



NOSQL DATABASES

Evann De Bailliencourt, Stéphane Hamaili, Amaia Peñagaricano, Aiyim Raikhanova

SUMMARY

- INTRODUCTION
 - Facts
 - Timeline
 - Benefits
 - Features
- COMPARISON WITH MONGODB
- HOW TO USE IT
- DEMO

FACTS

- RavenDB is ranked among the top 10 Document Databases Worldwide (Source DB-Engines).
- Over 1,000 organizations use RavenDB for their data needs.
- RavenDB was the first NoSQL Database to become Fully Transactional.
- RavenDB has 1.5 million instances of RavenDB running throughout its 37,000 locations.
- RavenDB is Open Source.



TIMELINE

- MAY 2010 : RavenDB 1.0 becomes the pioneer Document Database to offer fully transactional.
- JANUARY 2013: RavenDB 2.0 is released.
- JULY 2019: RavenDB Cloud is released.
- DECEMBER 2019: RavenDB Cloud is launched on Google Cloud Platform. Latest stable 4.2.6

- SPRING 2020: RavenDB 5.0 scheduled to include time-series data for IoT applications

 RAVEN DB

 **RAVENDB**
open source 2nd generation document DB

 **RAVENDB**
Safe by Default, Optimized for Efficiency

 **RAVENDB**

BENEFITS

- **Easy to use**
 - Either by developers and non-technical people
- **Easy to install**
 - Downloading the executable, extracting and running it
- **Raven studio**
 - Front-end to interact with RavenDB. It's included with any license, including free community version.



FEATURES

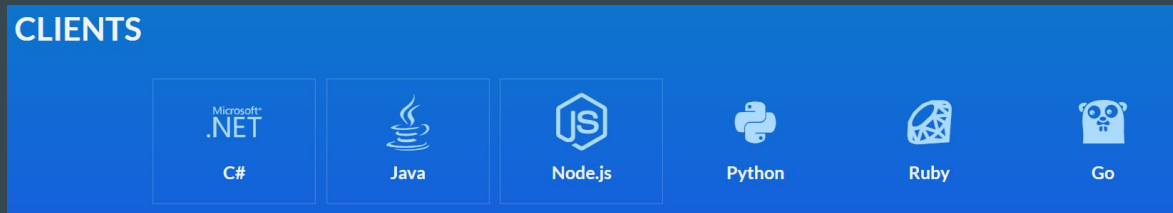
- **Multi-platform**

- Runs on Windows, Linux, macOS, Docker, Raspberry Pi and others.



- **Multiple Clients**

- Can be accessed using the major programming languages in the market, including C#, Java, Node, Python, Ruby and Go!.



FEATURES

- **Transactional**

- Is the first non-relational database to achieve ACID across the entire database. Maintain the best of SQL while boosting your capacity to the next level.

- **Management options**

- You can set up a distributed data cluster in minutes. Replicate your database in real time so you can be everywhere at once, and always available to your users.

- **Index Support**

- Indexes in RavenDB are one of the strongest points. Being a NoSQL database, it's an intelligent solution to avoiding multiple requests to database by merging multiple tables.

FEATURES

- **Simple CRUD**

- Especially important to developers. It means easily Creating/Updating/Deleting records, quicker to test and to release, no migration scripts, simple and powerful api.

- **Simple NoSQL Querying**

- RavenDB has powerful ways to do query including geospatial, faceted, full-text and map-reduce operations. But since most of the time, what you want is very easily available to you via the api and indexes.

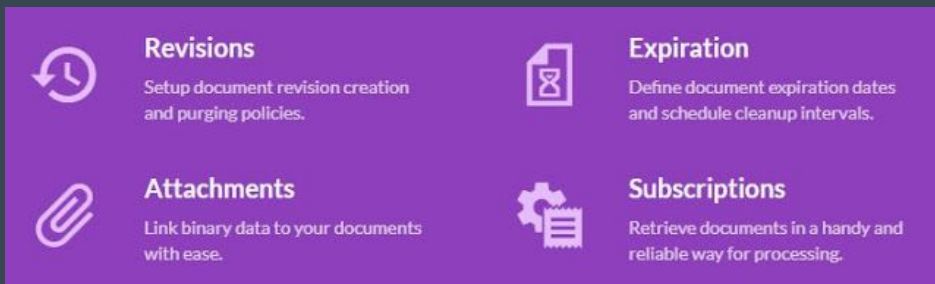
FEATURES





- **In-Memory Database**

- You can use to persist data from your application or better yet, use in your tests so you have real db operations (in-memory) without hassle of creating mocks and simulating your tests.

- **Extensions / Bundles**

- You can extend the database with bundles and extensions. Some come already built for us, others can be created through what they call Bundles. It's more of a technical solution but very interesting feature to have available.



 Revisions Setup document revision creation and purging policies.	 Expiration Define document expiration dates and schedule cleanup intervals.
 Attachments Link binary data to your documents with ease.	 Subscriptions Retrieve documents in a handy and reliable way for processing.

COMPARISON WITH MONGODB

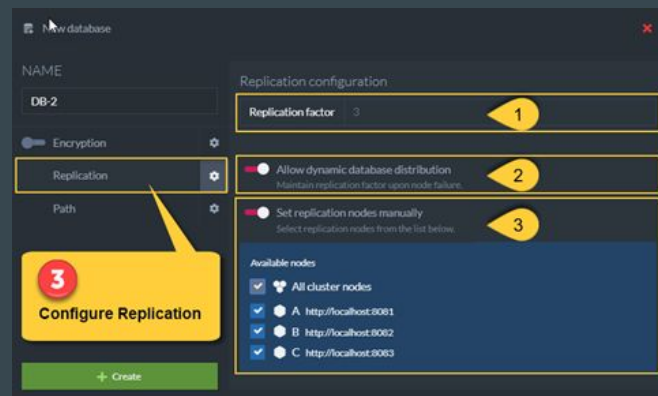
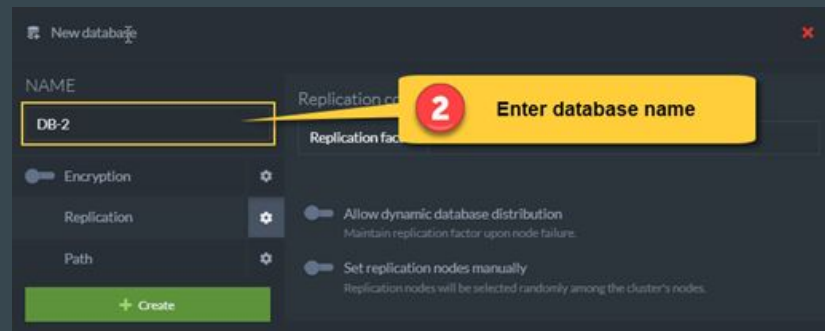
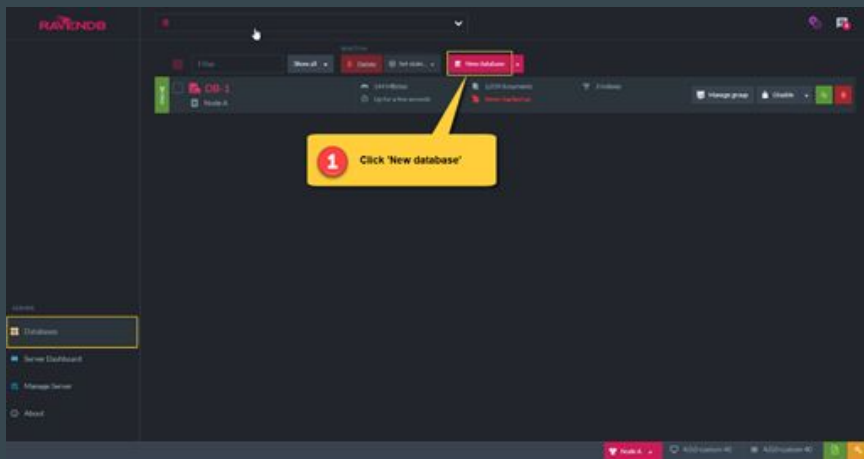
		
Primary DB Model	Document Store	Document Store
Rank (Documents Stores)	#1	#14
Replication Methods	Master-slave replication	Multi-master replication
MapReduce	yes	yes

(Source: <https://ravendb.net/articles/ravendb-vs-mongodb-performance-cost-and-complexity>)

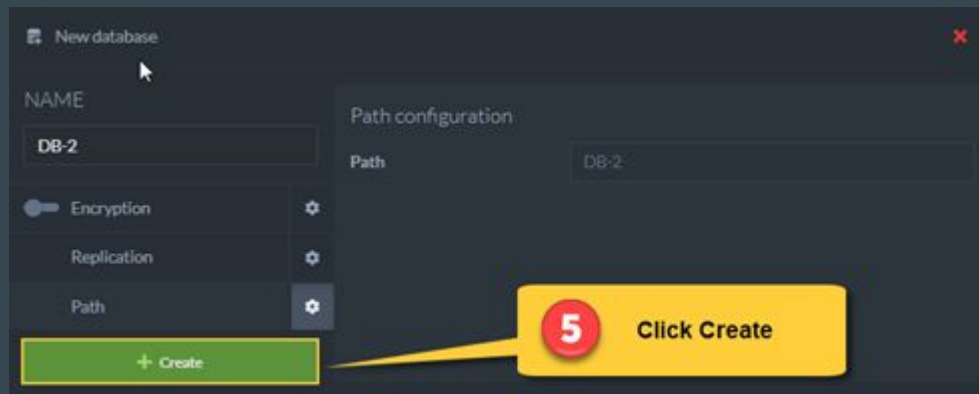
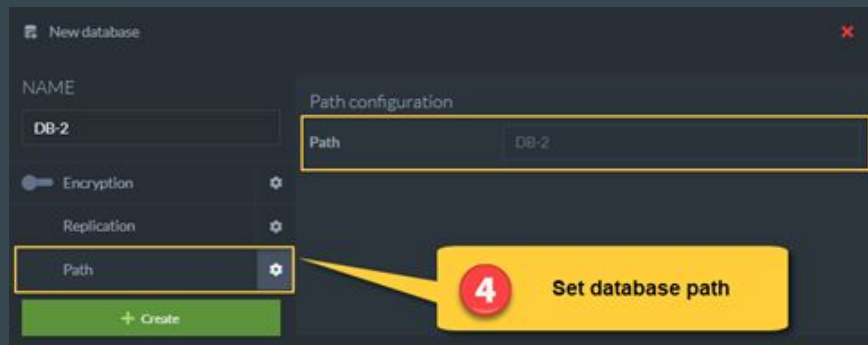
HOW TO USE IT



CREATING



CREATING



THE DOCUMENT STORE

The Document Store is the main Client API object that establishes the communication between your client application and the RavenDB cluster.

```
1 public class DocumentStoreHolder
2 {
3     private static readonly Lazy<IDocumentStore> _store = new Lazy<IDocumentStore>(CreateDocumentStore);
4
5     private static IDocumentStore CreateDocumentStore()
6     {
7         string serverURL = "http://localhost:8080";
8         string databaseName = "YourDatabaseName";
9
10        IDocumentStore documentStore = new DocumentStore
11        {
12            Urls = new[] {serverURL},
13            Database = databaseName
14        };
15
16        documentStore.Initialize();
17        return documentStore;
18    }
19
20    public static IDocumentStore Store
21    {
22        get { return _store.Value; }
23    }
24 }
```

THE SESSION

The session, which is derived from the Document Store, is the primary way your client code interacts with your RavenDB databases.

```
1  using (IDocumentSession session = DocumentStoreHolder.Store.OpenSession("YourDatabaseName"))
2  {
3      // Run your business logic:
4      //
5      // Store documents
6      // Load and Modify documents
7      // Query indexes & collections
8      // Delete documents
9      // .... etc.
10
11     session.SaveChanges();
12 }
```

Querying

Raven Query Language(RQL) allows you to execute all available types of queries and is a part of our JavaScript patching API.

</> RQL

```
from Orders // select
where Lines.Count > 4 // filter
select Lines[].ProductName as ProductNames, // project
    OrderedAt, ShipTo.City
```


Querying. Basics

- **Indexes are used by RavenDB to satisfy queries.** Each query in RavenDB must be expressed by **RQL**, our query language. Each query must match an index in order to return the results. The full query flow is as follows:
 1. **from index | collection**
 - First step. When a query is issued, it locates the appropriate index. If our query specifies that index, the task is simple - use this index. Otherwise, a query analysis takes place and an auto-index is created.
 2. **where**
 - When we have our index, we scan it for records that match the query predicate.
 3. **load**
 - If a query contains a projection that requires any document loads to be processed, they are done just before projection is executed.
 4. **select**
 - From each record, the server extracts the appropriate fields. It always extracts the `id()` field (stored by default).
 - If a query is not a projection query, then we load a document from storage. Otherwise, if we stored all requested fields in the index, we use them and continue. If not, the document is loaded from storage and the missing fields are fetched from it.
 - If a query indicates that **projection** should be used, then all results that were not filtered out are processed by that projection. Fields defined in the projection are extracted from the index (if stored).
 5. **include**
 - If any **includes** are defined, then the results are being traversed to extract the IDs of potential documents to include with the results.
 6. Return results.

Querying. Filtering

Filtering out data and return records that match a given condition.

WHERE

```
from index 'Employees/ByFirstAndLastName'  
where FirstName = 'Robert' and LastName =  
'King'
```

WHERE-NESTED/NUMERIC PROPERTY

```
from Orders where ShipTo.City = 'Albuquerque'
```

WHERE+ANY

```
from index 'Order/ByOrderLinesCount'  
where Lines_ProductName = 'Teatime Chocolate  
Biscuits'
```

WHERE+IN

```
from Products  
where endsWith(Name, 'ra')
```

WHERE+ContainsAny

```
from index 'BlogPosts/ByTags' where Tags IN  
('Development', 'Research')
```

Querying. Paging

Paging, or pagination, is the process of splitting a dataset into pages, reading one page at a time.

```
1 | const results = await session
2 |   .query({ indexName: "Products_ByUnitsInStock" })
3 |   .whereGreaterThan("UnitsInStock", 10)
4 |   .skip(20) // skip 2 pages worth of products
5 |   .take(10) // take up to 10 products
6 |   .all(); // execute query
```

Querying. Searching

Use the `Search()` extension method to perform a full-text search on a particular field. `Search()` accepts a string containing the desired search terms separated by spaces. These search terms are matched with the terms in the index being queried.

An index's terms are derived from the values of the documents' textual fields. These values were converted into one or more terms depending on which **Lucene analyzer** the index used.

Search

```
from Users where search(Name, 'John Steve')
```

Multiple Field

```
from Users where search(Name, 'Steve') or search(Hobbies, 'sport')
```

Indexes. Types of Indexes

Map indexes contain one (or more) mapping functions that indicate which fields from documents should be indexed. They indicate which documents can be searched by which fields.

Map-Reduce indexes allow complex aggregations to be performed in a two-step process. First by selecting appropriate records (using the Map function), then by applying a specified reduce function to these records to produce a smaller set of results.

Indexes

Create an index that will map documents from the `Employees` collection and enable querying by `FirstName`, `LastName`, or both.

```
public class Employees_ByFirstAndLastName : AbstractIndexCreationTask<Employee>
{
    public Employees_ByFirstAndLastName()
    {
        Map = employees => from employee in employees
                            select new
                            {
                                FirstName = employee.FirstName,
                                LastName = employee.LastName
                            };
    }
}
```



DEMO

Thank you for your time.

