

Cachalot DB - version 2



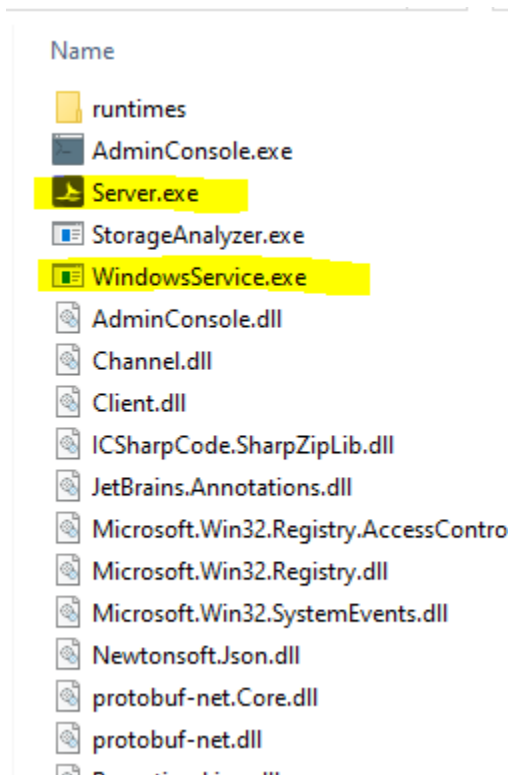
Administration Guide

Table des matières

Introduction	3
CONNECT	6
DESC	7
READONLY / READWRITE	9
DROP	10
TRUNCATE	11
DELETE	12
SELECT	13
IMPORT	14
DUMP	15
RESTORE	16
RECREATE	17
The @ACTIVITY table	18
Understanding the execution plan.....	19

Introduction

When deploying a Cachalot server (usually as a node in a multi-server cluster), copy the content of the “Bin” folder from the distribution package to your installation folder.



Server.exe is the server packaged as a console application used on Windows and Linux.

WindowsService.exe is the same server as a Windows service. We can install it with the **install.cmd** script from the same folder. We need to execute the script with administrative privileges.

Starting **Server.exe** without a command line parameter will read its configuration from the file **node_config.json**. When we specify a parameter, it will be interpreted as a suffix to the name of the configuration file.

For example, “**Server 01**” will use the config file **node_config_01.json**.

A typical configuration file :

```
{  
  "IsPersistent": true,  
  "ClusterName": "test",  
  "TcpPort": 48401,  
  "DataPath": "root/01",  
  "FullTextConfig": {  
    "TokensToIgnore": ["qui", "du", "rue"]  
  }  
}
```

Is Persistent:

When true, it works in database mode, otherwise as a distributed cache.

Cluster Name :

It is used for monitoring only, and it needs to be the same for all the cluster nodes.

TCP port:

The port and needs to be unique on a machine.

Data Path:

It is used only in database mode, and it is the directory where data and logs are stored.

FullTextConfig:

It is an optional list of tokens to ignore to speed up the full-text search. See the `USERGUIDE` for details.

When starting in database mode (IsPersistent=true), the directory “DataPath” is automatically created. Two subdirectories are also created: **logs** and **data**.

The “data” directory contains all the persistent data:

- The append-only transaction log
 - o All changes are **synchronously** appended to this file before being applied to the memory
- The permanent storage
 - o All the entries from the transaction log are **asynchronously** copied to this file by a background thread
- The “schema” file containing the definition of the collections
- The “sequence” file containing the last values for the unique key generators

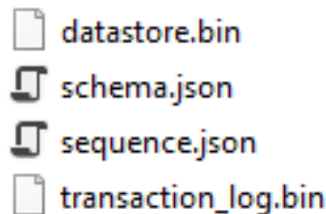


Figure 1The content of the data folder

When the server starts on a non-empty database, it does some cleanup operations before accepting user connections:

- It applies all the pending transactions to the permanent storage
- Then it compactifies the transaction log
- Then it cleans up the datastore by removing deleted and dirty records
 - o Dirty record = a record in the permanent storage that is not used anymore as the object it contained was updated, its size has grown, and it moved to the end of the file

Simply restarting a server can release lots of disk space.

Administrative operations can be done manually using the **AdminConsole** or programmatically using the administration interface.

The admin console has an integrated HELP function and a powerful **autocompletion** feature.

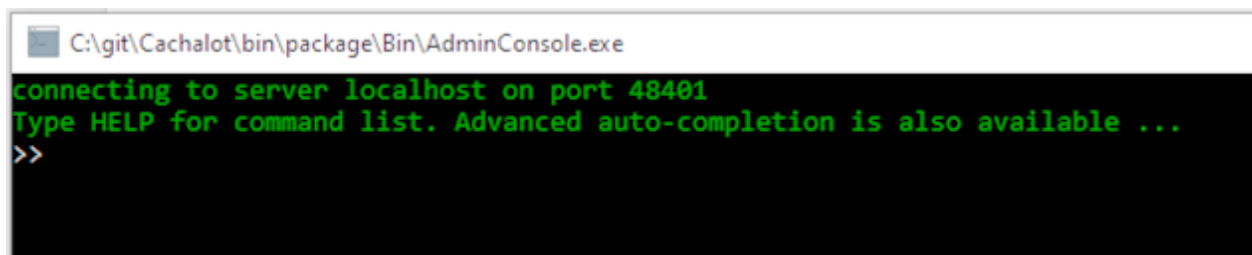
CONNECT

Connect to a single node or a cluster

Syntax:

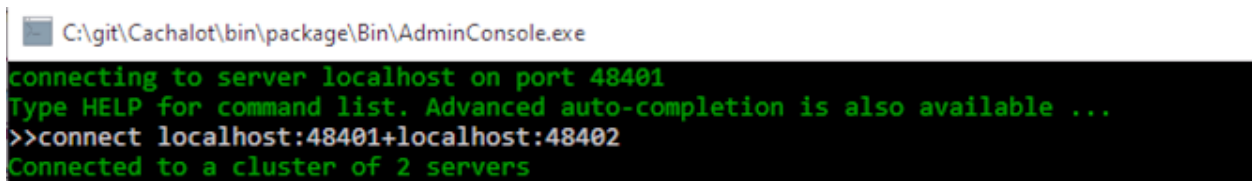
CONNECT <connection string>

When starting the admin console, it will try to connect the default address **localhost:48401**.



```
C:\git\Cachalot\bin\package\Bin\AdminConsole.exe
connecting to server localhost on port 48401
Type HELP for command list. Advanced auto-completion is also available ...
>>
```

To connect to a cluster, use the standard connection string as described in the USERGUIDE: **host1:port1+host2:port2+...**



```
C:\git\Cachalot\bin\package\Bin\AdminConsole.exe
connecting to server localhost on port 48401
Type HELP for command list. Advanced auto-completion is also available ...
>>connect localhost:48401+localhost:48402
Connected to a cluster of 2 servers
```

DESC

Get information about the nodes in the cluster and the collections

Syntax:

DESC [<collection name>]

When used without parameters, it will display information about the cluster servers and list the collections.

```
>>desc
DESC took 5 ms

Server process
-----
image type = 64 bits
started at = 11/25/2021 7:22:22 AM
active clients = 2
threads = 16
physical memory = 589 MB
virtual memory = 2204354 MB
software version = 2.0.8.1

Server process
-----
image type = 64 bits
started at = 11/25/2021 7:22:32 AM
active clients = 1
threads = 16
physical memory = 586 MB
virtual memory = 2204325 MB
software version = 2.0.8.1

Tables
-----
|               Name |   Zip |
|-----|-----|
|          outlets | False |
|         products | False |
|           sales | False |
|       sales_detail | False |
|           Stock | False |
|         @ACTIVITY | False |
|-----|-----|

The call took 5 milliseconds
>>
```

The “Zip” column indicates if the data compression is used for the corresponding collection. More on data compression in the [USERGUIDE](#).

It accepts one parameter, the collection name. This use-case displays information about the server-side values for this collection.

Server-side values may be indexed or not; see [USERGUIDE](#) for details.

```
>>desc products
DESC took 14 ms

PRODUCTS (products)
-----
|               property |   index type |
|-----|
|      ProductId        |     Primary  |
|         Name          |   Dictionary |
|         Brand          |   Dictionary |
|        ScanCode       |   Dictionary |
|         ImageId        |         None  |
|      Categories       |   Dictionary |
|          Tags          |   Dictionary |
|      Ingredients      |   Dictionary |
|-----|
The call took 14 milliseconds
```

Programatically you can do the same with

```
connector.GetClusterDescription()
```


READONLY / READWRITE

Block changes to the database

Syntax:

READONLY

READWRITE

No parameter is required.

The command switches on or off the read-only mode for all the servers in the cluster.

It can be helpful when exporting data or creating a backup in a multi-server cluster to ensure data consistency.

```
>>
>>readonly
>>delete from products
Can not execute GET : Error in RemoveMany(Database is in read-only mode)
DELETE took 1 ms
Deleted 0 items. The call took 1 milliseconds
```

```
>>
>>readwrite
>>delete from products
DELETE took 0 ms
Deleted 10 items. The call took 0 milliseconds
>>
```

Programmatically setting the readonly mode:

```
connector.AdminInterface().ReadOnlyMode(true);
```

And resetting it to read-write

```
connector.AdminInterface().ReadOnlyMode(false);
```

DROP

Reset your database to the original state. Destroys all data and schema information

Syntax:

DROP

No parameter is required.

Completely deletes all data and schema information, and it is the only action for which the admin console asks for a confirmation.

```
>>drop  
This will delete ALL your data. Are you sure (y/n) ?  
y  
>>
```

To drop a database programmatically:

```
connector.AdminInterface().DropDatabase();
```

TRUNCATE

The fastest way to delete the whole content of a collection

Syntax:

TRUNCATE <collection name>

This command deletes all the content of a collection.

The collection will still exist but empty, and schema information is preserved.

```
>>count from products
COUNT took 0 ms
Found 8 items. The call took 0.0000 milliseconds
>>truncate products
DELETE took 0 ms
Deleted 8 items. The call took 0 milliseconds
>>count from products
COUNT took 0 ms
Found 0 items. The call took 0.0000 milliseconds
```

DELETE

Delete all items that match a condition

Syntax:

DELETE FROM <collection name> [WHERE <where clause>]

This command is useful when used with a where clause. Otherwise, it will do the same as TRUNCATE but slower.

SELECT, COUNT, and DELETE use the same syntax for the WHERE clause. The “SQL and LINQ guide” explains it in detail.

```
>>delete from products where brand = REVLON  
DELETE took 9 ms  
Deleted 2 items. The call took 9 milliseconds  
>>
```

SELECT

Visualize or export data

Syntax:

```
SELECT [DISTINCT] [PROPERTY1, PROPERTY2] FROM <collection name>  
[WHERE <where clause>] [ORDER BY <property>]  
[DESCENDING] [TAKE <n>] [INTO <file name>]
```

SELECT, COUNT, and DELETE use the same syntax for the WHERE clause. The “SQL and LINQ guide” explains it in detail.

```
>>select distinct brand from products  
[  
  {  
    "brand": "LOREAL"  
  },  
  {  
    "brand": "NIVEA"  
  },  
  {  
    "brand": "OLAY"  
  },  
  {  
    "brand": "Advanced Clinicals"  
  }  
]
```

If we add INTO at the end, it exports the result into an external JSON file.

```
>>select from outlets into /data/outlets.json  
SELECT took 1 ms
```

IMPORT

Import data from a file

Syntax:

IMPORT <collection name> <file name>

This command imports data from an external JSON file into an **existing** collection.

It adds new items or updates items that are already in the collection (as identified by the primary key).

Data in the file must be a JSON array of objects.

```
>>
>>count from sales
COUNT took 2 ms
Found 100000 items. The call took 2.0000 milliseconds
>>select from sales into /data/sales.json
SELECT took 2513 ms
Found 100000 items. The call took 2513 milliseconds
>>truncate sales
DELETE took 178 ms
Deleted 100000 items. The call took 178 milliseconds
>>count from sales
COUNT took 3 ms
Found 0 items. The call took 3.0000 milliseconds
>>import sales /data/sales.json
Data successfully imported
>>count from sales
COUNT took 1 ms
Found 100000 items. The call took 1.0000 milliseconds
>>
```

The most common use-case is to export data using SELECT INTO, process the data manually or programmatically, and reimport it.

Exported data is pretty printed to facilitate manual edition.

DUMP

Create a full backup of data.

Syntax:

DUMP <directory name>

It is straightforward to use; it creates a subdirectory named “yyyy-mm-dd” with lots of files containing all the data in the database, including schemas and the state of the unique value generators.

If this directory exists, the content is overridden. If you need more than one backup a day, you can have more than one directory: morning_backup, lunch_backup, etc.

There are nonetheless some essential points to understand:

The backup and restore system is designed to be as fast as possible.

All servers in the cluster back up their data in parallel. **The dump directory must be visible by all servers**, not necessarily the admin console.

So, unless you want a local backup of a local server, **the backup directory must be a network share**.

RESTORE

Restore the database from a backup

Syntax:

RESTORE <directory name>[/yyy-mm-dd]

You can either specify only the root directory, and in this case, the last backup will be restored, or choose a specific date.

All data in the database is lost—no need to drop the database before.

When you restore a cluster, the number of servers must be the same as when you created the backup. To restore data on a different cluster configuration, use the RECREATE command

RECREATE

Initialize a cluster from a dump if the configuration changed

Syntax:

RECREATE <directory name>[/yyy-mm-dd]

The syntax is similar to RESTORE, but the behavior is quite different.

When you restore from a backup, each server in the cluster restores its data. This operation works only if the cluster configuration does not change: **the same number and order of servers in the connection string.**

When you recreate a cluster from backup, all data is read by the admin console and fed to the cluster. The operation is longer than a restore.

If you are reusing a node, all data should be cleaned. Use DROP before RECREATE.

For example, you add a new server to a single server cluster

CONNECT server1:port1

DUMP \\FILESERVER \dump

CONNECT server1:port1 + server2:port2

DROP

RECREATE \\FILESERVER \dump

The @ACTIVITY table

@ACTIVITY is a unique table that exists even in an empty database. It is non-persistent, limited to 20 000 entries. It contains details of all the commands processed by the server. Every entry has a timestamp and includes the server-side execution time.

It allows querying server activities like any other collection of data.

It is like a structured and indexed log.

We can use all the SQL syntax, and for most common queries, simpler aliases are defined.

These are the server-side values that we can use for queries on this collection:

```
>>>desc @ACTIVITY
DESC took 23 ms

@ACTIVITY (@ACTIVITY)
-----
|                property |    index type |
|-----|-----|
|              Id        |    Primary    |
|            TimeStamp   |    Ordered    |
|              Type      |    Dictionary  |
|          CollectionName |    Dictionary  |
| ExecutionTimeInMicroseconds |    Ordered    |
|              Detail    |     None      |
|              Query     |     None      |
|-----|-----|

The call took 23 milliseconds
>>
```

For example, to find the **last n commands**, we can write the query:

```
select from @activity order by TimeStamp descending take <n>
```

As we use this query quite often, a simple alias is defined:

```
last <n>
```

Another use case for which we defined an alias: **the longest queries**.

We can find them with SQL:

```
select from @ACTIVITY where type=SELECT order by  
ExecutionTimeInMicroseconds descending take <n>
```

Or with the alias:

```
longest <n>
```

Understanding the execution plan

The entries of the @ACTIVITY collection that represent queries also explain the execution plan of the **query manager**, an essential server component.

- How it divides complex queries into simpler ones
 - o Which indexes (if any) it uses for every simple query
 - o How much time was spent choosing the indexes
 - o How much time was spent using the indexes
 - o If indexes were not used for all the conditions in the query, it spends some time checking for non-indexable ones
- How the query manager merges the results of simple queries
 - o How much time for ORDER BY
 - o How much time to apply the DISTINCT operator

Example 1 - simple query using the OR operator

For this query

select from products where Brand = DOVE or Categories contains soap

the execution plan will look like this:

```
"ExecutionPlan": {
  "$type": "Client.Core.ExecutionPlan, Client",
  "TotalTimeInMicroseconds": 8044,
  "QueryPlans": [
    {
      "$type": "Client.Core.QueryExecutionPlan, Client",
      "Query": "Brand = DOVE",
      "SimpleQueryStrategy": true,
      "IndexTimeInMicroseconds": 11,
      "UsedIndexes": [
        "Brand"
      ]
    },
    {
      "$type": "Client.Core.QueryExecutionPlan, Client",
      "Query": "Categories Contains soap",
      "SimpleQueryStrategy": true,
      "IndexTimeInMicroseconds": 10,
      "UsedIndexes": [
        "Categories"
      ]
    }
  ],
  "MergeTimeInMicroseconds": 80
}
```

The queries containing an OR operator are split into simpler ones executed in parallel. In the end, we merge the results of simple queries on each server in the cluster, then re-merge on the client-side. The process is, of course, transparent for the client code.

In this case, we notice that the query was split into two simple parts: they are explained by the **QueryPlans** list.

SimpleQueryStrategy: true means only one index can be used, and the query manager may skip the index choice step.

Example 2 – query with AND operator and multiple clauses

For this query

*select from sales_detail where IsDelivered = True AND Amount > 80 and
Quantity = 1*

the execution plan will be:

```
"ExecutionPlan": {
  "$type": "Client.Core.ExecutionPlan, Client",
  "TotalTimeInMicroseconds": 19466,
  "QueryPlans": [
    {
      "$type": "Client.Core.QueryExecutionPlan, Client",
      "Query": "IsDelivered = True AND Amount > 80 AND Quantity = 2",
      "PlanningTimeInMicroseconds": 27,
      "IndexTimeInMicroseconds": 15667,
      "ScanTimeInMicroseconds": 3755,
      "UsedIndexes": [
        "IsDelivered",
        "Amount"
      ]
    }
  ]
}
```

PlanningTimeInMicroseconds: The query manager spent 27 **microseconds** choosing an **ordered** list of indexes to use.

IndexTimeInMicroseconds: time spent searching the two indexes and computing the intersection of the two result sets.

ScanTimeInMicroseconds: The **Quantity** property is not indexed, so the query manager needs to check for this condition *Quantity = 1* on the result set produced by the indexes.

Example 3- using the execution plan to debug a real-life performance issue

Looking for the queries that take the longest time to execute on a real system (using “longest” command alias), we found this one:

select from sales_detail where date > 2020-01-01 and date <= 2020-01-15

The execution plan is :

```
"ExecutionPlan": {
  "$type": "Client.Core.ExecutionPlan, Client",
  "TotalTimeInMicroseconds": 32853,
  "QueryPlans": [
    {
      "$type": "Client.Core.QueryExecutionPlan, Client",
      "Query": "Date in range (1/1/2020 12:00:00 AM +01:00, 1/15/2020 12:00:00 AM +01:00]",
      "SimpleQueryStrategy": true,
      "FullScan": true,
      "ScanTimeInMicroseconds": 32822
    }
  ]
}
```

The first thing we notice is that the double comparison gets optimized as a range operator, which usually dramatically improves index performance.

But the query is still executed as a full scan and has no index usage at all.

The explanation is quite simple, finally.

```
>>desc sales_detail
```

```
DESC took 7 ms
```

```
SALES_DETAIL (sales_detail)
```

property	index type
Id	Primary
SaleId	Dictionary
Amount	Ordered
Quantity	None
ProductId	Dictionary
ClientId	Dictionary
Date	Dictionary
DayOfWeek	Dictionary
Month	Dictionary
Year	Dictionary
IsDelivered	Dictionary
Channel	Dictionary

```
The call took 7 milliseconds
```

“Date” is indexed as a dictionary, not an ordered index, so the comparison operators can not use it.

We changed the index type to “Ordered,” which significantly improved the execution time (server-side, more than ten-fold).

Changing the schema definition in the client code automatically reindexes the data.
