NoSQL Database

NoSQL Database is referred as a non-SQL or non relational or Not Only SQL database. It provides a mechanism for storage and retrieval of data other than the tabular relations model used in relational databases. NoSQL databases don't use tables for storing data. It is generally used to store big data and real-time web applications.

History behind the creation of NoSQL Databases

- In the early 1970, Flat File Systems were used. Data was stored in flat files and the biggest problems with flat files are each company implements their own flat files and there are no standards. It is very difficult to store data in the files, retrieve data from files because there is no standard way to store data.
- Then the relational database was created by E.F. Codd and these databases answered the question of having no standard way to store data. But later relational database also get a problem that it could not handle big data, due to this problem there was a need of database which can handle every types of problems then NoSQL database was developed.

Features of NoSQL

Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn’t require object-relational mapping and data normalization
- No complex features like query languages, query planners,referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language

- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

Types of NoSQL Databases

NoSQL Databases are mainly categorised into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

1. Key-value Pair Based
2. Column-oriented Graph
3. Graphs based
4. Document-oriented

Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- Object-oriented programming which is easy to use and flexible

- NoSQL databases don’t need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity
- Excels at distributed database and multi-data center operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn’t work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

Introduction to MongoDB

MongoDB is a non-relational document database that provides support for JSON-like storage. The MongoDB database has a flexible data model that enables you to store unstructured data, and it provides full indexing support,

- MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time

- The document model maps to the objects in your application code, making data easy to work with
- Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data
- MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use
- MongoDB is free to use.

MongoDB Datatypes

Following is a list of usable data types in MongoDB.

Data Types	Description
String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.
Integer	Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using.
Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it.

MongoDB Create Database

How and when to create a database

- If there is no existing database, the following command is used to create a new database.

Syntax:

```
use DATABASE_NAME
```

- If the database already exists, it will return the existing database.

  Let' 's take an example to demonstrate how a database is created in MongoDB. In the following example, we are going to create a database "maintenanceDB".

Few commands for database operation

use DATABASE_NAME	To connect or create database
db	Return current connected/selected database instance
show dbs	Returns the list of databases
db.dropDatabase()	To delete / drop current/selected database

Example of Creating / Selecting a Database :

```
use maintenanceDB
Switched to db maintenanceDB

local 0.078GB
```

- Here, your created database "maintenanceDB" will not be present in the list, insert at least one document into it to display database:

  ```
  db.buildings.insert({"name":"first Building"})
  ```

Example to list all databases on server:

```
Show dbs

-config
-admin
-maintenanceDB
```

Example to Drop a database:

```
db.dropDatabase();
```

Here, db represents connected database instance

MongoDB Create Collection

- In MongoDB, db.createCollection(name, options) is used to create collection. But usually you don't need to create collection.
- MongoDB creates collection automatically when you insert some documents.

Syntax:

```
db.createCollection(name, options)
```

Here,

- **Name**: is a string type, specifies the name of the collection to be created.
- **Options**: is a document type, that specifies the memory size and indexing of the collection. It is an optional parameter.

Following is the list of options that can be used.

Field	Type	Description
Capped	Boolean	(Optional) If it is set to true, enables a capped collection. Capped collection is a fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
AutoIndexID	Boolean	(Optional) If it is set to true, automatically create index on ID field. Its default value is false.
Size	Number	(Optional) It specifies a maximum size in bytes for a capped collection. Ifcapped is true, then you need to specify this field also.
Max	Number	(Optional) It specifies the maximum number of documents allowed in the capped collection.

- Let's take an example to create collection. In this example, we are going to create a collection name requests.

```
use maintenanceDB

db.createCollection("requests")


{ "ok" : 1 }
```

- To check the created collection, use the command "show collections".

```
show collections
```

How does MongoDB create collections automatically

- MongoDB creates collections automatically when you insert some documents.

For example:

- Insert a document named into a collection named request. The operation will create the collection if the collection does not currently exist.

```
db.users.insert({"type" : "admin"})

show collections
```

- If you want to see the inserted document, use the find() command.

Syntax:

```
db.collection_name.find()
```

Drop a Collection in MongoDB:

- If you want to drop a collection in MongoDB use drop() command in following way\

```
db.collection_name.drop()
```

What is CRUD in MongoDB?

CRUD operations describe the conventions of a user-interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:

- The **Create** operation is used to insert new documents in the MongoDB database.
- The **Read** operation is used to query a document in the database.
- The **Update** operation is used to modify existing documents in the database.
- The **Delete** operation is used to remove documents in the database.

Create Operation(Insert)

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()

- As the namesake, insertOne() allows you to insert one document into the collection.

```
db.users.insertOne({name:"adminOne",password:"default"});
```

- If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()

- It's possible to insert multiple items at one time by calling the *insertMany()* method on the desired collection. In this case, we pass multiple items into our chosen collection and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries.

```
db.users.insertMany(
    {name:"userOne",password:"default"},
    {name:"usertwo",password:"default"}
);
```

```
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

- on successful operation, two new documents are created. The function will return an object where “acknowledged” is “true” and “insertIds” are the newly created “ObjectIds”

Read Operation(Select)

- The read operations allow you to supply special query filters and criteria that let you specify which documents you want.
- Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

- In order to get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

- Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.
- If you want to get more specific with a read operation and find a desired subsection of the records, you can use the filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

```
db.RecordsDB.find({"species":"Cat"})
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

findOne()

- In order to get one document that satisfies the search criteria, we can simply use the *findOne()* method on our chosen collection.
- If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk.
- If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection})
```

Let's take the following collection—say, *RecordsDB*, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

And, we run the following line of code:

```
db.RecordsDB.findOne({"age":"8 years"})
```

We would get the following result:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations

- Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.
- You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

- We can update a currently existing record and change a single document with an update operation.
- To do this, we use the *updateOne()* method on a chosen collection, which here is "RecordsDB."

- To update a document, we provide the method with two arguments: an update filter and an update action.
- The update filter defines which items we want to update, and the update action defines how to update those items.
- We first pass in the update filter. Then, we use the "$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne(
        {name: "Marsh"},
        {$set:
            {ownerAddress: "451 W. Coffee St. A204"}
        })
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

updateMany()

- *updateMany()* allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
```

```
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species"
: "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species"
: "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

replaceOne()

- The *replaceOne()* method is used to replace a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species"
: "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

Delete Operations

- Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document.
- You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection.
- The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

- *deleteOne()* is used to remove a document from a specified collection on the MongoDB server.
- A filter criteria is used to specify the item to delete.
- It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
```

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species"
: "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

deleteMany()

- *deleteMany()* is a method used to delete multiple documents from a desired collection with a single delete operation.
- A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
```

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Query Selector & Projection Operator in MongoDB

Comparison

Name	Description	Syntax
`$eq`	Matches values that are equal to a specified value.	{ : { $eq: } }
`$gt`	Matches values that are greater than a specified value.	{ field: { $gt: value } }
`$gte`	Matches values that are greater than or equal to a specified value.	{ field: { $gte: value } }
`$in`	Matches any of the values specified in an array.	{ field: { $in: [, , ...] } }
`$lt`	Matches values that are less than a specified value.	{ field: { $lt: value } }

Example

```
db.contributor.find({branch: {$eq: "CSE"}})
```

Logical

Name	Description	Syntax
`$and`	Joins query clauses with a logical `AND` returns all documents that match the conditions of both clauses.	{ $and: [{ }, { } , ... , { }] }
`$not`	Inverts the effect of a query expression and returns documents that do *not* match the query expression.	Syntax: { field: { $not: { } } }
`$nor`	Joins query clauses with a logical `NOR` returns all documents that fail to match both clauses.	{ $nor: [{ }, { }, ... { }] }
`$or`	Joins query clauses with a logical `OR` returns all documents that match the conditions of either clause.	{ $or: [{ }, { }, ... , { }] }

Example

```
db.contributor.find(
        {
            $and:
                [{branch: "CSE"}, {joiningYear: 2018}]
        }
    )
```

Element Query Operators

Name	Description	Syntax
`$exists`	Matches documents that have the specified field.	{ field: { $exists: } }
`$type`	Selects documents if a field is of the specified type.	{ field: { $type: } }

Example

```
db.spices.find( { saffron: { $exists: true } } )
```

- The results consist of those documents that contain the field `saffron`, including the document whose field `saffron` contains a null value.
-

What is Projection?

MongoDB provides a special feature that is known as **Projection**. It allows you to select only the necessary data rather than selecting whole data from the document. For example, a document contains 5 fields, i.e.,

```
{
name: "Roma",
age: 30,
branch: EEE,
department: "HR",
salary: 20000
}
```

But we only want to display the *name* and the *age* of the employee rather than displaying whole details. Now, here we use projection to display the name and age of the employee.

One can use projection with `db.collection.find()` method. In this method, the second parameter is the projection parameter, which is used to specify which fields are returned in the matching documents.

Projection Operators

Name	Description
`$`	Projects the first element in an array that matches the query condition.
`$elemMatch`	Projects the first element in an array that matches the specified `$elemMatch` condition.
`$slice`	Limits the number of elements projected from an array. Supports skip and limit slices.

Update Operators

Fields

Name	Description
`$currentDate`	Sets the value of a field to current date, either as a Date or a Timestamp.
`$inc`	Increments the value of the field by the specified amount.
`$min`	Only updates the field if the specified value is less than the existing field value.
`$max`	Only updates the field if the specified value is greater than the existing field value.
`$mul`	Multiplies the value of the field by the specified amount.
`$rename`	Renames a field.
`$set`	Sets the value of a field in a document.
`$setOnInsert`	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
`$unset`	Removes the specified field from a document.

Example

Create the `products` collection:

```
db.products.insertOne(   {     _id: 1,     sku: "abc123",     quantity: 10,     metrics:
{ orders: 2, ratings: 3.5 }   })
```

The following updateOne() operation uses the `$inc` operator to:

- increase the `"metrics.orders"` field by 1
- increase the `quantity` field by -2 (which decreases `quantity`)

```
db.products.updateOne(   { sku: "abc123" },   { $inc: { quantity: -2, "metrics.orders": 1 } })
```

The updated document would resemble:

```
{  _id: 1,  sku: 'abc123',  quantity: 8,  metrics: { orders: 3, ratings: 3.5 }}
```

What is Aggragation ?

- Aggregationis a way of processing a large number of documents in a collection by means of passing them through different stages.
- The stages make up what is known as a pipeline. The stages in a pipeline can filter, sort, group, reshape and modify documents that pass through the pipeline.
- One of the most common use cases of Aggregation is to calculate aggregate values for groups of documents.
- This is similar to the basic aggregation available in SQL with the GROUP BY clause and COUNT, SUM and AVG functions.

Aggragation: sort()

Method specifies the order in which the query returns the matching documents from the given collection. You must apply this method to the cursor before retrieving any documents from the database. It takes a document as a parameter that contains a field: value pair that defines the sort order of the result set. The value is 1 or -1 specifying an ascending or descending sort respectively.

- If a sort returns the same result every time we perform on same data, then such type of sort is known as a stable sort.
- If a sort returns a different result every time we perform on same data, then such type of sort is known as unstable sort.
- MongoDB generally performs a stable sort unless sorting on a field that holds duplicate values.
- We can use limit() method with sort() method, it will return first m documents, where m is the given limit.

- MongoDB can find the result of the sort operation using indexes.
- If MongoDB does not find sort order using index scanning, then it uses top-k sort algorithm.

Syntax:

```
db.Collection_Name.sort({field_name:1 or -1})
```

Parameter:

- The parameter contains a field: value pair that defines the sort order of the result set. The value is 1 or -1 that specifies an ascending or descending sort respectively. The type of parameter is a document.

Return:

- It returns the documents in sorted order.

Aggregation: $limit

This aggregation stage limits the number of documents passed to the next stage.

Example

```
db.movies.aggregate([ { $limit: 1 } ])
```

- This will return the 1 movie from the collection.

Aggregation Commands

Name	Description
`aggregate`	Performs aggragation tasks such as `$group` using an aggregation pipeline.
`count`	Counts the number of documents in a collection or a view.
`distinct`	Displays the distinct values found for a specified key in a collection or a view.
`mapReduce`	Performs map-reduce aggregation for large data sets.

Aggregation Methods

Name	Description
`db.collection.aggregate()`	Provides access to the aggragation pipeling
`db.collection.mapReduce()`	Performs map-reduce aggregation for large data sets.

What is the Aggregation Pipeline in MongoDB?

- The aggregation pipeline refers to a specific flow of operations that processes, transforms, and returns results. In a pipeline, successive operations are informed by the previous result.
- Let's take a typical pipeline:

  Input -> `$match` -> `$group` -> `$sort` -> `output`

- In the above example, input refers to one or more documents. `$match`, `$group`, and `$sort` are various stages in a pipeline.
- The output from the `$match` stage is fed into `$group` and then the output from the `$group` stage into `$sort`.
- These three stages collectively can be called an **aggregation pipeline**.
- Implementing a pipeline helps us to **break down queries into easier stages**. Each stage uses namesake operators to complete transformation so that we can achieve our goal.

Arithmetic Expression Operators

Arithmetic expressions perform mathematic operations during aggragation process on numbers. Some arithmetic expressions can also support date arithmetic.

Name	Description
`$abs`	Returns the absolute value of a number.
`$add`	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
`$ceil`	Returns the smallest integer greater than or equal to the specified number.
`$divide`	Returns the result of dividing the first number by the second. Accepts two argument expressions.
`$exp`	Raises *e* to the specified exponent.
`$floor`	Returns the largest integer less than or equal to the specified number.
`$ln`	Calculates the natural log of a number.
`$log`	Calculates the log of a number in the specified base.
`$log10`	Calculates the log base 10 of a number.
`$mod`	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
`$multiply`	Multiplies numbers to return the product. Accepts any number of argument expressions.
`$pow`	Raises a number to the specified exponent.
`$round`	Rounds a number to to a whole integer *or* to a specified decimal place.
`$sqrt`	Calculates the square root.
`$subtract`	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.
`$trunc`	Truncates a number to a whole integer *or* to a specified decimal place.