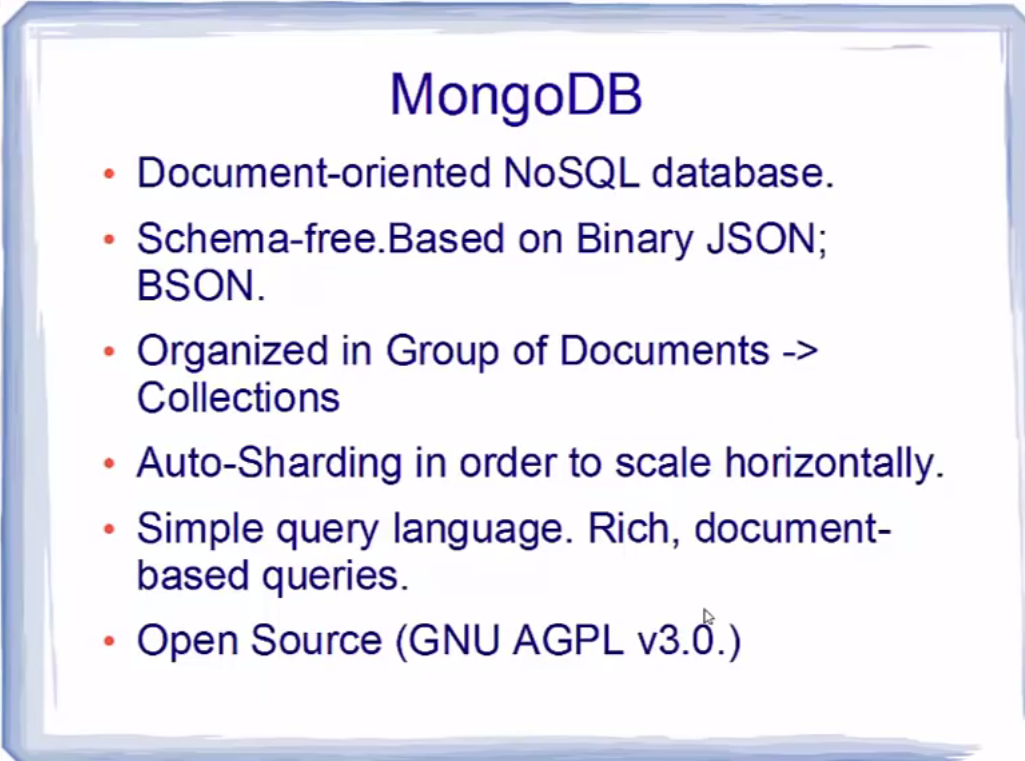
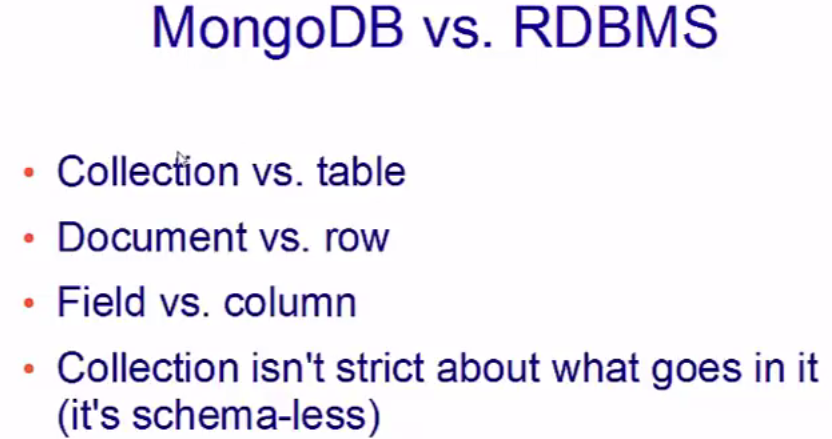
Introduction on MongoDB:

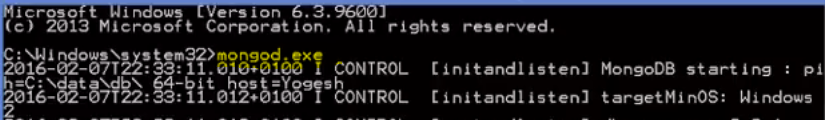


1. It is Document-oriented NoSQL open source database i.e. it is not a relational database.
2. MongoDB is a cross-platform, document-oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.
3. Relational database has tables, Columns & Rows which are related to each other but in NoSQL these collections or data are not related to each other that is why they are non-relational.
4. MongoDB is a schema free database and it is based on Binary JSON (BSON) data format. b
5. What is BJSON? In short it also called BSON.
6. MongoDB organises itself in Group of Documents 🡪 called Collections.
7. What is Auto-Sharding?
8. Using MongoDB Auto Sharding in order to scale horizontally is easier.
9. MongoDB uses simple query language, which is little bit different bit normal database queries.
10. MongoDB is written in C++.

**Difference between MongoDB and Relational Database:**



1. In MongoDB instead of table, Collection is used.
2. Inside collection we have document instead of row.
3. And inside document we have Fields instead of columns.
4. In normal RDBMS it is strict about what goes in the table, row & columns but Collection is not strict about what goes in it (because it is schema less)
5. MongoChef is used for GUI. To run this GUI, we need to run first mongod.exe from command prompt.



**MongoDB - Overview**

MongoDB works on concept of collection and document.

**Database**: Database is a physical container for collections.Each database gets its own set of files on the file system

**Collection:** Collection is a group of documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields.

**Document:** A document is nothing but equivalent to ROW of RDBMS having complete information in the form a set of key-value pairs.In Other words, …A document is a set of key-value pairs.

Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

In other words, each documents(row) could have different set of fields(columns) or structure and common fields (similar columns) could have different type of data in the collection’s document (Table’s Row)

The following table shows the relationship of RDBMS terminology with MongoDB.

|  |  |
| --- | --- |
| **RDBMS** | **MongoDB** |
| Database | Database |
| Table | Collection |
| Tuple/Row | Document |
| column | Field |
| Table Join | Embedded Documents |
| Primary Key | Primary Key (Default key \_id provided by mongodb itself) |
| **Database Server and Client** | |
| Mysqld/Oracle | mongod |
| mysql/sqlplus | mongo |

|  |
| --- |
| {  \_id: ObjectId(7df78ad8902c)  title: 'MongoDB Overview',  description: 'MongoDB is no sql database',  by: 'tutorials point',  url: 'http://www.tutorialspoint.com',  tags: ['mongodb', 'database', 'NoSQL'],  likes: 100,  comments: [  {  user:'user1',  message: 'My first comment',  dateCreated: new Date(2011,1,20,2,15),  like: 0  },  {  user:'user2',  message: 'My second comments',  dateCreated: new Date(2011,1,25,7,45),  like: 5  }  ]  } |
| \_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide \_id while inserting the document.  If you don’t provide then MongoDB provides a unique id for every document (ROW).  These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE. |

**Advantages of MongoDB over RDBMS**

* Schema less − MongoDB is a document-oriented database in which one collection(table) holds different documents(rows). Number of fields(column), content and size of the document can differ from one document(row) to another.
* No complex joins.
* Supports dynamic queries- MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
* Tuning.
* Ease of scale-out − MongoDB is easy to scale.
* Conversion/mapping of application objects to database objects not needed.

**Why Use MongoDB?**

* Document Oriented Storage − Data is stored in the form of JSON style documents.
* Index on any attribute
* Replication and high availability
* Auto-sharding
* Rich queries
* Fast in-place updates
* Professional support by MongoDB

**Where to Use MongoDB?**

* Big Data
* Content Management and Delivery
* Mobile and Social Infrastructure
* User Data Management
* Data Hub

**MongoDB - Create Database**

**What are indexes:**

Indexes are the special data structure that store a small portion of data set in the easy to parse. Indexes support sufficient resolution of queries. Without index it is very difficult to find out one document from millions of data in MongoDB. Indexes makes it easy to parse.

# MongoDB - Data Modelling

Data in MongoDB has a dynamic schema. in the same collection. They do not need to have the same set of fields or structure Common fields in a collection’s documents and each documents may hold different types of data.

## Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

### Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal\_details, Contact and, Address, you can embed all the three documents in a single one as shown below −

{

\_id:,

Emp\_ID:"10025AE336"

Personal\_details:{

First\_Name:"Radhika",

Last\_Name:"Sharma",

Date\_Of\_Birth:"1995-09-26"

},

Contact:{

e-mail:"radhika\_sharma.123@gmail.com",

phone:"9848022338"

},

Address:{

city:"Hyderabad",

Area:"Madapur",

State:"Telangana"

}

}

### Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

**Employee:**

|  |  |
| --- | --- |
| {  \_id:<ObjectId101>,  Emp\_ID:"10025AE336"  }  Personal\_details:  {  \_id:<ObjectId102>,  empDocID:" ObjectId101",  First\_Name:"Radhika",  Last\_Name:"Sharma",  Date\_Of\_Birth:"1995-09-26"  } | Contact:  {  \_id:<ObjectId103>,  empDocID:" ObjectId101",  e-mail:"radhika\_sharma.123@gmail.com",  phone:"9848022338"  }  **Address:**  {  \_id:<ObjectId104>,  empDocID:" ObjectId101",  city:"Hyderabad",  Area:"Madapur",  State:"Telangana"  } |

## Considerations while designing Schema in MongoDB

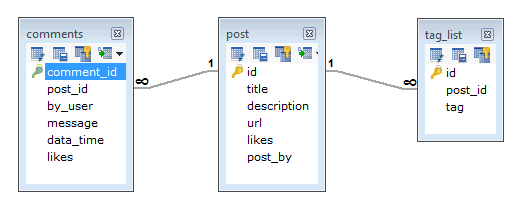
* Design your schema according to user requirements.
* Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
* Duplicate the data (but limited) because disk space is cheap as compare to compute time.
* Do joins while write, not on read.
* Optimize your schema for most frequent use cases.
* Do complex aggregation in the schema.

## Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

* Every post has the unique title, description and url.
* Every post can have one or more tags.
* Every post has the name of its publisher and total number of likes.
* Every post has comments given by users along with their name, message, data-time and likes.
* On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure −

{

\_id: POST\_ID

title: TITLE\_OF\_POST,

description: POST\_DESCRIPTION,

by: POST\_BY,

url: URL\_OF\_POST,

tags:[TAG1, TAG2, TAG3],

likes: TOTAL\_LIKES,

comments:[

{

user:'COMMENT\_BY',

message: TEXT,

dateCreated: DATE\_TIME,

like: LIKES

},

{

user:'COMMENT\_BY',

message: TEXT,

dateCreated: DATE\_TIME,

like: LIKES

}

]

}

So, while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

**JSON Fields: The DBObject object (com.ibm.nosql.json.api.DBObject)**

[**https://www.ibm.com/developerworks/data/library/techarticle/dm-1307nosqlforjson3/index.html**](https://www.ibm.com/developerworks/data/library/techarticle/dm-1307nosqlforjson3/index.html)

<http://mongodb.github.io/mongo-java-driver/3.4/javadoc/?com/mongodb/DBObject.html>

The DBObject can be used to build JSON documents from key-value pairs. The API converts the data into BSON-like format and sends it for storage to the database server. Upon retrieval, the API converts the binary BSON-like data back to DBObjects, preserving the data types.

Note: The storage format closely follows the BSON specification, but includes some IBM-specific extensions.

Keys in the DBObject are always of String type. For data values, the DB2 JSON API supports several data types, as shown in the following table.

## DBObject is an Interface : Interface DBObject and its implantation classes are

[BasicDBList](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBList.html), [BasicDBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html), [CommandResult](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/CommandResult.html), [GridFSDBFile](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/gridfs/GridFSDBFile.html), [GridFSFile](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/gridfs/GridFSFile.html), [GridFSInputFile](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/gridfs/GridFSInputFile.html), [LazyDBList](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/LazyDBList.html), [LazyDBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/LazyDBObject.html), [ReflectionDBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/ReflectionDBObject.html)

1. Class BasicDBList

|  |
| --- |
| Class BasicDBList  * [java.lang.Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)   + [java.util.AbstractCollection](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractCollection.html?is-external=true)<E>     - [java.util.AbstractList](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html?is-external=true)<E>       * [java.util.ArrayList](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html?is-external=true)<[Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)>         + [org.bson.types.BasicBSONList](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/types/BasicBSONList.html)   com.mongodb.BasicDBList   * All Implemented Interfaces:   [DBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/DBObject.html), [Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html?is-external=true), [Cloneable](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html?is-external=true), [Iterable](http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html?is-external=true)<[Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)>, [Collection](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html?is-external=true)<[Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)>, [List](http://docs.oracle.com/javase/7/docs/api/java/util/List.html?is-external=true)<[Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)>, [RandomAccess](http://docs.oracle.com/javase/7/docs/api/java/util/RandomAccess.html?is-external=true), [BSONObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/BSONObject.html)  public class BasicDBListextends [BasicBSONList](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/types/BasicBSONList.html)implements [DBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/DBObject.html) |

## Class BasicDBObject

|  |
| --- |
| * [java.lang.Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)   + [java.util.AbstractMap](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html?is-external=true)<K,V>     - [java.util.HashMap](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html?is-external=true)<K,V>       * [java.util.LinkedHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html?is-external=true)<[String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html?is-external=true),[Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)>         + [org.bson.BasicBSONObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/BasicBSONObject.html)   com.mongodb.BasicDBObject   * All Implemented Interfaces:   [DBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/DBObject.html), [Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html?is-external=true), [Cloneable](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html?is-external=true), [Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html?is-external=true)<[String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html?is-external=true),[Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true)>, [BSONObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/BSONObject.html), [Bson](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/conversions/Bson.html)  Direct Known Subclasses:  [CommandResult](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/CommandResult.html)  public class BasicDBObjectextends [BasicBSONObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/BasicBSONObject.html)implements [DBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/DBObject.html), [Bson](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/org/bson/conversions/Bson.html)  A basic implementation of BSON object that is MongoDB specific. A DBObject can be created as follows, using this class:  DBObject obj = new BasicDBObject();  obj.put( "foo", "bar" ); |
| Constructor Summary  |  | | --- | | Constructors | | **Constructor and Description** | | [BasicDBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html#BasicDBObject--)()  Creates an empty object. | | [BasicDBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html#BasicDBObject-int-)(int size)  Creates an empty object | | [BasicDBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html#BasicDBObject-java.util.Map-)([Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html?is-external=true) map)  Creates an object from a map. | | [BasicDBObject](http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html#BasicDBObject-java.lang.String-java.lang.Object-)([String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html?is-external=true) key, [Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true) value)  Creates an object with the given key/value | |

<http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html>

## Basic concepts

### JSON Objects in the DB2 JSON Java API

**JSON Namespaces: The DB object**

Obtain a handle to a DB object to access the JSON store. The DB object represents a specific JSON namespace (DB2 schema). It can be used, for example, to create and drop collections, start and commit transactions, insert JSON documents in batches, and so on. All DB objects are cached internally so they can be reused.

There are several ways to get a handle to a DB object programmatically.

* Using a JDBC DataSource object:

|  |  |
| --- | --- |
| 1  2 | javax.sql.DataSource ds =(javax.sql.DataSource)InitialContext.lookup("jdbc/DB2");  com.ibm.json.nosql.DB db = NoSQLClient.getDB (ds); |

* The DB objects are cached internally based on the datasource as the key.
* Using a JDBC URL:

|  |  |
| --- | --- |
| 1 | com.ibm.json.nosql.DB db = getDB(jdbcUrl); |

* The DB objects are cached internally based on the url+user as the key.
* Using a JDBC connection object:

|  |  |
| --- | --- |
| 1  2  3 | java.sql.Connection con =         java.sql.DriverManager.getConnection(jdbcurl, user, password);  com.ibm.json.nosql.DB db = NoSQLClient.getDB(con); |

* The DB objects are cached internally based on the url+user as the key.

**JSON Collections: The Collection object**

With the DB2 JSON feature, JSON documents are organized in collections. The JSON collections do not enforce a document structure except that a document identifier is required to make each document uniquely identifiable. It is therefore not necessary to define a table structure for a collection like for relational tables. Nevertheless, documents in a collection usually share common characteristics to make it easier to find and group data.

* **Obtaining a handle to a collection object**  
  The actual collection may or may not actually exist. For example:

|  |  |
| --- | --- |
| 1 | DBCollection coll = db.getCollection( "employee" ); |

* **Implicit creation of a collection**

If a collection does not exist, it will be automatically created with default settings when the first JSON document is inserted. In the example, it is with an employee with first name "Joe". The insert operation returns a WriteResult which wraps execution status and messages. For example:

|  |  |
| --- | --- |
| 1  2  3 | DBCollection employee = db.getCollection( "employee" ); <br>  BasicDBObject operson = new BasicDBObject("firstname", "Joe");<br>  WriteResult wr = employee.insert(operson); |

* **Create a collection explicitly**

Collections can be created automatically with default settings. However, it is often desirable to create collections explicitly with custom settings. To create a collection use the createCollection() method as follows. Optional characteristics may include directives for the datatype of identifiers, tablespaces and bufferpool usage, compression, etc.

Note that bi-temporal capabilities and time travel queries are not supported for JSON collections in DB2 for z/OS. Some restrictions also apply to collection storage characteristics, for example, the TABLESPACE option is not currently supported in the createCollection command with DB2 for z/OS.

See the reference documentation for details. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // create a collection with this name and default characteristics  DBCollection em1 = db.createCollection( "employee1" );    // create a collection "employee1" with integer identifiers  DBCollection dtc = \_db.getCollection ("employee1");  dtc.createCollection ("{\_id:'$int'}"); |

* **Renaming a collection**

Existing collections can be renamed to a new name. For example:

|  |  |
| --- | --- |
| 1  2 | DBCollection em1 = db.getCollection( "employee1" ); <br>  DBCollection em2 = em1.rename("employee2"); |

* **Dropping a collection**

Existing collections can be dropped from the JSON store. For example:

|  |  |
| --- | --- |
| 1  2 | DBCollection c = db.getCollection( "employee" );  c.drop(); |

# Writing applications with the Java API

Manage JSON documents - with and without transactions

Marion Behnen, Henry Chiu, Jyh-Chen Fang, Manish Sehgal, and Tony Sun  
Published on July 03, 2013

### Content series:

## [This content is part 3 of 3 in the series: DB2 JSON capabilities](javascript:void();)

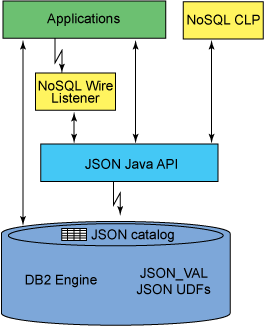
## Overview

### Introduction

DB2JSON is available with DB2 Linux, Unix and Windows 10.5 and with the IBM DB2 Accessories Suite for z/OS V3.1 to work with DB2 for z/OS V10. It provides the following features:

* A command-line shell to administer and query JSON data
* A Java™ API for application development
* A wire listener to accept and respond to requests sent over the network.

##### Figure 1. DB2 JSON components



This article explains how to use the Java interface to manage and query JSON documents in a DB2 JSON document store. It also discusses some aspects of available ACID support and limitations on combining options for transactional support with options to increase throughput.

### Prerequisites

To run the sample application, you must have at least IBM DB2 10.5 for Linux, UNIX, and Windows installed on the system, or use the IBM DB2 Accessories Suite for z/OS V3.1 to enable the use of DB2 for z/OS V10 as JSON datastore.

If you already have a JSON-enabled DB2 database available that can be used to run the samples, note the hostname or IP address and port number and continue with [Terminology](https://www.ibm.com/developerworks/data/library/techarticle/dm-1307nosqlforjson3/index.html#basics_terms). Otherwise review and complete the following steps depending on your database platform.

**IBM DB2 on Linux, UNIX and Windows**

1. Review environment variables:
   * Make sure that a Java Runtime Environment (at minimum JRE 1.5) is included in the PATH,
   * and that the CLASSPATH includes the db2nosql.jar and the JDBC driver (db2jcc.jar version 3.66 or above or db2jcc4.jar).
2. Create a database.
3. Enable the database.

The following example shows how to create and enable a database.

##### Listing 1. Create and enable the database

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE DATABASE jsondb automatic storage yes using codeset utf-8            territory US collate using system   pagesize 32 k    db2nosql -user bob -hostName bobshost -port 50000 -db jsondb -password mypwd -setup enable |

Also see Part 2 of the series, [Using the command-line interface](http://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson2/index.html) .

**IBM DB2 for z/OS**

Applications can use the DB2 JSON API or the DB2 JSON WireListener on a distributed system platform and use a DB2 Connect JDBC driver to use DB2 for z/OS as a JSON datastore.

The following components are required for this deployment scenario:

* A DB2 10.5 FP3 (or later) for Linux, Unix, and Windows server or client installation with the DB2 JSON and DB2 Driver components. After installation these components are located in SQLLIB/json.
* The DB2 Accessories Suite V3.1.
* DB2 for z/OS Version 10 with required APARs: See APAR II14727 for details about prerequisites and installation of DB2 JSON with DB2 for z/OS.

With DB2 for z/OS, enabling a database will be performed as a one-time installation task by a DB2 for z/OS Database or System Administrator. The enablement via the db2nosql command line tool is not supported for z/OS. See APAR PM98357 for details about enablement of DB2 JSON with DB2 for z/OS.

Catalog the database on the distributed system and note the connection information or the URL of the DB2 database on z/OS for use with the DB2 JSON client.

### Terminology

* **JSON store:** The DB2 database that serves as JSON document store. Before you can start using the IBM JSON solution for processing JSON documents you need to get the necessary connection and authorization information for access to the DB2 database server that serves as JSON store.
* **JSON namespace:** The DB2 SQL schema is used to fully qualify the collection and is conceptually similar to a MongoDB database.
* **JSON collection:** A collection holds a set of documents and is represented in a DB2 table. JSON collections support flexible document schemas; they do not enforce a structure.

## A simple Java application sample

Download Sample.java from the [Downloadable resources](https://www.ibm.com/developerworks/data/library/techarticle/dm-1307nosqlforjson3/index.html#artdownload) section of this article. To run this sample program in a db2cmd window, complete the following steps:

1. Setup the CLASSPATH: Include nosqljson.jar and db2jcc.jar or db2jcc4.jar into the classpath.
   * On Windows:   
     Assuming db2jcc.jar is located in the C:\Program Files\IBM\SQLLIB\java directory and nosqljson.jar is located in the C:\Program Files\IBM\SQLLIB\json\lib directory, set the CLASSPATH as follows:

|  |  |
| --- | --- |
| 1  2 | set CLASSPATH=.;C:\Program Files\IBM\SQLLIB\java\db2jcc.jar;  C:\Program Files\IBM\SQLLIB\json\lib\nosqljson.jar |

* + On UNIX:  
    Assuming db2jcc.jar is located in /sqllib/java directory and nosqljson.jar is located in /sqllib/json/lib directory, set the CLASSPATH as follows:

|  |  |
| --- | --- |
| 1 | export CLASSPATH=.:/sqllib/java/db2jcc.jar:/sqllib/json/lib/nosqljson.jar |

1. Create a test directory and copy the sample program into this test directory.
2. Modify the sample program to adapt the database name and connection information.
3. Compile and run the Java sample program in this test directory.
   * To compile the sample program:   
     javac Sample.java
   * To run the Java sample program:   
     java Sample

### Annotated Java sample program

This sample program inserts one document and submits a query to find it again. See [Sample.java](https://www.ibm.com/developerworks/data/library/techarticle/dm-1307nosqlforjson3/index.html#artdownload) for the complete sample code.

1. Get the DB object.   
   Note that DB2 uses SQL schemas as a JSON namespace to build fully qualified collection names. The DB object should therefore be obtained with the connection information and the desired namespace (DB2 schema).

|  |  |
| --- | --- |
| 1 | DB db = NoSQLClient.getDB (databaseUrl, user, password, ns); |

1. Get a collection handle.   
   The collection is represented as a table in DB2. However, the collection will not be created by obtaining a collection handle. Instead, if the collection does not exist, it will be created automatically when the first document is inserted. In this sample, the collection will be created later with this implicit approach. To create a collection explicitly, use the method db.createCollection(name, indexSpec).

|  |  |
| --- | --- |
| 1 | DBCollection col = db.getCollection ("books"); |

1. Insert a document.   
   With the attempt to insert a document, the collection will be automatically created if it does not yet exist. Also note that if the document does not contain an \_id field, the collection is created with a primary key that takes generated binary identifiers. If subsequently inserted documents contain an \_id field, the datatype of that \_id field must match with the datatype of the generated \_id.

|  |  |
| --- | --- |
| 1  2  3  4  5 | BasicDBObject json = new BasicDBObject ();  json.append ("author", "Smith");  json.append ("title", "My first book");  System.out.println ("Inserting: " + json);  col.insert (json); |

1. Submit a query.   
   Build an object with the desired search criteria and call the find method for the collection. The query is executed with the first call to hasNext() and the results are available through the cursor. Note that the query has no projection list, so the results will include all attributes, including the generated identifier.

|  |  |
| --- | --- |
| 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 | DBCursor cursor = col.find (new BasicDBObject ("author", "Smith"));   1. try 2. {    1. while (cursor.hasNext ()) {    2. DBObject obj = cursor.next ();    3. System.out.println ("Retrieved: " + obj);    4. } 3. } 4. finally 5. {    1. cursor.close (); 6. } 7. } 8. } |

Expected sample output:

* Inserted: {"author":"Smith","title":"My first book"}
* Retrieved: {"\_id":"$oid":"51bf710b416e9107ff9bc734"}, "author":"Smith","title":"My first book"}

## Basic concepts

### JSON Objects in the DB2 JSON Java API

**JSON Namespaces: The DB object**

Obtain a handle to a DB object to access the JSON store. The DB object represents a specific JSON namespace (DB2 schema). It can be used, for example, to create and drop collections, start and commit transactions, insert JSON documents in batches, and so on. All DB objects are cached internally so they can be reused.

There are several ways to get a handle to a DB object programmatically.

* Using a JDBC DataSource object:

|  |  |
| --- | --- |
| 1  2 | javax.sql.DataSource ds =(javax.sql.DataSource)InitialContext.lookup("jdbc/DB2");  com.ibm.json.nosql.DB db = NoSQLClient.getDB (ds); |

* The DB objects are cached internally based on the datasource as the key.
* Using a JDBC URL:

|  |  |
| --- | --- |
| 1 | com.ibm.json.nosql.DB db = getDB(jdbcUrl); |

* The DB objects are cached internally based on the url+user as the key.
* Using a JDBC connection object:

|  |  |
| --- | --- |
| 1  2  3 | java.sql.Connection con =         java.sql.DriverManager.getConnection(jdbcurl, user, password);  com.ibm.json.nosql.DB db = NoSQLClient.getDB(con); |

* The DB objects are cached internally based on the url+user as the key.

**JSON Collections: The Collection object**

With the DB2 JSON feature, JSON documents are organized in collections. The JSON collections do not enforce a document structure except that a document identifier is required to make each document uniquely identifiable. It is therefore not necessary to define a table structure for a collection like for relational tables. Nevertheless, documents in a collection usually share common characteristics to make it easier to find and group data.

* **Obtaining a handle to a collection object**  
  The actual collection may or may not actually exist. For example:

|  |  |
| --- | --- |
| 1 | DBCollection coll = db.getCollection( "employee" ); |

* **Implicit creation of a collection**

If a collection does not exist, it will be automatically created with default settings when the first JSON document is inserted. In the example, it is with an employee with first name "Joe". The insert operation returns a WriteResult which wraps execution status and messages. For example:

|  |  |
| --- | --- |
| 1  2  3 | DBCollection employee = db.getCollection( "employee" ); <br>  BasicDBObject operson = new BasicDBObject("firstname", "Joe");<br>  WriteResult wr = employee.insert(operson); |

* **Create a collection explicitly**

Collections can be created automatically with default settings. However, it is often desirable to create collections explicitly with custom settings. To create a collection use the createCollection() method as follows. Optional characteristics may include directives for the datatype of identifiers, tablespaces and bufferpool usage, compression, etc.

Note that bi-temporal capabilities and time travel queries are not supported for JSON collections in DB2 for z/OS. Some restrictions also apply to collection storage characteristics, for example, the TABLESPACE option is not currently supported in the createCollection command with DB2 for z/OS.

See the reference documentation for details. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // create a collection with this name and default characteristics  DBCollection em1 = db.createCollection( "employee1" );    // create a collection "employee1" with integer identifiers  DBCollection dtc = \_db.getCollection ("employee1");  dtc.createCollection ("{\_id:'$int'}"); |

* **Renaming a collection**

Existing collections can be renamed to a new name. For example:

|  |  |
| --- | --- |
| 1  2 | DBCollection em1 = db.getCollection( "employee1" ); <br>  DBCollection em2 = em1.rename("employee2"); |

* **Dropping a collection**

Existing collections can be dropped from the JSON store. For example:

|  |  |
| --- | --- |
| 1  2 | DBCollection c = db.getCollection( "employee" );  c.drop(); |

**JSON Fields: The DBObject object**

The DBObject can be used to build JSON documents from key-value pairs. The API converts the data into BSON-like format and sends it for storage to the database server. Upon retrieval, the API converts the binary BSON-like data back to DBObjects, preserving the data types.

Note: The storage format closely follows the BSON specification, but includes some IBM-specific extensions.

Keys in the DBObject are always of String type. For data values, the DB2 JSON API supports several data types, as shown in the following table.

##### Table 1. Data types

| **Datatype** | **Example in JSON format** |
| --- | --- |
| java.lang.String | "string" |
| java.lang.Integer | 3 |
| java.lang.Long | 4294967296 |
| java.lang.Double | 6.2 |
| java.lang.Boolean | true / false |
| java.lang.Byte [] | { "$binary": "(base64-encoded value)", "$type": "0" } |
| java.util.Date (millisecond precision, in UTC) | { "$date" : "1998-03-11T17:14:12.456Z" } |
| java.util.regex.Pattern | { "$regex" : "ab\*" , "$options" : "" } |
| java.util.UUID | { "$uuid" : "fb46f9a6-13df-41a2-a4e3-c77e85e747dd" } |
| com.ibm.nosql.bson.types.ObjectId | { "$oid" : "51d2f200eefac17ea91d6831" } |
| com.ibm.nosql.bson.types.Code | { "$code" : "mycode" } |
| com.ibm.nosql.bson.types.CodeWScope | { "$code" : "i=i+1", "$scope" : {} } |
| com.ibm.nosql.json.api.BasicDBObject | { "a" : 1, "b": { "c" : 2 } } |
| com.ibm.nosql.json.api.BasicDBList | [1 , 2, "3", "abc", 5] |

For Date string values, the client converts the value to UTC to be stored in DB2, and it is also retrieved as a Date in UTC format.

The following example shows how to insert field values with different data types.

##### Listing 2. Insert a JSON document with various data types as a DBObject

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | BasicDBObject jsonObj = new BasicDBObject ();  jsonObj.append ("\_id", new ObjectId ("51d2f200eefac17ea91d6831"));    SimpleDateFormat df = new SimpleDateFormat ("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");  df.setCalendar (Calendar.getInstance (TimeZone.getTimeZone ("UTC")));  try {    jsonObj.append ("date", df.parse ("1998-03-11T17:14:12.456Z"));  }  catch (ParseException e) {    e.printStackTrace();  }  jsonObj.append ("pattern", Pattern.compile ("ab\*", BSON.regexFlags ("")));  jsonObj.append ("code", new Code ("mycode"));  jsonObj.append ("codews", new CodeWScope ("i=i+1", new BasicBSONObject ()));  jsonObj.append ("null", null);  jsonObj.append ("uuid",                  UUID.fromString ("fb46f9a6-13df-41a2-a4e3-c77e85e747dd"));  jsonObj.append ("str", new String ("hello world"));  jsonObj.append ("int", new Integer (3));  jsonObj.append ("long", new Long (4294967296L));  jsonObj.append ("double", new Double (6,2));  jsonObj.append ("bool", new Boolean (true));  jsonObj.append ("subobject",  new BasicDBObject ()      .append ("a", 1)      .append ("b", new BasicDBObject ().append ("c", 2))    );  jsonObj.append ("array",    new BasicDBList (1, 2, "3", "abc", 5)    );  jsonObj.append ("objarray",    new BasicDBList (      new BasicDBObject ().append ("name", "Joe").append ("age", 5),      new BasicDBObject ().append ("name", "Mary").append ("age", 4)    ));    System.out.println ("Inserting: " + jsonObj);  c.insert (jsonObj); |

Instead of creating key-value pairs as DBObjects, it is also possible to insert JSON documents in string format. The API will internally parse the input string into objects. The following example shows how to insert field values with different data types in a string representation.

##### Listing 3. Insert a JSON document with various data types as a string

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | String jsonString =  "{"    + "\_id : { $oid : \"51d2f200eefac17ea91d6831\" },"    + "date : { $date : \"1998-03-11T17:14:12.456Z\" },"    + "pattern : { $regex : \"ab\*\" , \"$options\" : \"\" },"    + "code : { $code : \"mycode\"},"    + "codews : { $code : \"i=i+1\" , \"$scope\" : { }},"    + "null :  null,"    + "uuid : { $uuid : \"fb46f9a6-13df-41a2-a4e3-c77e85e747dd\" },"    + "str : \"hello world\","    + "int: 3,"    + "long : 4294967296,"    + "double : 6.2,"    + "bool : true,"    + "subobject:{ a:1, b:{ c: 2 } },"    + "array : [1 , 2, \"3\", \"abc\", 5],"    + "objarray : [{name:\"Joe\", age:5}, {name:\"Mary\", age:4}]"   + "}";    System.out.println ("Inserting: " + jsonString);  c.insert (jsonString); |

Output (formatted for readability):

##### Listing 4. Output from inserting a JSON document as a string

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Inserting: {\_id : { $oid : "51d2f200eefac17ea91d6831" },  date : { $date : "1998-03-11T17:14:12.456Z"  },  pattern : { $regex : "ab\*" , "$options" : "" },  code : { $code : "mycode"},  codews : { $code : "i=i +1" , "$scope" : { }},  null :  null,  uuid : { $uuid : "fb46f9a6-13df-41a2-a4e3-c77e85e747dd" },  str :  "hello world",  int: 3,  long : 4294967296,  double : 6.2,  bool : true,  subobject:{ a:1, b:{ c: 2 } },  array :  [1 , 2, "3", "abc", 5],  objarray : [{name:"Joe", age:5}, {name:"Mary", age:4}]} |

## Document operations

### Inserting JSON documents

The DB2 JSON API provides DBCollection.insert() functions to insert JSON documents in DB2. Each document is stored as a BLOB in BSON-like format with a unique identifier.

When inserting a document into a collection, the DB2 JSON API will automatically create the table if it does not exist. In such a case, if the first document inserted contains an \_id field, the field is used as an unique identifier. If the first document does not contain an \_id field, a unique identifier will be automatically generated and the document inserted with this generated \_id. In either case, subsequently inserted documents must then also contain a unique \_id field of the same data type. For example:

##### Listing 5. Inserting a document into a collection

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | DB db = NoSQLClient.getDB (jdbcUrl, user, pass);  DBCollection c = db.getCollection ("books");    // Create a document to insert  BasicDBObject json = new BasicDBObject ();  json.append ("isbn", "123-456-789");  json.append ("author", "Verne, Jules");  json.append ("title", "Journey to the Center of the Earth");  json.append ("abstract", "Classic science fiction novel in an unusual setting");  json.append ("price", 6.00);  json.append ("pages", 276);  json.append ("category", "Fantasy");    System.out.println ("Inserting: " + json);    // If the table "books" does not exist, it is automatically created  c.insert (json); |

Output (formatted for readability):

##### Listing 6. Formatted output

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Inserting: {"isbn":"123-456-789",  "author":"Verne, Jules",  "title":"Journey to the Center of the  Earth",  "abstract":"Classic science fiction novel in an unusual  setting",  "price":6.0,  "pages":276,  "category":"Fantasy"} |

### Finding JSON documents

The DBCollection provides several options to count, find, and aggregate data in collections. You can specify the conditions that need to be matched by the documents (the query) and determine which attributes should be retrieved (the projection list).

To specify the projection list, use 0 to exclude attributes, 1 to include attributes. You cannot mix inclusion and exclusion, except to exclude the \_id which is otherwise always included automatically.

Methods that return a DBCursor also allow controlling the result set through cursor directives, including setting a maximum size of the result set (limit(n)) and paging (skip(n)).

Note that the DBCursor object represents a forward-only cursor that iterates over results and is only executed with next/hasNext.

The following listing shows some examples how to search JSON documents. For a list of supported comparison and boolean operators, see the Reference documentation and also refer to Part 2 of the series [Using the command-line interface](http://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson2/index.html) .

##### Listing 7. Search documents

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | // Find all documents with author 'Verne, Jules'  DBCursor cursor = col.find (new BasicDBObject ("author", "Verne, Jules"));      // Find all documents with author 'Smith' and price greater 6.20;  // include title and price (the \_id is  automatically included)    BasicDBObject match = new BasicDBObject();  match.append("author", "Smith");  match.append("price", new BasicDBObject("$gt", 6.20));    BasicDBObject projection = new BasicDBObject();  projection.append("title", 1);  projection.append("price", 1);  DBCursor cursor2 = col.find (match, projection);      // Re-use the query, but exclude the \_id  projection.append("\_id", 0);  cursor = col.find (match, projection);      // Find books with sales greater 1000, sort by sales, get first 2 only  BasicDBObject fproject = new BasicDBObject("author", 1);  fproject.put ("title", 1);  fproject.put ("sales", 1);  fproject.put ("rating", 1);  BasicDBObject cond = new BasicDBObject("sales", new BasicDBObject("$gt",1000));    DBCursor fcursor = dtc.find(cond, fproject);  fcursor.sort (new BasicDBObject("sales", 1));  fcursor.limit (2);    while (fcursor.hasNext()) {     System.out.println(fcursor.next());  }      // Count entries where sales are less than 1000  long count = dtc.getCount (new BasicDBObject ("sales",        new BasicDBObject("$lt",1000)));      // Find distinct authors  List result = dtc.distinct("author");      // Get distinct data sorted by author, get first 2 entries only  BasicDBObject query = new BasicDBObject ("topic", "Mystery");  BasicDBObject keyObject = new BasicDBObject("author", 1);  keyObject.put ("rating", 1);    DBCursor dcursor = dtc.distinctAsCursor(keyObject, query);  dcursor.sort (new BasicDBObject("author", 1));    while (dcursor.hasNext()) {     System.out.println(dcursor.next());  } |

To access values in nested objects or in arrays, use the dot notation for array values, that is, the attribute name followed by the position (starting with 0). For example, with an array of [1, 2, "3", "abc", 5] the value of array.3 is "abc".

To access values in nested objects, specify the full path with a dot-notation. For example, with a nested document like {customer: {name: {"Joe"}, {state: "AZ"}} the value of customer.state is "AZ".

### Updating JSON documents

The DB2 JSON API also provides DBCollection.update() functions to update JSON documents that satisfy the search criteria. Specific fields in the document can be updated or appended, or the entire document can be replaced. The following example shows an update to the "pages" field for documents with an "isbn" of "123-456-789".

##### Listing 8. Update a specific field with the $set operator

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | DB db = NoSQLClient.getDB (jdbcUrl, user, pass);  DBCollection c = db.getCollection ("books");    // Update  c.update (new BasicDBObject ("isbn", "123-456-789"),            new BasicDBObject ("$set", new BasicDBObject ("pages", 299)),            false,            true);    // Find document  cursor = c.find (new BasicDBObject ("isbn", "123-456-789"));  System.out.println ("Retrieved: " + cursor.next ());  cursor.close (); |

##### Listing 9. Output (formatted for readability)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Retrieved: {"\_id":{"$oid":"51a7a4d9cd862910f0992b33"},  "isbn":"123-456-789",  "author":"Verne,  Jules",  "title":"Journey to the Center of the Earth",  "abstract":"Classic science fiction novel in an  unusual setting",  "price":6.0,  "pages":299,  "category":"Fantasy"} |

If the third parameter of update (upsert) is set to true, the document will be:

* updated if the document exists, or
* inserted if it does not exist.

If the fourth parameter of update (multi) is set to true, all documents matching the query will be updated. If it is set to false, only the first document that matches the query will be updated.

The $set operator replaces the old value with the new value if the field exists; otherwise it inserts the field into the document. The $unset operator can be used to delete a particular field.

**Caution:** Using update without the $set or $unset operator will replace the content of the document with the specified fields. Make sure to use $set or $unset if you intend to keep other fields in the document.

### Removing or deleting JSON documents

The DBCollection.remove() function deletes JSON documents from a collection. The remove function also takes a query object to define conditions to specify a subset of documents for removal. The collection and any indexes are preserved.

For example, to remove all documents from collection "books", use:

|  |  |
| --- | --- |
| 1  2  3  4 | DB db = NoSQLClient.getDB (jdbcUrl, user, pass);  DBCollection c = db.getCollection ("books");    c.remove (); |

### Importing or exporting JSON Files

Collection data can be imported into a collection and data can also be exported from a collection. The format used is JSON, and the files use a .js extension.

To **import** a collection, use the importFile method:

|  |  |
| --- | --- |
| 1  2 | DBCollection dtc = dtc.getCollection("docs"); <br>  dtc.importFile("C:/docs.js",10); <br> |

The method locates the file named docs.js and imports the content with a commit frequency of 10. Committing less often, that is, decreasing the commit frequency, increases the import performance.

Imports read each file line by line; carriage return and such line end characters therefore must be avoided inside a document. Errors will be thrown for each line that does not adhere to JSON format standards, and for documents that contain a duplicate identifier or an identifier with a different data type than expected for the collection. However, processing will then continue with the next line and valid data imported.

With current documents in the collection of this structure:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Row 1:  {  "\_id":1,  "x":1  } |

... and a document in the import file as follows:

{"\_id":"abc","abstract":"Spiders and Dragons","author":"Tolkien, J.R","isbn":"122-456-789","pages":216, "price":5.0,"title":"The Hobbit"}

... an import will throw a conversion error because the \_id for the collection is an integer and the \_id for the book is a string.

The structure of the document will be preserved during the import, including any nested documents.

To **export** a collection into a file, call the exportFile method:

|  |  |
| --- | --- |
| 1  2 | DBCollection dtc = dtc.getCollection("docs");  dtc.exportFile("C:/exported.js") |

This will export the "docs" collection into the file named "exported.js". The exported file will be a standard JSON file.

## Working with indexes

When a field in a JSON document is often used as selection criteria in queries, creating an index on the field can speed up data retrieval times for larger workloads.

For an index to be effective, it should be highly selective (field values should be mostly unique). Fields that are more selective require fewer index reads to find documents that satisfy the query.

Indexes can improve retrieval performance, but at the cost of slower writes and storage. Too many indexes on documents that are frequently updated can slow down inserts, updates, and deletes since indexes need to be updated with each operation.

### Creating an index

When a collection is created, an index on the identifier is automatically created as unique index. To add indexes for other fields, use the DBCollection.ensureIndex() function. Indexes can be created on a single field or multiple fields.

Note: For DB2 on Linux, Unix and Windows, indexes can be in ascending and descending order and indexing an element in an array is supported. With DB2 for z/OS, indexes can only be in ascending order and indexes on arrays are not supported.

To create an index, the data type of the field to be indexed needs to be specified. If the search value in the query has a different type than what is specified by the index, the index will not be used for query execution.

In the following example, the index option array is set to false since the field 'copy' is not an array or contained in an array.

##### Listing 10. Create an ascending index on field 'copy' with type integer

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | DB db = NoSQLClient.getDB (jdbcUrl, user, pass);  DBCollection c = db.getCollection ("books");    // insert 100 books, with different values for field 'copy'  for (int i = 1; i <= 100; i++)  {    BasicDBObject json = new BasicDBObject ();    json.append ("isbn", "123-456-711");    json.append ("author", "Verne, Jules");    json.append ("title", "Journey to the Center of the Earth");    json.append ("abstract", "Classic science fiction novel in an unusual setting");    json.append ("price", 6.00);    json.append ("pages", 276 + i);    json.append ("category", "Fantasy");    json.append ("copy", i);      System.out.println ("Inserting: " + json);    c.insert (json);  }      // create index on field 'copy'  c.ensureIndex (  // 1 for ascending, $int is data type  new BasicDBObject ("copy", new BasicDBList (1, "$int")),  // index option 'array' explicitly set to false  new BasicDBObject ("array", false));    // find document where 'copy' = 1  DBCursor cursor = c.find (new BasicDBObject ("copy", 1));    // call to DBCursor.next() will execute the query and utilize the index  System.out.println ("Retrieved: " + cursor.next ());  cursor.close (); |

Index option unique can be set to true if a unique index is desired. If there are existing documents, ensure there are no duplicate values for the same field.

### Dropping an index

To drop an index, use the DBCollection.dropIndex() function.

The following example uses dropIndex() with the same specification that was used to create the index.

|  |  |
| --- | --- |
| 1 | c.dropIndex (new BasicDBObject ("copy", new BasicDBList (1, "$int"))); |

**Note:** the function DBCollection.removeIndex() will also drop the index.

## Performance features and transactions

### Lazy fetch

Queries using the DBCollection.find() methods return a DBCursor object, which represents a forward-only cursor that iterates over results. By default, queries use a method where all of the selected data is fetched eagerly into memory. To fetch blocks of large results as you iterate over the cursor, use lazy fetch (DBCursor.lazyFetch()).

**Caution:** Lazy fetch must be specified before the cursor is opened. It is important to close the cursor after fetching results; otherwise memory leaks may occur.

The following example shows how to set the lazy fetch option on the cursor.

##### Listing 11. Use lazy fetch to retrieve JSON documents

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | DB db = NoSQLClient.getDB (jdbcUrl, user, pass);  DBCollection c = db.getCollection ("customer");    DBCursor cur = c.find ().lazyFetch (); // specify lazy fetch    try  {     // iterate through all documents     while (cur.hasNext ())     {      // not pre-fetching, we fetch as we go      DBObject cust = cur.next ();      System.out.println (cust);     }  }  finally  {     // make sure to close the cursor to prevent leaks     cur.close ();  } |

### Transactions

There are two options to connect to the DB2 JSON store, either through a single-mode connection which is explicitly established with the provided connection information, or through a shared connection pool.

By default, a connection pool is used, and operations like insert(), update(), and so on, will attempt to obtain a connection from the pool, execute the operation with that connection, and return it to the pool when done.

For connections in single-mode, the DB2 JSON API allows control of transactional behavior when working with JSON documents. It is thus possible to combine multiple operations in a transaction, control auto-commit behavior, and trigger a rollback if errors occur.

##### Table 2. Methods on DB to control transactional behavior

|  |  |
| --- | --- |
| public void startTransaction() | Gets a connection if needed, sets auto-commit to false. If a connection pool or datasource is being used, puts the DB in single connection mode until a future commitTransaction() or rollbackTransaction(). |
| public void commitTransaction() | Commits a transaction started by startTransaction(). |
| public void rollbackTransaction() | Rollback a transaction that was started by startTransaction(). |
| public void setAutoCommit(boolean autoCommit) | Sets auto-commit if DB2 is using single connection mode. It is an error to call this method in DataSource mode. |

**Note:** The transaction APIs described in this table are not applicable for the "Fire and Forget" mode.

It is recommended to explicitly obtain a single-mode connection when using the transaction APIs to avoid situations where starting a transaction forcibly changes the connection pool mode into single-mode.

The following example illustrates how to obtain a single-mode connection and commit the transaction in a Java program using the DB2 JSON API. When db.commitTransaction() is executed successfully, the document for customer Joe and the document with Joe's order of a refrigerator will be inserted into the customer and order collections, respectively. None of the documents will be inserted if an error occurs for any of the documents. With these APIs, it is guaranteed that either all or none of the documents will be inserted.

##### Listing 12. Transactions with single-mode connection

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | // use the url, user and password to create a JDBC connection.  Connection con = DriverManager.getConnection (url, user, password);    // get a db object with this connection  DB db = NoSQLClient.getDB (con);    // get a handle to the affected collections  DBCollection dtc = db.getCollection("customer");  DBCollection dto = db.getCollection("order");    // start a transaction for this db object.  db.startTransaction ();   try{     // Create JSON object:     BasicDBObject json = new BasicDBObject ();     json.append ("name", "Joe");     json.append ("cid", 12);     dtc.insert(json);       json = new BasicDBObject ();     json.append ("product", "refrigerator");     json.append ("cid", 12);     dto.insert(json);     db.commitTransaction ();   } catch (Exception ex) {     db.rollbackTransaction();   } finally{     } |

### Fire and forget mode

"Fire and forget" mode enables multi-threaded, asynchronous inserts and can be set on the collection to enhance performance for inserts. The downside to using this mode is that the data is not guaranteed to be written to the server. Moreover, the application will not see an exception raised if an error did occur during an insert. However, for application scenarios that can tolerate loss of data, the performance gain from using this mode can be significant.

Fire and forget mode can only be activated when applications use a connection pool. If an application enables fire and forget for a single connection, the mode setting is ignored and the insert will be executed single-threaded.

The number of threads used for fire and forget is 10 by default. This value can be changed by setting the asyncMaxThreadCount in the nosql.properties file. For example, to set the number of threads to 100, use nosql.asyncMaxThreadCount=100 .

To enable fire and forget mode, the collection must set the WriteConcern value to either NONE or NORMAL. Other WriteConcern values such as SAFE and JOURNAL\_SAFE disable fire and forget mode because they guarantee writes to the database. See the Java documentation for more information on WriteConcern.

In the following example, a collection handle dtc that maps to a collection named "firenforget" is obtained from the DB object \_db. When dtc.setWriteConcern is called with option NONE, fire and forget mode is enabled.

|  |  |
| --- | --- |
| 1  2 | DBCollection dtc = \_db.getCollection("firenforget");  dtc.setWriteConcern (WriteConcern.NONE); |

To begin inserting in Fire and Forget mode, use the same insert code as for other insert modes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | for(int i=0; i>100; i++){    DBObject obj = BasicDBObjectBuilder.start().             append("name", "Joe"+i).             add("nums", new  BasicDBList(i+1, i+2, i+3)).get();      dtc.insert(obj);    }   \_db.waitQueue(); |

The above code will insert 100 documents using multiple threads. The code is similar to non fire and forget mode inserts, however, take note of this statement:

\_db.waitQueue()

The application will wait until all the inserts have been processed and subsequent references to the collection in the application will access the collection only after the inserts have completed. Without this method call a search such as DBCursor cursor = dtc.find(new BasicDBObject("nums", 1)); might return an empty collection because the inserts may not have been completed after the find() method is executed.

The code in the example above ignores potential insert errors. To retrieve insert errors that may have occurred during processing, use the getLastError() method as shown in the following example.

|  |  |
| --- | --- |
| 1  2  3 | WriteResult wr = dtc.insert(obj);    CommandResult cr =  wr.getLastError (); |

For more details on fire and forget mode, please refer to the WriteResult class and DBCollection class in the Java documentation of the DB2 JSON API.

### Batching

Batching accumulates documents and then sends multiple documents together to the JSON store instead of each one separately. The DB2 JSON API provides a programmatic way of batching several JSON documents together. The major advantage of batching is better performance.

To batch documents, apply startBatch() to mark the starting of a batch, and endBatch() to trigger the execution of the operation (insert, update, remove or save) for the documents. Note that if an operation in the batch fails, processing will continue with the next operation in the batch.

The DB2 JSON API supports two types of batching: homogeneous batching and heterogeneous batching.

* **Homogeneous batching:** This means that all JSON documents that are batched are part of the same collection, and the same operation is applied to all documents.

The following example code uses batching to insert 3 documents in one collection:

##### Listing 13. Homogeneous batching

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | DBCollection batch = db.getCollection("batch");    BasicDBObject dbObj1 = new BasicDBObject("name", "Joe1");  dbObj1.put("\_id", 1);    BasicDBObject dbObj2 = new BasicDBObject("name", "Joe2");  dbObj2.put("\_id", 2);    BasicDBObject dbObj3 = new BasicDBObject("name", "Joe3");  dbObj3.put("\_id", 3);    db.startBatch();    batch.insert(dbObj1);  batch.insert(dbObj2);  batch.insert(dbObj3);    db.endBatch(); |

* **Heterogeneous batching:** JSON documents that are batched together can be part of different collections.

The following example code uses batching to insert three documents in collection "batch1" and two documents in collection "batch2".

##### Listing 14. Heterogeneous batching

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | DBCollection batch1 = db.getCollection("batch1");    BasicDBObject dbObj1 = new BasicDBObject("name", "Joe1");  dbObj1.put("\_id", 1);    BasicDBObject dbObj2 = new BasicDBObject("name", "Joe2");  dbObj2.put("\_id", 2);    BasicDBObject dbObj3 = new BasicDBObject("name", "Joe3");  dbObj3.put("\_id", 3);    DBCollection batch2 = \_db.getCollection("batch2");    BasicDBObject dbObj4 = new BasicDBObject("name", "Joe4");  dbObj4.put("\_id", 4);    BasicDBObject dbObj5 = new BasicDBObject("name", "Joe5");  dbObj5.put("\_id", 5);    db.startBatch();    // inserting documents to collection batch1    batch1.insert(dbObj1);  batch1.insert(dbObj2);  batch1.insert(dbObj3);    // inserting documents to collection batch2    batch2.insert(dbObj4);  batch2.insert(dbObj5);    db.endBatch(); |

The above examples show an insert operation being used in a batch but you can also batch other operations like update, drop or remove, and save, as well as combine different operations in a batch.

## Conclusion

This article has introduced some basic features of the DB2 JSON Java API and described options to control transactions and improve throughput. To find out more about DB2 JSON, see how to work with the command-line interface, or how to use the wire listener to receive and handle requests, see the other articles in this series. Details about DB2 JSON Java interfaces can also be found in the DB2 JSON reference documentation, in particular the provided Java documentation.

#### Downloadable resources

* [PDF of this content](https://www.ibm.com/developerworks/data/library/techarticle/dm-1307nosqlforjson3/dm-1307nosqlforjson3-pdf.pdf)
* [Sample Java for this article](http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=936357&filename=Sample.zip&method=http&locale=) (Sample.zip | 1KB)

#### Related topics

* Access the [DB2 Information Center](http://pic.dhe.ibm.com/infocenter/db2luw/v10r1/index.jsp).
* Download a trial version of [DB2 on Linux, Unix and Windows](http://www-01.ibm.com/software/data/db2/linux-unix-windows/downloads.html) to evaluate the DB2 JSON support for yourself.