection in NosDB through PowerShell with the Invoke-SQL –Query command:

```
Invoke-SQL -Query 'DROP COLLECTION Products {"Database": "northwind"}''
```

This command drops the collection Products in the database northwind.
12. Indexes

In This Chapter:
- Create Index
- Drop Index

12.1. Create Index

Similar to collections, indexes can be created using CREATE statement.

- NosDB Open Source Edition only supports single attribute indexes.

This command creates an index *ProductIndex* on database *northwind* and collection *Products*, where the attribute for the index is *ProductID*, sorted in ascending order.

```
Invoke-SQL -Query 'CREATE INDEX ProductIndex {"Database": "northwind", "Collection": "products", "Attributes": {"Attribute": "ProductID", "SortOrder": "ASC"}}'
```

12.2. Drop Index

This command drops the index *ProductIndex* from the collection *Products* in database *northwind*.

```
Invoke-SQL -Query 'DROP INDEX ProductIndex {"Database": "northwind", "Collection": "products"}'
```
Configuring Security

NosDB Open Source Guide
13. Security Concepts

In This Chapter:
- Authentication
- Roles

13.1. Logins

Logins facilitate authentication in a system. 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 Enterprise supports two types of logins:

Windows Login

This includes a challenge-response based authentication model involving Microsoft Active Directory for the users of a local system, domain or workgroup.

NosDB Login

This is similar to the SQL Server Authentication in SQL Server, which consists of a custom username-password authentication model.

NosDB OpenSource Edition offers Windows Login. If a user belongs to the domain and is 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.

13.2. Roles and Users

A role is a set of operations that can be assigned to a login to define his/her responsibilities as a user. 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:

13.2.1. Cluster Roles

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbcreator</td>
<td>A user with the dbcreator role can perform any data definition (DDL) operation on the databases, like CREATE, DROP or ALTER databases over the cluster.</td>
</tr>
<tr>
<td>clustermanager</td>
<td>A user with the clustermanager role can perform start/stop operations on the shards and nodes of the cluster.</td>
</tr>
</tbody>
</table>
A user with the clusteradmin 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.

### 13.2.2. Database Roles

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_datareader</td>
<td>A user with the db_datareader role can perform only read operations on the database. The db_datareader role is owned by db_user.</td>
</tr>
<tr>
<td>db_datawriter</td>
<td>A user with the db_datawriter role can perform only write (INSERT/UPDATE/DELETE) operations on the database. The db_datawriter role is owned by db_user.</td>
</tr>
<tr>
<td>db_user</td>
<td>A user with the db_user role can perform read and write operations on the database.</td>
</tr>
<tr>
<td>db_admin</td>
<td>A user with the db_admin 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>
<tr>
<td>db_owner</td>
<td>A user with the db_owner role can perform all managerial operations on the database as performed by the db_admin, along with having the authorization to DROP the database.</td>
</tr>
</tbody>
</table>
14. Logins

In This Chapter:
Add Login
Drop Login

14.1. Add Login

You can add a new login through the Add-Login tool or through DDL querying.

14.1.1. Using Add-Login tool

Add-Login creates a login on a remote server. It will create an account for the current logged in user of PowerShell on the remote server using Windows credentials of the remote server.

Any login created through this tool will be granted sysadmin role by default.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Username]</td>
<td>User account name for Windows account of the remote server</td>
</tr>
<tr>
<td>[-Password]</td>
<td>Password for Windows account of the remote server</td>
</tr>
<tr>
<td>[-Server]</td>
<td>IP of the remote server to be added</td>
</tr>
<tr>
<td>[-Port]</td>
<td>Configuration port of the remote server</td>
</tr>
</tbody>
</table>


- Adds login alachisoft\john_smith on remote server 192.168.0.22 and Port 2280.

Add-Login -Username alachisoft\john_smith -Password letMeIn -Server 192.168.0.22 -Port 2280

14.1.2. Using DDL Query

Logins can also be created through Invoke-SQL.

Invoke-SQL -Query "CREATE LOGIN alachisoft\john_smith {"UserType": "Windows", "Password": "nosdb123"}"

Copyright © 2005-2016 Alachisoft
14.2. Drop Login

The DROP command drops the existing login `alachisoft\john_smith` and its details.

```sql
Invoke-SQL -Query 'DROP LOGIN alachisoft\john_smith {"UserType": "Windows", "Password": "nosdb123"}'
```
15. Roles and Users

In This Chapter:

Cluster Roles
Database Roles

Internally, the cluster name is stored as “cluster”. Hence, make sure the cluster name is “cluster” while configuring roles.

You can grant or revoke roles to logins to create/remove users by invoking DCL commands – GRANT and REVOKE.

15.1. Cluster Roles

A cluster user with cluster-wide security privileges is created if a registered login is granted with cluster roles. Cluster roles include:

1. dbcreator
2. clustermanager
3. clusteradmin

For in-depth explanation of these roles, please refer to Roles in Security Concepts.

- Grant Roles

This command grants a clusteradmin role to the login alachisoft\john_smith registered on the cluster. This creates a user with the clusteradmin privileges.

```
Invoke-SQL -Query 'GRANT "clusteradmin" ON "cluster" TO ''alachisoft\john_smith'''
```

- Revoke Roles

This command revokes the role clusteradmin from the login alachisoft\john_smith on the cluster the user is logged on.

```
Invoke-SQL -Query 'REVOKE "clusteradmin" ON "cluster" FROM ''alachisoft\john_smith'''
```

15.2. Database Roles

Database roles grant database-wide security privileges to a user. Database roles include:

1. db_datareader
2. db_datawriter
3. db_user
4. db_admin
5. **db_owner**

- **Grant Roles**

This command grants a *db_admin* role to the login *alachisoft\john_smith* on the database *northwind*, creating a database user.

```
Invoke-SQL -Query 'GRANT "db_admin" ON "cluster/northwind" TO ''alachisoft\john_smith'''
```

- **Revoke Roles**

This command revokes the *db_admin* role from the login *alachisoft\john_smith* on the database *northwind*.

```
Invoke-SQL -Query 'REVOKE "db_admin" ON "cluster/northwind" FROM ''alachisoft\john_smith'''
```
16. Backup Database

Before creating backups for the database, please refer to the in-depth explanation and requirements of Backup and Restore in Conceptual Guide.

16.1. Distributed vs. Consolidated Storage

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\NosDBbackups.

  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.

16.2. Backup Database

**Full Backup**

NosDB OpenSource offers Full Backup. Make sure that if the domain IP is being provided, it is in the format "xx-xxx-xx-xx".

**Distributed vs Consolidated Storage**

NosDB automatically detects whether the backup is distributed or consolidated, based on the Path provided.

Invoke-SQL -Query 'BACKUP DATABASE northwind {"Path": "\server1\Backups", "UserName": "domain1\john", "Password": "admin1234"}"

The backup is created in the specified location in a folder named with the format *databasename-timestamp-status*. For example, for a complete backup made on the database northwind, the folder is created with the name "northwind-20160621161706536-completed". In case the backup job fails due to any unexpected failure, there might be some shards showing the status "completed", while others might still be named with the status "in-progress". This helps in distinguishing whether the backup was fully made or not.
17. Restore Database

Before restoring the database, please refer to the in-depth explanation and requirements of *Backup and Restore* in *Conceptual Guide*.

**Full Restore**
Since NosDB OpenSource offers Full Backup, the restoration process is Full Restoration.

**Distributed vs. Consolidated Storage**
Similarly, the restoration from distributed or consolidated is determined by the path of the backed up database provided.

```
Invoke-SQL -Query 'RESTORE DATABASE northwind_restored {
"Path": "\server1\Backups\northwind-20160624162210440-completed",
"UserName": "domain1\john",
"Password": "admin1234",
"SourceDatabase": "northwind"}"
```

18. Get Task Information

Since the backup and restore tasks are non-blocking, you can carry on other tasks while the database is backed up or restored. *Get-TaskInfo* fetches and displays the current status of the backup and restore jobs being performed.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-ShowHistory]</td>
<td>Flag to show task information for all tasks executed.</td>
<td>false</td>
</tr>
</tbody>
</table>

**Examples:**

- This command shows task information for the current running task.
  
  Get-TaskInfo

- This command shows task information for all tasks that have been executed.
  
  Get-TaskInfo -ShowHistory
**19. Import Data**

_In This Chapter:_

Import CSV Data
Import JSON Data

NosDB PowerShell allows importing data from a JSON or CSV file to a collection. Moreover, the client has the facility to update the data if any data with the same key exists in the collection.

**Importing Data into NosDB**

Import-Data imports data from the specified file into the specified collection. This command only works in the context of the specified database. Note that for CSV Import, additional parameters are required like TextQualifier, ColumnDelimiter and RowDelimiter, which are explained further in the section _Import CSV Data_.

```
PS NosDB:\cluster\databases\[database-name]>
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Path]*</td>
<td>Location of the input file to be imported</td>
<td>-</td>
</tr>
<tr>
<td>[-Format]*</td>
<td>Format of data in the input file</td>
<td>-</td>
</tr>
<tr>
<td>[-CollectionName]*</td>
<td>Name of the NosDB collection into which data is to be imported.</td>
<td>-</td>
</tr>
<tr>
<td>[-Overwrite]</td>
<td>Specifies whether the existing data is to be overwritten or not.</td>
<td>False</td>
</tr>
<tr>
<td>[-ReportCount]</td>
<td>Report interval in terms of imported rows. No status printed if -Verbose is not specified.</td>
<td>1000</td>
</tr>
</tbody>
</table>

**19.1. Import CSV Data**

**Preconditions**

Before importing CSV to a NosDB collection, consistency has to be ensured between the format of data in the CSV file and JSON collection format. Hence, a CSV file import requires taking the following guidelines into consideration:

1. The header and its corresponding value should not mismatch as NosDB will populate the collection according to the provided format.

2. The format of any date specified within the CSV should comply with **ISO 8601** (YYYY-MM-DD). Any other format of date will be treated as string.

3. Since CSV is schema-independent, the system is actually “guessing” the datatypes while importing it into the collection. Hence, a numerical value specified as a string will be treated as string in the collection, which might lead to inconsistency.
4. Any attribute named "_key" in the CSV will be treated as the document key by default. If no such attribute exists, NosDB will generate the document key automatically.

**How to create CSV data from SQL Server**

NosDB allows creating CSV data from data stored in SQL Server 2012 and 2016.

- Export each table in SQL Server into CSV format as explained in the SQL Server Documentation with the following configurations:
  - **Text Delimiter**: double-quote (""")
  - **Column Delimiter**: TAB
  - **Row Delimiter**: {CR} {LF}
- Import the CSV files into NosDB through Import-Data:
  
  ```
  Import-Data -Format CSV -Path 'D:\SQLServerData\product.csv' -CollectionName "Products" -TextQualifier '' -ColumnDelimiter TAB -RowDelimiter CRLF
  ```

**Different CSV Options**

Apart from the aforementioned parameters, the NosDB supports importing data with varying delimiters and qualifiers.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-TextQualifier]</td>
<td>Specifies additional qualifier identifying text fields, such as single quote (&quot;&quot;') or double quote (&quot;&quot;).</td>
<td>-</td>
</tr>
<tr>
<td>[-ColumnDelimiter]</td>
<td>Delimiter for column. Possible values: TAB, CRLF, CR, LF, &quot;&quot;, &quot;:&quot;, &quot;,&quot;, &quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>[-RowDelimiter]</td>
<td>Delimiter for row. Possible values: TAB, CRLF, CR, LF, &quot;&quot;, &quot;:&quot;, &quot;,&quot;, &quot;</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

**Examples**

- Imports CSV file `products.csv` into collection **Products** with default reporting interval, i.e. 1000, no text qualifier, default column delimiter (""'), default row delimiter (CRLF) and overwriting of existing rows disabled.

  ```
  Import-Data -Format CSV -Path 'D:\Northwind\products.csv' -CollectionName "Products"
  ```

- Imports CSV file `products.csv` into collection **Products** with reporting interval of 2000 rows, specified text qualifier (""'), specified column delimiter (TAB), and row delimiter (CRLF). Existing rows with same document ID will be overwritten.

  ```
  Import-Data -Format CSV -Path 'D:\Northwind\products.csv' -CollectionName "Products"
  ```
19.2. Import JSON Data

Since data is stored in the collection in JSON format, the data export from JSON to JSON is smooth and requires no special limitations except that the JSON file being provided is an array of JSON documents.

Examples:

- Imports JSON file *products.json* into collection *Products* with default reporting interval, i.e. 1000 and overwriting of existing rows disabled.

```
Import-Data -Format JSON -Path 'D:\Northwind\products.json' -CollectionName "Products"
```

- Imports JSON file *products.json* into collection *Products* with default reporting interval, i.e. 1000 and where existing rows with same document ID will be overwritten.

```
Import-Data -Format JSON -Path 'D:\Northwind\products.json' -CollectionName "Products" -Overwrite
```

How to create JSON data from SQL Server

Importing Data from SQL Server 2012

For SQL Server versions 2012 and prior, JSON export is not supported. Thus, in case your source data is stored in a SQL Server 2012 database, you can export the tables into CSV format, convert them to JSON format and then import the JSON files into NosDB.

- Export each table in SQL Server into CSV format as explained in the [SQL Server Documentation](https://docs.microsoft.com/en-us/sql/relational-databases/import-and-export/import-and-export-data) with the same configurations as explained in CSV Import.
- This CSV file can now be converted to JSON format using the online converter [CSVJSON](https://csvjson.com).
- Your SQL Server tables as JSON can now be easily imported into NosDB through Import-Data.

```
Import-Data -Format JSON -Path 'D:\SQLServerData\product.json' -CollectionName "Products"
```

Importing Data from SQL Server 2016

SQL Server 2016 allows exporting the tables directly into JSON format, which is NosDB compliant. Hence, if you wish to import your SQL Server 2016 data into NosDB, you can do so by exporting the data into JSON format from SQL Server, and then importing the JSON files according to the aforementioned steps.
20. Export Data

In This Chapter:
Export CSV Data
Export JSON Data

NosDB PowerShell allows exporting data from a collection to a file in JSON or CSV format. In case a query is specified, the result of the query is stored to a file.

Exporting Data from NosDB

Export-Data exports either the whole data from a collection, or filtered data as a query result to the file. Note that for CSV Export, additional parameters are required like TextQualifier, ColumnDelimiter and RowDelimiter, which have been described in the section Export CSV Data.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Format]*</td>
<td>Format of data in the output file</td>
<td>-</td>
</tr>
<tr>
<td>[-Path]*</td>
<td>Location for the output file</td>
<td>-</td>
</tr>
<tr>
<td>[-Query]</td>
<td>Query to filter the data to be exported</td>
<td>-</td>
</tr>
<tr>
<td>[-ReportCount]</td>
<td>Report interval in terms of exported rows.</td>
<td>1000</td>
</tr>
</tbody>
</table>

This command works in the context of the specified database if a query is specified, else it will only work through the context of a collection.

- With Query specified:

  PS NosDB:\cluster\databases\[database-name]>

- Without Query specified:

  PS NosDB:\cluster\databases\[database-name]\collections\[collection-name]>

20.1. Export CSV Data

A CSV file is comparatively more rigid than JSON, thus the collection data in JSON has to be parsed accordingly before export. As a CSV file requires headers, NosDB creates the headers by taking the first document of the collection and parsing the attributes accordingly.

The data sequence and number of attributes in the collection should be consistent as the CSV headers are created using the first document in the collection. Hence, in successive documents, the number of attributes can be lesser than those defined as the header but should preferably not be more than them.
Different CSV Options
Apart from the aforementioned parameters, the NosDB supports exporting data with varying delimiters and qualifiers.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-TextQualifier]</td>
<td>Specifies additional qualifier identifying text fields, such as a single quote or double quote</td>
<td>-</td>
</tr>
<tr>
<td>[-ColumnDelimiter]</td>
<td>Delimiter for column. Possible values: TAB, CRLF, CR, LF, “;”, “,”, “</td>
<td>”</td>
</tr>
<tr>
<td>[-RowDelimiter]</td>
<td>Delimiter for row. Possible values: TAB, CRLF, CR, LF, “;”, “,”, “</td>
<td>”</td>
</tr>
</tbody>
</table>

Examples

- Exports result set from the query specified on northwind database (in the context) into CSV file with default report count, i.e. 1000 and specified text qualifier (""), specified column delimiter (TAB), and row delimiter (CRLF).

```bash
cd \cluster\databases\northwind\Export-Data -Format CSV -Path "D:\Northwind\products.csv" -TextQualifier '"' -ColumnDelimiter TAB -RowDelimiter CRLF -Query "Select * from Products"
```

- Exports collection products (in the context) into CSV file with default report count, i.e. 1000 and specified text qualifier (""), specified column delimiter (TAB), and row delimiter (CRLF).

```bash
cd \cluster\databases\northwind\collections\productsExport-Data -Format CSV -Path "D:\Northwind\products.csv" -TextQualifier '"' -ColumnDelimiter TAB -RowDelimiter CRLF
```

20.2. Export JSON Data

Since data is stored in the collection in JSON format, the data export from JSON to JSON is smooth and requires no special limitations.

Data will be exported in the form of an array of JSON documents to the JSON file.

Examples
• Exports data from collection *products* (in the context) into JSON file with report count set to 2000 rows.

```bash
cd \cluster\databases\northwind\collections\products
Export-Data -Format JSON -Path "D:\Northwind\products.json" -ReportCount 2000 -Verbose
```

• Exports result set from the query specified on *northwind* database (in the context) into JSON file with default report count, i.e. 1000.

```bash
cd \cluster\databases\northwind\
Export-Data -Format JSON -Path "D:\Northwind\products.json" -Query "Select * from Products"
```

21. **Export Cluster Configuration**

NosDB PowerShell allows exporting configuration of the specified database cluster to a file in JSON format.

Export-Configuration exports the configuration to a JSON file named as *clusternametimestamp.json*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Path]</td>
<td>Location for the output file</td>
</tr>
</tbody>
</table>

*Export-Configuration [-Path]*

**Example:**

```bash
Export-Data -Path 'D:\NorthwindConfiguration'
```
22. Using SQL Queries

In This Chapter:
Invoking SQL Queries
Invoking SQL Scripts

22.1. SQL Queries in PowerShell

NosDB PowerShell allows you to execute DDL and DML queries over the underlying database directly through PowerShell. Since the DDL queries are executed through ExecuteNonQuery(), there is no result returned to the command prompt, apart from any exceptions occurring during the execution of the query. You can perform CREATE, DROP, ALTER, BACKUP and RESTORE queries via NosDB PowerShell. For DML queries, the result for the SELECT queries is displayed on the console.

Commands are invoked through the Invoke-SQL command. NosDB offers two methods of executing queries through Invoke-SQL:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-Query]</td>
<td>Query to be executed</td>
</tr>
<tr>
<td>[-InputFile]</td>
<td>Path of the sql file containing DDL statements</td>
</tr>
</tbody>
</table>

Any parameters within the query will be passed in JSON format as explained in Programmers’ Guide.

22.1.1. Invoking SQL Queries

Data Definition Language (DDL) Queries

For more detail on DDL syntax and examples, refer to DDL Support in Database Programming using NosDB .NET API.

1. CREATE

Invoke-SQL -Query 'CREATE DATABASE northwind {"MultiFile": false, "CacheSize":2117, "MaxCollections": 17}''

2. DROP

Invoke-SQL -Query 'DROP COLLECTION Products {"Database": "northwind"}''

3. ALTER
Invoke-SQL -Query 'ALTER DATABASE northwind {"Journal": {"FileSizeLimit": 991}}'

Data Manipulation Language (DML) Queries

All non-query DML operations can be performed through PowerShell which include SELECT, INSERT, UPDATE and DELETE. For more detail on DML syntax and examples, refer to DML Support in Database Programming using NosDB .NET API.

1. SELECT

Invoke-SQL -Query 'SELECT CategoryName, UnitsInStock FROM Products WHERE Name = "Chai"'

2. INSERT

Invoke-SQL -Query 'INSERT INTO Products (ID, ProductName, SupplierID, UnitPrice, UnitsInStock, Discontinued, CategoryName) VALUES ('1','Chai',1,18.000,39,'false','Beverages')'

3. UPDATE

Invoke-SQL -Query 'UPDATE Products SET CategoryName = 'Drinks' WHERE CategoryName = 'Beverages''

4. DELETE

Invoke-SQL -Query 'DELETE FROM Products WHERE ProductID = 100'

22.2. Invoking SQL Scripts

Invoke-SQL [-InputFile]

Example:

Invoke-SQL -inputFile "D:\SQLProjects\SQLScripts.sql"
23. Move Collection among Shards

NosDB PowerShell allows moving single-sharded between shards, as they are contained within a single shard.

Move-Collection moves the collection to the specified shard. This command only works in the context of a single sharded in NosDB PSDrive.

```powershell
PS NosDB: \cluster\databases\[database-name]\collections\[collection-name]>
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-NewShard]</td>
<td>Name of shard on which the collection will be moved</td>
</tr>
</tbody>
</table>

**Examples:**

- Moves the collection into `shard1`.

```powershell
Move-Collection shard1
```

If the name of the shard includes any special characters, specify the name within single quotes.

```powershell
Move-Collection ‘shard-1’
```

24. Get NosDB Version Information

Get-NosDBVersion returns details about the currently installed version of NosDB. The following is displayed:

- Utility version
- Copyright
- Registered User details (provided during installation)
- Edition
- Evaluation period

```powershell
Get-NosDBVersion
```
Database Monitoring

NosDB Open Source Guide
25. NosDB Counters

Performance counters provide valuable information about system hardware, services, networks and applications that can be used to determine their performance. It helps in diagnosing problems, fine-tuning systems and applications, monitoring real-time application performance and resolving bottleneck cause of system components.

Performance counters can be viewed from the PerfMon tool.

**Performance Counters**

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetches/sec</td>
<td>Number of successful Get operations per second.</td>
</tr>
<tr>
<td>Inserts/sec</td>
<td>Number of successful Insert operations per second.</td>
</tr>
<tr>
<td>Updates/sec</td>
<td>Number of successful Update operations per second.</td>
</tr>
<tr>
<td>Deletes/sec</td>
<td>Number of successful Delete operations per second.</td>
</tr>
<tr>
<td>Cache Hits/sec</td>
<td>Number of successful Get operations from cache per second.</td>
</tr>
<tr>
<td>Cache Misses/sec</td>
<td>Number of failed Get operations from cache per second.</td>
</tr>
<tr>
<td>Cache Evictions/sec</td>
<td>Number of documents evicted from cache per second.</td>
</tr>
<tr>
<td>Requests/sec</td>
<td>Number of requests being processed per second.</td>
</tr>
<tr>
<td>Cache Count</td>
<td>Number of documents in the cache.</td>
</tr>
<tr>
<td>Cache Size</td>
<td>Size of the cache in bytes.</td>
</tr>
<tr>
<td>Average Update Time</td>
<td>Average time in microseconds taken to complete one update operation.</td>
</tr>
<tr>
<td>Average Insert Time</td>
<td>Average time in microseconds taken to complete one insert operation.</td>
</tr>
<tr>
<td>Average Delete Time</td>
<td>Average time in microseconds taken to complete one delete operation.</td>
</tr>
<tr>
<td>Average Fetch Time</td>
<td>Average time in microseconds taken to complete one fetch operation.</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>Average time in microseconds taken to complete execution of a query.</td>
</tr>
<tr>
<td>Average Document Size</td>
<td>Average size of documents in bytes.</td>
</tr>
<tr>
<td>Database Size</td>
<td>Size of the current database in bytes.</td>
</tr>
<tr>
<td>Documents Count</td>
<td>Number of documents in the database.</td>
</tr>
<tr>
<td>Documents Persisted/second</td>
<td>Number of documents persisted to store per second.</td>
</tr>
<tr>
<td>Pending Persistent Documents</td>
<td>Number of documents which are yet to be persisted to store.</td>
</tr>
</tbody>
</table>

**Debug Counters**

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State transfer/sec</td>
<td>Number of documents the current node is either reading from other nodes or sending to other nodes during a state transfer mode.</td>
</tr>
<tr>
<td>Data Balance/sec</td>
<td>Number of items the current node is either reading from other nodes or sending to other nodes during a Data Load Balancing mode.</td>
</tr>
</tbody>
</table>
26. **Windows Performance Monitor**

NosDB publishes cache server counters in PerfMon under category NosDB. This category has all the counters related to the cache server. Follow the steps given below to monitor the NosDB counters through PerfMon tool:

- Click on the Windows **Start** menu, type **PerfMon** and press **ENTER**.

- PerfMon tool opens up. Click on the **Performance Monitor** under **Monitoring Tools**.

- Click on the cross (X) button to remove the default counter which is already added to it.

- Click on the plus (+) button to open the **Add Counters** dialog box.

- Using the vertical slider of available counters list box, scroll upward to find **NosDB** category.
Click on the down arrow head icon to expand the NosDB category. All of its counters are listed under it.
• All running shards and databases appear inside of **Instances of selected objects** list box.

• Add the counters by selecting them from the list.

• Select the required instance or simply click on the `<All instances>` and click on the **Add >>** button. All of the selected counters for all the selected instances of databases appear in **Added counters** list box on the right side.

• Click **OK**. All of the selected counters will appear in PerfMon tool and can be monitored:
You can also change the view of PerfMon from **Report** view to **Line** view:
- Click on the **Line** option of the drop down menu. This changes the PerfMon view from Report to Line.

- In **Line** view, counters values are displayed in the form of a continuous graph:
27. Windows Event Logs

NosDB provides support to view events in Windows Event Log. NosDB logs its important events in **Window Event Logs**. This allows monitoring all such events through Windows Event Viewer or with any other tool that supports Window Event Logger. Some of the events that are logged include:

- All NosDB installation events. These events are combination of successful or aborted installation.
- All errors encountered by NosDB Configuration, Database and Distributor services.
- Service start and stop events.
- Shard start and stop.
- Node joining or leaving a shard.

![Windows Event Logs Example](image)

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of events: 3.552 (!) New events available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Date and Time</td>
</tr>
<tr>
<td>Information</td>
<td>4/11/2016 3:20:00 PM</td>
</tr>
<tr>
<td>Information</td>
<td>4/11/2016 3:19:43 PM</td>
</tr>
<tr>
<td>Information</td>
<td>4/11/2016 3:19:31 PM</td>
</tr>
<tr>
<td>Information</td>
<td>4/11/2016 3:19:12 PM</td>
</tr>
<tr>
<td>Information</td>
<td>4/11/2016 3:19:11 PM</td>
</tr>
<tr>
<td>Error</td>
<td>4/11/2016 3:18:48 PM</td>
</tr>
</tbody>
</table>
28. **NosDB Logs**

NosDB provides a rich set of modules for logging and viewing run-time statistics of different components available in NosDB. It also provides different levels of logging that includes log-files, report views, statistics and events.

NosDB logs the states of the components into dedicated log files. These log files are created whenever the component is started, and are locked until the component or its respective service is stopped. Log files encapsulate different levels of information based on logging status.

NosDB creates each log file for the particular component with specific time stamps. This time initializes as the component or tool starts and continues until its status changes from "running" to "stopped".

- **Database** (%INSTALL_DIR%/logs/dblog_timestamp.log)
- **Configuration Server** (%INSTALL_DIR%/logs/cslog_timestamp.log)

NosDB writes the log file with the standard default format that provides the user with familiarity while reading the file. It also logs the thread name whenever an operation is performed by any running thread. NosDB presents different stages of logging. These logging levels are configurable and are described as follows:

- **INFO**: Info level describes any useful information about any operation performed on database, such as starting and stopping state of state transfer operations. Info level is disabled by default.
- **ERROR**: This log flag provides the cause of errors that are raised during operation execution. This status log is useful in most of troubleshooting scenarios such as during start of shard nodes, or while establishing connection with the server.
- **WARNING**: This flag logs the storage status of any action or operation on the database, and will be logged once storage is about to exceed. This log information is useful to optimize database performance.
- **DEBUG**: This flag is disabled by default but the user can configure it. This log option prints detailed information about each operation like key and buckets during state transfer. This log information allows analyzing any database issues in detail whenever normal debugging is not possible and lets the user to drill down into any hidden issue. However, note that enabling debug logging is an expensive operation, performance wise.
29. Setting Up Visual Studio Environment

In order to begin development of applications using NosDB features, you need to have Microsoft Visual Studio and NosDB installed.

The following section has been implemented using **Visual Studio 2015** and **NosDB version 1.3**.

- Open **File** -> **New** -> **Project**.

  ![Microsoft Visual Studio]

  - Select **Console Application** from the range of generic templates.
  - Specify the **Name** for the application.
  - Specify the appropriate **Location** to save the project.
  - Click **OK** to create a new project.
29.1. Reference NosDB Assemblies

- To this solution, you need to add references to NosDB APIs in order to utilize its features.
- In Solution Explorer, right-click on the application name and select Add.
- Select Reference.
• **Reference Manager** Screen appears.
• Select **Browse** from the left panel and click the **Browse** button to select the reference.
• **Select the files to reference** screen appears.
• **Browse to** `%INSTALL_DIR%\bin\assembly`
• Select the following references from the list of NosDB assemblies:
  1. Alachisoft.NosDB.Client.dll
  2. Alachisoft.NosDB.Common.dll
  3. Alachisoft.NosDB.Core.dll
  4. Alachisoft.NosDB.Serialization.dll
29.2. Specify Namespaces

- You can now begin creating applications utilizing NosDB’s features, however, you need to specify the following namespaces to get started:
  - `using Alachisoft.NosDB.Common;`
  - `using Alachisoft.NosDB.Client;`
29.3. Using NosDB NuGet Package

NosDB delivers NuGet packages for all of its editions which provide assemblies used for performing different API operations on NosDB databases.

- Go to Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution... in Visual Studio.
- **NuGet Package Manager** opens up for the current solution.
- In the search box, type **NosDB** with the **Package source** set to **nuget.org**.
- NuGet packages for all three editions (Enterprise, Professional, and OpenSource) are displayed.
- Select the package required.
- Select the project solution to apply the change to and click **Install**.
- **Review Changes** window pops up, showing the changes to be made to the solution.
- Click **OK**.
- **License Acceptance** window appears, requiring acceptance of the license terms for the package.

- Click *I Accept*. 
The Output Window in Visual Studio displays the status of installation of the package.
30. Database Connection

To utilize the following APIs, include the following namespace in your application:

```csharp
using Alachisoft.NosDB.Client;
```

30.1. 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 `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>Connection Timeout</td>
<td>Time value to wait for while establishing connection to the data source</td>
</tr>
</tbody>
</table>

NosDB provides Windows Authentication for security:

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.

Windows Authentication can be specified by the client through the connection string while initializing the database. It does not require any username and password to be specified since NosDB automatically detects the login credentials through which the client is logged into the Windows machine.

The following examples connect to the database with default Windows Authentication.

- Initializing Clustered Database

```csharp
//connection string
string conn = "Data Source=20.200.20.44; Port=9950; Database=northwind; Local Instance=false;";
//initialize database
Database db = NosDB.GetDatabase(conn);
```
Initializing Standalone Database

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

```csharp
//connection string
string conn = "Data Source=20.200.20.44; Port=9950; Database=northwind; Local Instance=true;"
//initialize database
Database db = NosDB.GetDatabase(conn);
```
31. Basic Operations through Document API

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

NosDB allows performing basic CRUD operations on the database, for a single document or for bulk documents.

To utilize the following APIs, include the following namespace in your application:

```csharp
using Alachisoft.NosDB.Client;
```

31.1. Using NosDB Collection

An instance of NosDB’s `DBCollection` class is created and initiated through the `GetDBCollection()` 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.

31.1.1. With JSON Documents

```csharp
//create collection instance of JSONDocument type
dbCollection<JSONDocument> collection = null;

collection = db.GetDBCollection("Products"); // pre-requisite: “Products” exists
```

31.1.2. With Custom Objects

```csharp
//create collection instance of Product class type
dbCollection<Product> collection = null;

collection = db.GetDBCollection<Product>("Products"); // pre-requisite: “Products” exists
```
31.2. 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.

```csharp
//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 `[JsonProperty]` 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.

```csharp
public class Product
{
    [JsonProperty(PropertyName = "_key")]
    //key attribute will have value of ProductID
    public string ProductID { get; set; }
    public string Name { get; set; }
    public double UnitPrice { get; set; }
    public short UnitsInStock { get; set; }
    public bool Discontinued { get; set; }
}

class Category
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
}
### 31.2.1. Single document

The `InsertDocument()` method inserts a single document to the database by taking in an instance of the JSON document or custom object.

- **JSON Document**

```csharp
JSONDocument product = new JSONDocument();

product.Key = "key1";
product.Add("ProductID", "12");
product.Add("Name", "Chai");
product.Add("Discontinued", false);
product.Add("UnitPrice", 18.00);
product.Add("UnitsInStock", 38);

collection.InsertDocument(product);
```

- **Custom Object**

```csharp
DBCollection<Product> collection = db.GetDBCollection<Product>("Products"); //prerequisite: "Products" exists

Product product = new Product();

product.ProductID = "12";
product.Name = "Chai";
product.Discontinued = false;
product.UnitPrice = 18.00;
product.UnitsInStock = 38;

collection.InsertDocument(product);
```
31.2.2. Bulk Documents

The InsertDocuments() method inserts documents in bulk to the database by taking in an ICollection of documents or a custom object document.

- JSON Document

```csharp
product1.Key = "key1";
product1.Add("ProductID", "12");
product1.Add("Name", "Chai");
product1.Add("UnitPrice", 18.00);
product1.Add("UnitsInStock", 38);

product2.Key = "key2";
product2.Add("ProductID", "34");
product2.Add("Name", "Aniseed Syrup");
product2.Add("UnitPrice", 10.00);
product2.Add("UnitsInStock", 13);

ICollection<JSONDocument> docs = new List<JSONDocument>(); // collection of documents
docs.Add(product1);
docs.Add(product2);
List<FailedDocument> failedDocs = collection.InsertDocuments(docs);
```

- Custom Objects

```csharp
product1.ProductID = "12";
product1.Name = "Chai";
product1.Discontinued = false;
product1.UnitPrice = 18.00;
product1.UnitsInStock = 38;

product2.ProductID = "34";
product2.Name = "Aniseed Syrup";
product2.Discontinued = false;
product2.UnitPrice = 10.00;
product2.UnitsInStock = 13;

ICollection<Product> docs = new List<Product>();
docs.Add(product1);
docs.Add(product2);
List<FailedDocument> failedDocs = collection.InsertDocuments(docs);```
31.3. Get Documents

31.3.1. Single Document

The GetDocument() method fetches the corresponding document from the database by taking in a string document key. The document is returned to an instance of T, according the object type added.

- JSON Document

```csharp
JSONDocument result = collection.GetDocument("key1");
//perform operations
```

- Custom Objects

```csharp
Product result = collection.GetDocument("12");
//perform operations
```

31.3.2. Bulk Documents

The GetDocuments() method fetches the corresponding documents from the database specified through a ICollection of document keys or a specified criteria. The documents are returned to an instance of IDBCollectionReader, which can be enumerated to read the results.

- JSON Documents

```csharp
ICollection<string> docKeys = new List<string> { "key1", "key2" }; //collection of document keys
IDBCollectionReader reader = collection.GetDocuments(docKeys);
while (reader.ReadNext())
{
    IJSONDocument doc = reader.GetDocument();
    //perform operations
}

string criteria = "((ProductID > 5) OR (UnitsInStock < 20))";
IDBCollectionReader reader = collection.GetDocuments(criteria);
while (reader.ReadNext())
{
    IJSONDocument doc = reader.GetDocument();
    //perform operations
}
Custom Objects

```csharp
List<string> docKeys = new List<string> { "12", "34" }; // document keys to be fetched

IDBCollectionReader reader = collection.GetDocuments(docKeys);

while (reader.ReadNext())
{
    Product result = reader.GetObject<Product>();
    //perform operations
}
```

### 31.4. Update Documents

#### 31.4.1. Single Document

The UpdateDocument() method updates an existing document by taking in an instance of the updated document (against the same key).

- **JSON Document**

```csharp
JSONDocument product = new JSONDocument();

product.Key = "key1";
product.Add("ProductID", "12");
product.Add("ProductName", "Chai"); // Attribute name changed from Name to ProductName
product.Add("UnitPrice", 19.50); // UnitPrice changed from 18.00 to 19.50

collection.UpdateDocument(product);
```

- **Custom Objects**

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

product.ProductID = "12";
product.Name = "Chai";
product.Discontinued = false;
product.UnitPrice = 19.50; // UnitPrice changed from 18.00 to 19.50
product.UnitsInStock = 38;

collection.UpdateDocument(product);
```
31.4.2. Bulk Documents

The UpdateDocuments() method updates a bulk of existing documents, specified by a ICollection<T> of updated documents (against the unchanged keys) or of a custom object.

- JSON Documents

```csharp
product1.Key = "key1";
product1.Add("ProductID", "12");
product1.Add("ProductName", "Chai");
product1.Add("UnitPrice", 19.50);

product2.Key = "key2";
product2.Add("ProductID", "34");
product2.Add("Name", "Aniseed Syrup");
product2.Add("UnitPrice", 10.00);

IICollection<JSONDocument> docs = new List<JSONDocument>(); // collection of documents
docs.Add(product1);
docs.Add(product2);

List<FailedDocument> failedDocs = collection.UpdateDocuments(docs);
```

- Custom Objects

```csharp
product1.ProductID = "12";
product1.Name = "Chai";
product1.Discontinued = false;
product1.UnitPrice = 19.50;

product2.ProductID = "34";
product2.Name = "Aniseed Syrup";
product2.UnitsInStock = 04;

IICollection<Product> docs = new List<Product>();
docs.Add(product1);
docs.Add(product2);

List<FailedDocument> failedDocs = collection.UpdateDocuments(docs);
```
## 31.5. Delete Documents

### 31.5.1. Single Document

The `DeleteDocument()` method deletes the document specified through a `string` document key.

```csharp
collection.DeleteDocument("key1");
```

### 31.5.2. Bulk Documents

The `DeleteDocuments()` method deletes a bulk of existing documents specified by a `ICollection<string>` of document keys.

```csharp
ICollection<string> docKeys = new List<string> { "key1", "key2" }; // collection of document keys
List<FailedDocument> failedDocs = collection.DeleteDocuments(docKeys);
```
# 32. Querying in NosDB

In This Chapter:

- **Data Definition Language (DDL) Support**
- **Data Control Language (DCL) Support**
- **Data Manipulation Language (DML) Support**
- **Query Syntax**

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.

## 32.1. Data Definition Language (DDL) Support

NosDB offers DDL support, which allows the user to "define" (create, alter or drop) NosDB resources, and backup and restore the database. You can either provide the configuration in a string format, or by creating a JSON document for the configuration.

### 32.1.1. CREATE Statement

Creates the resources with specific configurations in the form of JSON object key-value pairs.

```sql
CREATE <Resource> ([<JSONConfigurationPair> [, <JSONConfigurationPair>*]) [;

<Resource> ::= DATABASE | COLLECTION | LOGIN | INDEX
```

### CREATE DATABASE

```sql
CREATE DATABASE <Database> ([<JSONConfigurationPair> [, <JSONConfigurationPair>*]) [;

<Database> ::= <String>

<JSONConfigurationPair> ::= [MultiFile : <Boolean>]
  | [CacheSize : <Number>]
  | [MaxFileSize : <Number>]
  | [MaxCollections : <Number>]
  | [Journal : <JournalObject>]

<JournalObject> ::= [ChecksumEnabled : <Boolean>]
  | [CleanupInterval : <Number>]
  | [FileSizeLimit : <Number>]
```
Examples:

- **String configuration**

  ```csharp
  string dbName = "northwind";
  string config = "{"MultiFile": false,"CacheSize":2117,"MaxCollections": 17,
  "Journal": {"ChecksumEnabled":true,"CleanupInterval":10}}";
  
  string query = "CREATE DATABASE " + dbName + config;
  db.ExecuteNonQuery(query);
  ```

- **JSON configuration**

  ```csharp
  string dbName = "northwind";
  int cacheSize = 1024;
  int max = 7;
  
  JSONDocument config = new JSONDocument();
  config.Add("MultiFile", true);
  config.Add("CacheSize", cacheSize);
  config.Add("MaxCollections", max);
  
  string query = "CREATE DATABASE " + dbName + config;
  db.ExecuteNonQuery(query);
  ```

### CREATE COLLECTION

```
CREATE COLLECTION <Collection> (<JSONConfigurationPair> [, <JSONConfigurationPair>]*)
```

- `<Collection>` ::= `<String>`
- `<JSONConfigurationPair>` ::= `[
[Database : <String>]
| [Distribution : <DistributionObject>]
| [Shard : <String>]
| [MaxDocuments : <Number>]
| [CappedSize : <Number>]
| [EvictionEnabled : <Boolean>]
]

- `<DistributionObject>` ::= `[
[Strategy : <Strategy> [, Ranges : <Ranges>]]`
- `<Strategy>` ::= `HashBased | RangeBased | NonSharded`
- `<Ranges>` ::= `[MinRange : <String>,MaxRange : <String>, Shard : <String>]`
Examples:

- **String configuration**

```csharp
string collName = "Products";
string config = "{"Database":"northwind","Distribution":{"Strategy":"HashBased"},"PartitionKey": [{"KeyName":"Age","KeyType":"String"}]}";

string query = "CREATE COLLECTION " + collName + config;
db.ExecuteNonQuery(query);
```

- **JSON configuration**

```csharp
string collName = "Products";
string dbName = "northwind";

JSONDocument distribution = new JSONDocument();
distribution.Add("Strategy", "HashBased");

JSONDocument par = new JSONDocument();
par.Add("KeyName", "Category");
par.Add("KeyType", "string");

JSONDocument[] partitionKey = new JSONDocument[1];
partitionKey[0] = par;

JSONDocument config = new JSONDocument();
config.Add("Database", dbName);
config.Add("Distribution", distribution);
config.Add("PartitionKey", partitionKey);

string query = "CREATE COLLECTION " + collName + config;
db.ExecuteNonQuery(query);
```

### CREATE LOGIN

```
CREATE LOGIN <Login> (<JSONConfigurationPair> [, <JSONConfigurationPair>])? [ ]
```

- `<Login>` ::= <String>
- `<JSONConfigurationPair>` ::= [ UserType : <UserType> ]
  - | [ Password : <String> ]
- `<UserType>` ::= Windows
Examples:

- **String configuration**

```csharp
string loginName = "JohnSmith";
string config = "{"UserType":"Custom", "Password":"myPassword"}";

string query = "CREATE LOGIN " + loginName + config;
db.ExecuteNonQuery(query);
```

- **JSON configuration**

```csharp
string loginName = "JohnSmith";
string userType = "Custom";
string password = "myPassword";

JSONDocument config = new JSONDocument();
config.Add("UserType", userType);
config.Add("Password", password);

string query = "CREATE LOGIN " + loginName + config;
db.ExecuteNonQuery(query);
```

- **CREATE INDEX**

NosDB Open Source Edition only supports single attribute indexes.

```
CREATE INDEX <Index> (<JSONConfigurationPair> [, <JSONConfigurationPair>]* ) [ ];
<JSONConfigurationPair> ::= [Database : <String>] |
Collection : <String>] |
JournalEnabled : <Boolean>] |
Attributes : <AttributeObject>]

<AttributeObject> ::= [Attribute : <String>] |
SortOrder : ASC | DESC]
```
Examples:

- String configuration

```csharp
string indexID = "ProductIndex";

string query = "CREATE INDEX " + indexID + 
{"Database":"northwind","Collection":"products","Attributes":[{"Attribute":"ProductID"},{"Attribute":"ProductName","SortOrder":"ASC"}],"JournalEnabled":true}

db.ExecuteNonQuery(query);
```

- JSON configuration

```csharp
string collName = "Products";
string dbName = "northwind";

JSONDocument att = new JSONDocument();
att.Add("Attribute", "Category");
att.Add("SortOrder", "ASC");

JSONDocument[] attributes = new JSONDocument[1];
attributes[0] = att;

JSONDocument config = new JSONDocument();
config.Add("Database", dbName);
config.Add("Collection", collName);
config.Add("Attributes", attributes);
config.Add("JournalEnabled", true);

string indexID = "Category";

string query = "CREATE INDEX " + indexID + config;

db.ExecuteNonQuery(query);
```
32.1.2. ALTER Statement

Alters existing objects with specific configurations in the form of JSON object key-value pairs.

Note that ALTER does **not** allow modifying indices as indices can only be updated by dropping and creating new indices.

```
ALTER <Resource> <ResourceName> (<<JSONConfigurationPair> [, <JSONConfigurationPair>]*) [;]/
```

- **ALTER DATABASE**

```
ALTER DATABASE <Database> (<<JSONConfigurationPair> [, <JSONConfigurationPair>]*) [;]/
```

- **Examples:**
  - **String configuration**
    ```
    string dbName = "northwind";
    string config = "{"Journal": {"FileSizeLimit":991}}";
    string query = "ALTER DATABASE " + dbName + config;
    db.ExecuteNonQuery(query);
    ```
  - **JSON configuration**
    ```
    string dbName = "northwind";
    IJSONDocument journal= new JSONDocument();
    journal.Add("ChecksumEnabled", false);
    journal.Add("CleanupInterval", 20);
    JSONDocument config = new JSONDocument();
    config.Add("Database", dbName);
    config.Add("Journal", journal);
    string query = "ALTER DATABASE " + dbName + config;
    db.ExecuteNonQuery(query);
    ```
### ALTER COLLECTION

**ALTER COLLECTION** <Collection> (<JSONConfigurationPair> [, <JSONConfigurationPair>]*)

![Syntax](image)

<Collection> ::= <String>

<JSONConfigurationPair> ::= [Database : <String>]
| [CappedSize : <Number>]
| [MaxDocuments : <Number>]

#### Examples:

- **String configuration**

  ```csharp
  string collName = "Products";
  string config = "\"Database\":\"northwind\",\"CappedSize\":525,\"MaxCapDocs\":1211\"";
  string query = "ALTER COLLECTION \" + collName + config;
  db.ExecuteNonQuery(query);
  ```

- **JSON configuration**

  ```csharp
  string dbName = "northwind";
  string collName = "Products";
  JSONDocument config = new JSONDocument();
  config.Add("Database", dbName);
  config.Add("CappedSize", 525);
  config.Add("MaxDocuments", 1211);
  string query = "ALTER COLLECTION \" + collName + config;
  db.ExecuteNonQuery(query);
  ```

#### 32.1.3. DROP Statement

Drops the existing database or collection with the pre-specified configurations.

**DROP** <Resource> (<JSONConfigurationPair> [, <JSONConfigurationPair>]*)

<Resource> ::= DATABASE | COLLECTION | LOGIN | INDEX
• **DROP DATABASE**

```csharp
DROP DATABASE <Database> [;]
<Database> ::= <String>
```

```csharp
string dbName = "northwind";
string query = "DROP DATABASE " + dbName;
db.ExecuteNonQuery(query);
```

• **DROP COLLECTION**

```csharp
DROP COLLECTION <Collection> <JSONConfigurationPair> [;]
<Collection> ::= <String>
<JSONConfigurationPair> ::= [Database : <String>]
```

**Examples:**
- **String configuration**

```csharp
string collName = "Products";
string query = "DROP COLLECTION " + collName + "{"Database":"northwind"}";
db.ExecuteNonQuery(query);
```
- **JSON configuration**

```csharp
string collName = "Products";
JSONDocument config = new JSONDocument();
config.Add("Database", "northwind");

string query = "DROP COLLECTION " + collName + config;
db.ExecuteNonQuery(query);
```

• **DROP INDEX**

```csharp
DROP INDEX <Index> <JSONConfigurationPair> [;]
[Index] ::= <String>
<JSONConfigurationPair> ::= [Database : <String>, Collection : <String>]
```

**Examples:**
- **String configuration**
```csharp
string indexID = "ProductIndex";
string config = "{"Database":"northwind","Collection":"Products"}";
string query = "DROP INDEX " + indexID + config;
db.ExecuteNonQuery(query);
```

- **JSON configuration**

```csharp
string indexID = "ProductIndex";

JSONDocument config = new JSONDocument();
config.Add("Database", "northwind");
config.Add("Collection", "Products");

string query = "DROP INDEX " + indexID + config;
db.ExecuteNonQuery(query);
```

- **DROP LOGIN**

```csharp
DROP LOGIN <Login> <JSONConfigurationPair> [];
<Login> ::= <String>
<JSONConfigurationPair> ::= [Password : <String>]
```

**Examples:**

- **String configuration**

```csharp
string loginName = "JohnSmith";
string query = "DROP LOGIN " + loginName + " {"Password":"myPassword"}"

db.ExecuteNonQuery(query);
```

- **JSON configuration**

```csharp
string loginName = "JohnSmith";
string password = "myPassword";

JSONDocument config = new JSONDocument();
config.Add("Password", password);

JSONDocument config = new JSONDocument();
string query = "DROP LOGIN " + loginName + config;
db.ExecuteNonQuery(query);
```
### 32.1.4. BACKUP Statement

Creates a backup of database, configuration metadata and Oplog that can be used to restore and recover the data after a failure. Backups of a database can also be used to restore a copy of the database to a new location.

```
BACKUP DATABASE <Database> <JSONConfigurationPair> [ ; ]
```

<Database> ::= <String>

<JSONConfigurationPair> ::= [ Path : <String> ]
                         | [ UserName : <String> ]
                         | [ Password : <String> ]

**Examples:**

- **String configuration**

  ```
  string pathForBackup = "\server1\Backup";
  string dbName = "northwind";
  string userName = "domain1\john";

  string config = "{ "Path" : "" + pathForBackup + ", "UserName" : "" + userName + ", "Password" : "nosAdmin123" }; 
  
  string query = "BACKUP DATABASE " + dbName + config;
  
  db.ExecuteNonQuery(query);
  ```

- **JSON configuration**

  ```
  string pathForBackup = "\server1\Backup";
  string dbName = "northwind";
  string userName = "domain1\john";

  JSONDocument config = new JSONDocument();
  config.Add("Path", pathForBackup);
  config.Add("UserName", userName);
  config.Add("Password", "nosAdmin123");

  string query = "BACKUP DATABASE " + dbName + config;
  
  db.ExecuteNonQuery(query);
  ```
### 32.1.5. RESTORE Statement

Recreates the database, Oplog and configuration metadata against a database from the provided backup.

```plaintext
RESTORE DATABASE <Database> <JSONConfigurationPair> [:]
<Database> ::= <String>
<JSONConfigurationPair> ::= Path : <String>
    | SourceDatabase : <String>
    | [UserName : <String>]
    | [Password : <String>]
```

**Examples:**

- **String configuration**

```plaintext
string pathOfBackup = "\\server1\Backup\northwind-20160621161706536-completed";
string srcDB = "northwind";
string dbName = "northwind_restored";
string userName = "domain1\john";

string config = "\"Path\":\"" + pathOfBackup + "\", \"UserName\":\"" + userName + "\", \
\"Password\":\"nosAdmin123\", \"SourceDatabase\":\"" + srcDB + "\"");

string query = "RESTORE DATABASE " + dbName + config;

db.ExecuteNonQuery(query);
```

- **JSON configuration**

```plaintext
string pathOfBackup = "\\server1\Backup\northwind-20160621161706536-completed";
string srcDB = "northwind";
string dbName = "northwind_restored";
string userName = "domain1\john";

JSONDocument config = new JSONDocument();

config.Add("SourceDatabase", srcDB);
config.Add("Path", pathOfBackup);
config.Add("UserName", userName);
config.Add("Password", "nosAdmin123");

string query = "RESTORE DATABASE " + dbName + config;

db.ExecuteNonQuery(query);
```
32.2. Data Control Language (DCL) Support

NosDB offers DCL support to enhance data access control with varying permissions granted to or revoked from the users.

NosDB provides two categories of roles:

<table>
<thead>
<tr>
<th>Database Roles</th>
<th>Cluster Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_owner</td>
<td>clusteradmin</td>
</tr>
<tr>
<td>db_admin</td>
<td>clustermanager</td>
</tr>
<tr>
<td>db_user</td>
<td>dbcreator</td>
</tr>
<tr>
<td>db_datawriter</td>
<td></td>
</tr>
<tr>
<td>db_datareader</td>
<td></td>
</tr>
</tbody>
</table>

32.2.1. GRANT Statement

GRANT provides the user with the permissions of a specific role.

```
GRANT <Role> ON <Resource> TO <Login>
```

```
<Role> ::= <String>
<Resource> ::= <String>
<Login> ::= <String>
```

- **Cluster Role**

Cluster roles grant cluster wide security privileges to a user.

```java
String role = "clusteradmin";
String resource = "cluster";
String login = "'JohnDoe'";
String query = "GRANT " + role + " ON " + resource + " TO " + login;
db.ExecuteNonQuery(query);
```

- **Database Role**

Database roles grant database wide security privileges to a user.

```java
String role = "db_admin";
String resource = "cluster/northwind";
String login = "'JohnDoe'";
String query = "GRANT " + role + " ON " + resource + " TO " + login;
db.ExecuteNonQuery(query);
```
### 32.2.2. REVOKE Statement

REVOKE nullifies the granted permissions of a specific role.

```plaintext
REVOKE <Role> ON <Resource> FROM <Login>

<Role> ::= <String>
<Resource> ::= <String>
<Login> ::= <String>
```

#### Standalone Database

```csharp
string role = "db_admin";
string resource = "local";
string login = "'JohnDoe'";

string query = "REVOKE " + role + " ON " + resource + " FROM " + login;
db.ExecuteNonQuery(query);
```

#### Cluster Role

```csharp
string role = "clusteradmin";
string resource = "cluster";
string login = "'JohnDoe'";

string query = "REVOKE " + role + " ON " + resource + " FROM " + login;
db.ExecuteNonQuery(query);
```

#### Database Role

```csharp
string role = "db_admin";
string resource = "cluster/northwind";
string login = "'JohnDoe'";

string query = "REVOKE " + role + " ON " + resource + " FROM " + login;
db.ExecuteNonQuery(query);
```

#### Standalone Database

```csharp
string role = "db_admin";
string resource = "local";
string login = "'JohnDoe'";

string query = "REVOKE " + role + " ON " + resource + " FROM " + login;
db.ExecuteNonQuery(query);
```
32.3. 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>Querying keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Clauses within &quot;[ ]&quot; are <strong>optional</strong>; e.g. <code>'[WHERE &lt;Predicate&gt;]'.</code></td>
</tr>
<tr>
<td>[ ]*</td>
<td>Arguments in <code>[ ]*</code> can be <strong>repeated for 0 or more number of times</strong>; e.g. <code>'[, &lt;BinaryExpression&gt;]*'.</code></td>
</tr>
<tr>
<td></td>
<td>Represents a <strong>choice</strong> between arguments of a query; e.g. `'[ASC</td>
</tr>
</tbody>
</table>

**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>
</table>
| ```
{
  "_key": "DA123",
  "ProductID": 10,
  "Name": "Chai",
  "UnitPrice": "18.00",
  "UnitsInStock": 23,
  "Discontinued": "False",
  "CategoryName": "Beverages"
}
``` | ```
{
  "_key": "TE1368",
  "OrderID": 10248,
  "CustomerID": "VINET",
  "OrderDate": "2015-07-04",
  "Freight": 32.38,
  "ShipName": "Vins et alcools Chevalier"
}
``` |
32.3.1. 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>
[WHERE <Predicate>]
[GROUP BY (<BinaryExpression> [ASC|DESC][, <BinaryExpression> [ASC|DESC]]*) ]
[ORDER BY (<BinaryExpression> [ASC|DESC][, <BinaryExpression> [ASC|DESC]]*) ]
[OFFSET <IntegerValue> ROWS | OFFSET <IntegerValue> ROWS FETCH NEXT <IntegerValue> ROWS
  ONLY] [:]
```

Examples:

```
"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'"

"SELECT CustomerID, OrderDate FROM Orders GROUP BY OrderDate"
```

32.3.2. 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.

```
INSERT INTO <Collection> (<Attribute> [, <Attribute>]*)
VALUES (<ConstantBinaryExpression> [, <ConstantBinaryExpression>]*)
[:]
<ConstantBinaryExpression> = The binary expression which involves constants only.
```
### 32.3.3. 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.

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

**Examples:**

```
"UPDATE Products SET (UnitPrice = 19.50) WHERE UnitPrice = 18.00"
```

```
“UPDATE Orders SET OrderDate = DateTime('2016-03-03') WHERE OrderDate = DateTime('2000-01-01')”
```

```
"UPDATE Orders SET CustomerID = @customer, ShipName = 'Hanari Carnes' WHERE ID = 10248"
```

### 32.3.4. 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:**

```
"DELETE FROM Orders"
```

```
"DELETE FROM Products WHERE ProductID < 500"
```
32.4. Query Syntax in NosDB

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

```json
{
    "_key": "DA123",
    "ProductID": 10,
    "ProductName": "milk",
    "Category": "Dairy",
    "UnitsInStock": 23,
    "PackageSize": ["S", 'M', 'L'],
    "Types": ["Skimmed", "Full Fat"],
    "TypeBySizeAvailable": [[12, 5], [4, 17], [21, 21]],
    "Order": "2016/1/16",
}
```

32.4.1. 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:

```sql
SELECT * FROM Products WHERE "Order" = DateTime("2016, 1, 20")
```

```sql
SELECT * FROM Products WHERE $Order$ = DateTime("2016, 1, 20")
```
32.4.2. **Binary Operators**

NosDB supports unary as well as binary expressions:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use (Example)</th>
</tr>
</thead>
</table>
| +        | 10 + 2  
          | 'App'+'1e' (string concatenation) |
| -        | 10 - 2 |
| *        | 10 * 2 |
| /        | 10 / 2 |
| %        | 10 % 2 |

32.4.3. **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.

32.4.4. **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>!=</td>
<td>Not equals to</td>
<td>&quot;SELECT * 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>&lt;</code></td>
<td>Less than</td>
<td>&quot;SELECT * FROM Products WHERE ProductID &lt; 20&quot;</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td>&quot;SELECT * FROM Products WHERE ProductID &gt; 35&quot;</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than equal to</td>
<td>&quot;SELECT * FROM Products WHERE ProductID &lt;= 20100&quot;</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than equal to</td>
<td>&quot;SELECT * FROM Products WHERE ProductID &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 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>
### 32.4.5. 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><strong>SUM</strong></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><strong>COUNT</strong></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><strong>AVG</strong></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><strong>MIN</strong></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><strong>MAX</strong></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>

### 32.4.6. 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><strong>UCASE</strong></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><strong>LCASE</strong></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><strong>LEN</strong></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><strong>ROUND</strong></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><strong>MID</strong></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>
</tbody>
</table>
32.4.7. 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. Apart from the operators provided by SQL, NosDB offers 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 IN</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>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. 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>INSERT</td>
<td>Checks if the array does not contain any of the specified values.</td>
<td>&quot;UPDATE Products SET PackageSize INSERT (‘S’, ‘M’, ‘XL’)&quot;</td>
</tr>
<tr>
<td>REPLACE</td>
<td>Replaces existing value with the specified value.</td>
<td>&quot;UPDATE Products SET PackageSize REPLACE (‘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>

32.4.8. 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>DateTime.now</td>
<td>Returns the current DateTime.</td>
<td>&quot;SELECT Order FROM Products WHERE $Order$ &lt; DateTime.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 $Orders$ = DateTime(&quot;2016, 1, 20&quot;)&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>
<tr>
<td>OFFSET</td>
<td>Skips the specified number of resultant records.</td>
<td>&quot;SELECT ProductName FROM Products WHERE ProductID = 10 OFFSET 10 ROWS&quot;</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>OFFSET FETCH</td>
<td>An extension of the OFFSET keyword, skips the specified rows and fetches a limited number of records from the result set.</td>
<td>&quot;SELECT ProductName FROM Products WHERE ProductID = 10 OFFSET 20 ROWS FETCH NEXT 500 ROWS ONLY&quot;</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Returns only the distinct values in the result set.</td>
<td>&quot;SELECT DISTINCT ProductName FROM Products&quot;</td>
</tr>
<tr>
<td>RENAME TO</td>
<td>Renames the attribute.</td>
<td>&quot;UPDATE Products SET RENAME PackageSize TO 'PackagingSize'&quot;</td>
</tr>
</tbody>
</table>
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, include the following namespace in your application:

```csharp
using Alachisoft.NosDB.Client;
```

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

```csharp
long ExecuteNonQuery(string queryText);
long ExecuteNonQuery(string queryText, ICollection<IParameter> parameters);

IDBCollectionReader ExecuteReader(string queryText);
IDBCollectionReader ExecuteReader(string queryText, ICollection<IParameter> parameters);

object ExecuteScalar(string queryText);
object ExecuteScalar(string queryText, ICollection<IParameter> parameters);
```

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecuteReader</td>
<td>IDBCollectionReader</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>ICollection&lt;IParameter&gt;</td>
<td>Used in case of parametrized queries.</td>
</tr>
</tbody>
</table>

#### 33.1. 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**

```csharp
//precondition: database has been initialized
string query = "SELECT UnitPrice, UnitsInStock FROM Products WHERE Name = 'Chai';";

IDBCollectionReader reader = db.ExecuteReader(query);
while (reader.ReadNext())
{
    IJSONDocument data = reader.GetDocument(); //return result as JSONDocument
    //perform operations
}
```

- **Custom Objects**

```csharp
//precondition: database has been initialized
string query = "SELECT UnitPrice, UnitsInStock FROM Products WHERE Name = 'Chai';";

IDBCollectionReader reader = db.ExecuteReader(query);
while (reader.ReadNext())
{
    string categoryName = reader.GetString("UnitPrice");
    string categoryDescription = reader.GetString("UnitsInStock");
    //perform operations
}
```

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

IDBCollectionReader reader = db.ExecuteReader(query);
while (reader.ReadNext())
{
    Product product = reader.GetObject<Product>(); //object is deserialized into Product class
    //perform operations
}
```
//precondition: database has been initialized
string query = "SELECT OrderDetails FROM Orders";

IDBCollectionReader reader = db.ExecuteReader(query);
while (reader.ReadNext())
{
    OrderDetail[] orderDetails = reader.GetArray<OrderDetail>("OrderDetails");
}

33.2. 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 and DDL queries. ExecuteNonQuery() returns the number of rows affected by the operation, in case the user wishes to track the success of the operation.

- **INSERT Query**

```
string query = "INSERT INTO Products (ProductID, ProductName, UnitsInStock, CategoryName) VALUES ('1','Chai', 1, 39, 'Beverages')";
long rowsAffected = db.ExecuteNonQuery(query);
```

- **UPDATE Query**

```
string query = "UPDATE Products SET(CategoryName = 'Farm Fresh') WHERE CategoryName = 'Dairy'";
long rowsAffected = db.ExecuteNonQuery(query);
```

- **DELETE Query**

```
string query = "DELETE FROM Products WHERE CategoryName = 'Beverages'";
long rowsAffected = db.ExecuteNonQuery(query);
```

- **DDL Queries**

```
string pathForBackup = "\server1\Backup";
string dbName = "northwind";
string userName = "domain1\john";

string config = "{\"Path\":" + pathForBackup + ", \"UserName\":" + userName + ", \"Password\":" + nosAdmin123}";

string query = "BACKUP DATABASE " + dbName + config;
db.ExecuteNonQuery(query);
```
33.3. 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()` all together.

1. Aggregate Querying

This command returns the count of all products which have `categoryName` "Beverages".

```csharp
//precondition: database has been initialized
string query = "SELECT count(*) FROM Products WHERE CategoryName = 'Beverages';

object count = db.ExecuteScalar(query);
//perform operations
```

2. Scalar Querying

This command returns the `ProductName` of the first result of all products which have `categoryName` "Beverages".

```csharp
//precondition: database has been initialized
string query = "SELECT ProductName FROM Products WHERE CategoryName = 'Beverages';

object name = db.ExecuteScalar(query);
//perform operations
```
Database Programming using NosDB ADO.NET Provider

NosDB Open Source Guide
NosDB also extends its accessibility through ADO.NET, which requires no code changes. Any distributed Windows or Web application can now easily utilize ADO.NET services with NosDB, as it allows querying and CRUD operations over your live data. ADO.NET is a framework introduced by Microsoft to separate the data access layer from the logical layer. This means NosDB’s provider for ADO.NET can be used to establish connections with NosDB and perform querying over the data.

NosDB implements the following ADO.NET abstract classes which aid Windows and Web applications in establishing a connection with NosDB:

- **DbProviderFactory**
  The DbProviderFactory is the main class which is provided to the application’s configuration (App.config). The provider factory provides instances of all the classes implemented by NosDB’s ADO provider which includes creating connections and providing the connection string so that it is not added through the application every time. This connection string specifies the database in the cluster that will be considered the data source.

  Moreover, the security credentials can be provided within the connection string as well, so data will travel over the network securely.

  For more detail about the methods provided through DbProviderFactory, please refer to [this MSDN page](https://msdn.microsoft.com).

- **DbConnection**
  The DbConnection class initiates the connection with the database. It takes in the connection string specified in App.config, which provides the IP, port, cluster and database information for opening the accurate connection.

  For more detail about the methods provided through DbConnection, please refer to [this MSDN page](https://msdn.microsoft.com).

- **DbCommand**
  The DbCommand class initiates all querying operations against the database, i.e., executing query with functions like ExecuteNonQuery, ExecuteReader and ExecuteScalar.

  Note that for SELECT statements, `SELECT *` is not possible because NosDB is schema-less. This means that the results may miss some columns while enumerating over the results because of the varying number of columns in the documents. To select all columns, specify all the column names in the query.
- **DbDataReader**
  The results of the ExecuteReader method from DbCommand is returned to an instance of IDataReader, which allows you to enumerate over the result.

- **DbDataAdapter**
  The DbDataAdapter is used to store the data in the form of a DataSet or DataGrid for any GUI based applications. It contains the reference to the DbCommand class so all querying operations can take place on the data stored within the DbDataAdapter. The DbDataAdapter also contains a DBConnection instance so that it connects to the database transparently without user intervention and reads/writes to it.

In other words, it acts as a local database with tables as it dynamically accommodates the incoming data from NosDB by creating tables according to the fields contained within the JSON documents.

For more detail about the methods provided through DbDataAdapter, please refer to this MSDN page.
34.1. Setup Visual Studio Environment

NosDB’s custom ADO.NET provider can be integrated into your .NET application in Visual Studio without any code changes. All it requires is a modification in the App.config of your application where the NosDB provider is specified.

34.1.1. Reference NosDB ADO.NET Provider

To this solution, you need to add a reference to NosDB ADO.NET Provider in order to utilize its features.

- In Solution Explorer, right-click on the application name and select Add.
- Select Reference.
- Reference Manager Screen appears.
- Select Browse from the left panel and click the Browse button to select the reference.
- Select the files to reference screen appears.
- Browse to %INSTALL_DIR%\integrations\ADO.NET Data Provider\.
- Select Alachisoft.NosDB.ADO.NETProvider.dll
- Click Add.

34.1.2. Specify ADO.NET Namespaces

To utilize NosDB ADO.NET Provider, include the following namespace in your application:

- using System.Data.Common;
- using System.Data;
- using System.Configuration;

34.1.3. Setup App.config for ADO.NET

You can specify the database, connection string and provider factory information through the App.config of your application. Add the following configuration tags within the <configuration> tag in App.config:
<configuration>
  
  <connectionStrings>
    <add name="NosDBConnection"
         connectionString="Data Source=20.200.20.44; Port=9950; Database=northwind;"
         providerName="Alachisoft.NosDB.ADO.NETProvider" />
  </connectionStrings>

  <system.data>
    <DbProviderFactories>
      <add invariant="Alachisoft.NosDB.ADO.NETProvider" name="ADOPrimer for .NET"
            description="ADO.NET Provider for .Net"
            type="Alachisoft.NosDB.ADO.NETProvider.NosProviderFactory,
            Alachisoft.NosDB.ADO.NETProvider, Version=1.3.0.0, Culture=neutral, PublicKeyToken=8a1e00327893b9ef" />
    </DbProviderFactories>
  </system.data>
</configuration>

- <connectionStrings>

<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>connectionString</td>
<td>The actual connection string containing the IP, port, cluster and database name to which the application will connect.</td>
</tr>
<tr>
<td>providerName</td>
<td>The fully qualified name of NosDB's implementation of ADO.NET's DBProviderFactory, which is the main class through which the connection is created in an application.</td>
</tr>
</tbody>
</table>

- <DbProviderFactories>

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>invariant</td>
<td>The Assembly name of NosDB's implementation of ADO.NET's DBProviderFactory.</td>
</tr>
<tr>
<td>name</td>
<td>The human-understandable name of the provider.</td>
</tr>
<tr>
<td>description</td>
<td>The human-understandable description of the provider.</td>
</tr>
<tr>
<td>type</td>
<td>The fully qualified name of the class which implements the ADO.NET Provider factory.</td>
</tr>
</tbody>
</table>
34.2. Connect to Database

Connect to the database using ADO.NET's abstract classes and connection string specified in App.config.

```csharp
//Obtain connection string from App.config
ConnectionStringSettings connectionSettings = ConfigurationManager.ConnectionStrings["NosDBConnection"];  

//Load provider factory
DbProviderFactory factory = DbProviderFactories.GetFactory(connectionSettings.ProviderName);  

//Create and open connection
DbConnection connection = factory.CreateConnection();
connection.ConnectionString = connectionSettings.ConnectionString;
connection.Open();
```

34.3. Basic Operations through SQL API

34.3.1. Using ExecuteReader() Method

```csharp
using (DbCommand command = factory.CreateCommand())
{
    //Provide query to execute
    command.CommandText = "SELECT UnitPrice, UnitsInStock FROM Products WHERE Name = 'Chai'";

    //Provide connection to command
    command.Connection = connection;

    //Execute query and read data into DbDataReader
    IDataReader reader = command.ExecuteReader();
    while (reader.Read())
    {
        IJSONDocument data = reader.GetDocument();  //return result as JSONDocument
        //perform operations
    }
}
```

34.3.2. Using ExecuteNonQuery() Method

- INSERT Query

```csharp
using (DbCommand command = factory.CreateCommand())
{
    command.CommandText = "INSERT INTO Products(ID, Name, QuantityPerUnit, UnitPrice,  
    "UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued) " +  
    "VALUES(10011, 'Mashed Potatoes', '56', 45, 13, 3, 8, true)";

    command.Connection = connection;

    int rowsAffected = command.ExecuteNonQuery();  // Insert records into database
}
```
### UPDATE Query

```csharp
using (DbCommand command = factory.CreateCommand())
{
    command.CommandText = "UPDATE Products SET(UnitsInStock = 45, UnitPrice = 125.0) WHERE ProductName = 'Chai';"
    command.Connection = connection;

    // Update existing records
    int rowsAffected = command.ExecuteNonQuery();
}
```

### DELETE Query

```csharp
using (DbCommand command = factory.CreateCommand())
{
    command.CommandText = "DELETE FROM Products WHERE ProductName = 'Chai';"
    command.Connection = connection;

    // Delete existing records
    int rowsAffected = command.ExecuteNonQuery();
}
```

#### 34.3.3. Using ExecuteScalar() Method

```csharp
using (DbCommand command = factory.CreateCommand())
{
    command.CommandText = "SELECT count(*) FROM Products";
    command.Connection = connection;

    object count = command.ExecuteScalar();
}
```

#### 34.4. Binding UI Controls with NosDB through ADO.NET

Data binding is the process of establishing a connection between the application UI and business logic. Traditionally, data binding refers to the synchronization of the data source and the application UI. This implies that when data changes in data source, it is reflected in the application UI.

NosDB supports data binding with UI controls using ADO.NET Data Provider implementation.

##### 34.4.1. Using WinForms

- Drag and drop a **DataGridView** from the toolbar to the **Controls** container in Visual Studio.
- Open the code view of the **Form Control** and write the following code in `OnLoad()` method.
It is recommended to use the connection in the `using` block, because it will dispose the connection while leaving the `using` block.

```csharp
using (DbConnection connection = factory.CreateConnection())
{
    connection.ConnectionString = connectionSettings.ConnectionString;

    DbDataAdapter adapter = factory.CreateDataAdapter();
    adapter.SelectCommand = factory.CreateCommand();
    adapter.SelectCommand.Connection = connection;
    adapter.SelectCommand.CommandText = queryText;

    DataSet ds = new DataSet();
    ds.Tables.Add("data");

    int ret = adapter.Fill(ds.Tables["data"]);
    if (ds.Tables.Count > 0)
    {
        this.dataGridview1.DataSource = ds.Tables["data"];
    }
}
```
34.4.2. Using ASP.NET WebForms

- Drag and drop a GridView from the toolbar to the Web Controls container in Visual Studio.
- Open the Code Behind file of the page and write the following code in Page_Load() method:

It is recommended to use the connection in the using block, because it will dispose the connection while leaving the using block.

```csharp
using (DbConnection connection = factory.CreateConnection())
{
    connection.ConnectionString = connectionSettings.ConnectionString;

    DbDataAdapter adapter = factory.CreateDataAdapter();
    adapter.SelectCommand = factory.CreateCommand();
    adapter.SelectCommand.Connection = connection;
    adapter.SelectCommand.CommandText = queryText;

    DataSet ds = new DataSet();
    ds.Tables.Add("data");

    int ret = adapter.Fill(ds.Tables["data"]);
    if (ds.Tables.Count > 0)
    {
        this.GridView1.DataSource = ds.Tables["data"];  
    }
}
```

![ GridView Displaying Data ]

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Beverages</td>
<td>Soft drinks, coffees, teas, beers, and ales</td>
</tr>
<tr>
<td>2</td>
<td>Condiments</td>
<td>Sweet and savory sauces, relishes, spreads, and seasonings</td>
</tr>
<tr>
<td>3</td>
<td>Confections</td>
<td>Desserts, candies, and sweet breads</td>
</tr>
<tr>
<td>4</td>
<td>Dairy Products</td>
<td>Cheeses</td>
</tr>
<tr>
<td>5</td>
<td>Grains/Cereals</td>
<td>Breads, crackers, pasta, and cereal</td>
</tr>
<tr>
<td>6</td>
<td>Meat/Poultry</td>
<td>Prepared meats</td>
</tr>
<tr>
<td>7</td>
<td>Produce</td>
<td>Dried fruit and bean curd</td>
</tr>
<tr>
<td>8</td>
<td>Seafood</td>
<td>Seaweed and fish</td>
</tr>
</tbody>
</table>