

Cachalot DB - version 1.1.3



Quick Start and Administration Guide

Index

What is new in this version	4
What is Cachalot DB?	4
Show me some code first	4
Other types of indexes and how to use them	7
1) Index Data Type.....	7
2) Index type.....	7
3) Ordered index.....	7
More code. Adding indexes to Home class	8
Indexing collection properties.....	10
Full-text search.....	11
Fine-tuning the full text search	12
Other methods of the API	13
Deleting items from the database.....	13
Inserting or updating many objects	13
Compressing object data.....	14
Storing polymorphic collections in the database	14
Conditional operations and “optimistic synchronization”	16
Two Stage Transactions.....	18
More on client configuration	22
The connection-pool	22
In-process server	24
Using Cachalot as a distributed cache with unique features	25
Serving single objects from a cache	25
Serving complex queries from a cache.....	27
First case: all data in the database is loaded into the cache	27
Second case: a subset of the database is loaded into the cache	29
What is Cachalot DB good at?	31
Administration.....	32
Connecting to the database	33
Visualizing data.....	34
Updating data.....	36
Backup and Restore.....	37
Change cluster configuration	39
Other commands.....	39

What is new in this version

Full-text search support. See the corresponding section [here](#).

The admin console was also ported to netcore. It can now be used on Linux and mac OSX

What is Cachalot DB?

A very fast, open source, no-sql, fully transactional database for .NET applications.

It is distributed, it scales linearly with the number of nodes. On a single node you can durably insert fifty thousand objects per second on a modest system.

A powerful LINQ provider is available. As well as an administration console.

It can also be used as a very powerful, transactional, distributed cache with unique features.

Much more detail in the next sections but...

Show me some code first

Let's prepare our business objects for database storage.

We start with a toy web site which allows to rent homes between individuals.

A simple description of a real estate property would be.

```
public class Home
{
    public string CountryCode { get; set; }
    public string Town { get; set; }
    public string Adress { get; set; }
    public string Owner { get; set; }
    public string OwnerEmail { get; set; }
    public string OwnerPhone { get; set; }
    public int Rooms { get; set; }
    public int Bathrooms { get; set; }
    public int PriceInEuros { get; set; }
}
```

The first requirement for a business object is to have a primary key. As there is no “natural” one in this case, we will add a numeric Id.

```
public class Home
{
    [PrimaryKey(KeyDataType.IntKey)]
    public int Id { get; set; }

    ...
}
```

Now the object can be stored in the database.

First step is to instantiate a **Connector** which needs a client configuration. More on the configuration later but, for now, it needs to contain the list of servers in the cluster. To start, only one run locally.

The configuration is usually read from an external file. For the moment, let's build it manually

```
var config = new ClientConfig
{
    Servers = {new ServerConfig {Host = "localhost", Port = 4848}}
};

using (var connector = new Cachalot.Linq.Connector(config))
{
    var homes = connector.DataSource<Home>();
    // the rest of the code goes here
}
```

One last step before storing an object in the database. We need to generate a unique value for the primary key. Multiple unique values can be generated with a single call.

Unlike other databases, you do not need to explicitly create a unique value generator. First call with an unknown generator name will automatically create it.

```
var ids = connector.GenerateUniqueIds("home_id", 1);

var home = new Home
{
    Id = ids[0],
    Adress = "14 rue du chien qui fume",
    Bathrooms = 1,
    CountryCode = "FR",
    PriceInEuros = 125,
    Rooms = 2,
    Town = "Paris"
};

homes.Put(home);
```

Now your first object is safely stored in the database.

For the moment, you can only retrieve it by primary key. That can be done in two equivalent ways.

```
var reloaded = homes[property.Id];
```

Or with a LINQ expression.

```
var reloaded = homes.First(p => p.Id == property.Id);
```

The first one is faster as there is no need to parse the expression tree.

In most relational databases we use two distinct operations: INSERT and UPDATE. In Cachalot Db only one operation is exposed: PUT

It will insert new items (new primary key) and will update existing items.

You probably have higher expectation from a modern database than simply storing and retrieving objects by primary key. And you are right.

Other types of indexes and how to use them

Three characteristics of an index need to be understood

1) Index Data Type

To be indexable in Cachalot DB, a .NET property needs to be convertible either to Int64 or string.

Using integer type makes the search slightly faster.

Automatic conversion to Int64 is provided for

- All numeric types
- DateTime and DateTimeOffset
- Enumerated types
- Boolean

Conversion to string is done by calling **ToString()** on the property value

2) Index type

Three types of indexed properties are available

- Primary key (the only mandatory one)
- Unique key: zero or more can be defined on type
- Index key: zero or more can be defined on type

3) Ordered index

On any index we can apply the equality operator.

If an index is declared as “ordered”, all the comparisons operators can equally be used: <, <=, >, >=

This type of index is essential for most modern systems but be aware that it has a cost because the ordered indexes must always be sorted.

Massive insert/update operations (**DataStore.PutMany** method) are well optimized. After a threshold is reached (50 items by default) the operation is treated like a “bulk insert”. Ordered indexes are sorted only once, at the end.

More code. Adding indexes to Home class

```
public class Home
{
    [PrimaryKey(KeyDataType.IntKey)]
    public int Id { get; set; }

    [Index(KeyDataType.StringKey)]
    public string CountryCode { get; set; }

    [Index(KeyDataType.StringKey)]
    public string Town { get; set; }
    public string Adress { get; set; }
    public string Owner { get; set; }
    public string OwnerEmail { get; set; }
    public string OwnerPhone { get; set; }

    [Index(KeyDataType.IntKey, ordered:true)]
    public int Rooms { get; set; }

    [Index(KeyDataType.IntKey)]
    public int Bathrooms { get; set; }

    [Index(KeyDataType.IntKey, ordered:true)]
    public decimal PriceInEuros { get; set; }
}
```


With these new indexes you can now do some useful queries

```
var results = homes.Where(  
p => p.PriceInEuros <= 200 && p.Rooms > 1 && p.Town == "Paris").Take(10);
```

The query is, of course, executed server-side including the **take** operator. At most ten objects are sent to the client through the network

The “Contains” extension method is also supported

```
var towns = new[] { "Paris", "Nice" };  
var one = homes.First(p => p.PriceInEuros < 150 && towns.Contains(p.Town));
```

This is equivalent to the SQL:

```
SELECT * from HOME where PriceInEuros < 150 and Town IN ("Paris", "Nice")
```

Another use of the “Contains” extension, which does not have any equivalent in traditional SQL, is explained in the next section.

Indexing collection properties

Let's enrich our business object. It would be useful to have the list of available dates on each home.

Adding this new property enables some interesting features.

This is a collection property and it can be indexed the same way as the scalar properties

```
[Index(KeyDataType.IntKey)]  
public List<DateTime> AvailableDates { get; set; } = new List<DateTime>();
```

We would like to be able search for homes available at a specific date

```
var availableNextWeek = homes.Where(  
    p => p.Town == "Paris" &&  
    p.AvailableDates.Contains(DateTime.Today.AddDays(7))  
).ToList();
```

This has no direct equivalent in the classical SQL databases. It conveniently replaces most of the uses for the classical JOIN operator.

You may need to dynamically create a query. For example, in a search screen, you add criteria to restrict you results. This can be done by chaining **Where** methods.

```
var query = homes.Where(p => p.Town == "Paris");  
query = query.Where(p => p.PriceInEuros < 200);  
query = query.Where(p => p.Rooms > 1);  
query = query.Where(p => p.AvailableDates.Contains(DateTime.Today));  
var result = query.ToList();
```

This is equivalent to a single query where all criteria are joined with the && (AND) operator.

Full-text search

Starting version 1.1.3 a very efficient and customizable full-text indexation is provided.

First you need to prepare the business objects for full-text indexation. This is done in the usual way with a specific tag. Let's index as full text the address, the town and the comments.

```
public class Home
{
    [PrimaryKey(KeyDataType.IntKey)]
    public int Id { get; set; }

    [Index(KeyDataType.StringKey)]
    public string CountryCode { get; set; }

    [FullTextIndexation]
    [Index(KeyDataType.StringKey)]
    public string Town { get; set; }

    [FullTextIndexation]
    public string Adress { get; set; }

    ...

    [Index(KeyDataType.IntKey, ordered:true)]
    public decimal PriceInEuros { get; set; }

    [Index(KeyDataType.IntKey)]
    public List<DateTime> AvailableDates { get; set; } = new List<DateTime>();

    [FullTextIndexation]
    public List<Comment> Comments { get; set; } = new List<Comment>();
}
```

You notice that full-text indexation can be applied both to normally indexed properties (Town is both indexed for LINQ queries and for full text search) and to properties that are not available to LINQ queries. It can be applied to scalar and collection properties.

A new LINQ extension method is provided: **FullTextSearch**. It is accessible through the Data Source class.

It can be used alone or mixed with usual predicates. In the second case the result will be the intersection of the sets returned by the LINQ query and by the full-text query.

In both cases the order is given by the full-text query (most pertinent items first).

```
var result = homes.FullTextSearch("Paris close metro").ToList();

var inParisAvailableTomorrow = homes
    .Where(p => p.Town == "Paris" && p.AvailableDates.Contains(DateTime.Today.AddDays(1)))
    .FullTextSearch("close metro")
    .ToList
```

Fine-tuning the full text search

In any language there are words that have no meaning by themselves but are useful to build sentences. For example in English: to, the, that, at, a... They are called “stop words” and are usually the most frequent words in a language.

The speed of the full text search is greatly improved if we do not index them. The configuration file “node_config.json” allows to specify them. This part should be identical for all nodes in a cluster.

```
{
  "IsPersistent": true,
  "ClusterName": "test",
  "TcpPort": 4848,
  "DataPath": "root/4848",
  "FullTextConfig": {
    "TokensToIgnore": ["qui", "du", "rue"]
  }
}
```

When a node starts, it generates in the “DataPath” folder a text file containing the 100 most frequent words: **most_frequent_tokens.txt**. These are good candidates to ignore. You may need to ignore other words depending on your business case. For example if you are indexing addresses “road” or “avenue” should be ignored.

Other methods of the API

In addition to querying and putting single items, other methods are exposed by the **DataSource** class

Deleting items from the database

```
home.Delete(home);
```

```
homes.DeleteMany(p => p.Town == "Paris");
```

Inserting or updating many objects

```
homes.PutMany(collection);
```

This method is very optimized for huge collections of objects

- Packets of objects are sent together in the network
- For massive collections, the ordered indexes are sorted only after the insertion of the last object. It is like a BULK INSERT in classical SQL databases

The parameter is an **IEnumerable**. This allows to dynamically generate data that will be inserted in the database. The full collection does not need to be present in client memory.

Compressing object data

The business objects are stored internally in a type-agnostic format. Index fields are stored as **Int64** or **string**, and all the object data is stored as UTF-8 encoded JSON. The process of transforming a .NET object in the internal format is called “packing”. Packing is done client-side, the server only uses the indexes and manipulates the object as row data. It has no dependency on the concrete .NET datatype.

In our example, adding availability dates to our business object has an impact on its size. By default, the object data is not compressed but for objects that take more than a few kilobytes, compression may be very useful. For an object that takes 10 KB in JSON, compression ratio is around 1:10.

To enable compression, add a single attribute on the business data type.

```
[Storage(compressed:true)]  
  
public class Home  
{  
    ...  
}
```

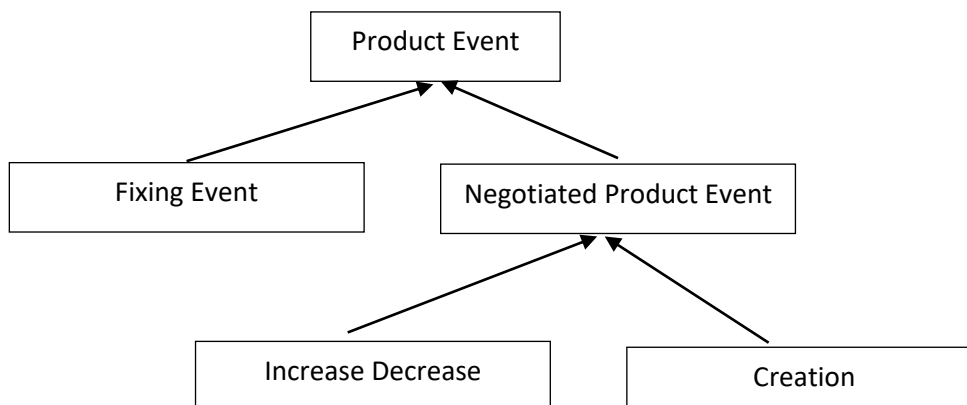
Using compressed objects is transparent for the client code. However, it has an impact on the packing time which is done on the client. When objects are retrieved they are unpacked (which may imply decompression)

As a conclusion, compression may be very useful starting with medium size objects if you are ready to pay a small price, client-side only, for data insertion and retrieval.

Storing polymorphic collections in the database

Polymorphic collections are natively managed. Type information is stored internally in the JSON and it is used to deserialize the proper concrete type.

A small example from a trading system



To store a polymorphic collection, we must expose all required indexes on the base type.

If this is not natural for your business-object hierarchy, an alternate solution is possible. Create a new data type that aggregates a base type and exposes all the required index fields.

Null values are perfectly acceptable for index fields which allows to expose indexed properties which make sense only for a specific child type.

Code example for the first solution:

```
public abstract class ProductEvent
{
    [PrimaryKey(KeyDataType.IntKey)]
    public int Id { get; set; }

    [Index(KeyDataType.StringKey)]
    public abstract string EventType { get; }

    [Index(KeyDataType.IntKey, ordered:true)]
    public DateTime EventDate { get; set; }

    [Index(KeyDataType.IntKey, ordered: true)]
    public DateTime ValueDate { get; set; }

    ...
}

public abstract class NegotiatedProductEvent: ProductEvent
{
    ...
}

public class IncreaseDecrease : NegotiatedProductEvent
{
    ...

    public override string EventType => "IncreaseDecrease";
}
```

This is an example of code which retrieves a collection of concrete events from a **Data Source** typed with an abstract base class.

```
var events = connector.DataSource<ProductEvent>();  
var increaseEvents = events.Where(  
    evt => evt.EventType == "IncreaseDecrease" &&  
    evt.EventDate == DateTime.Today  
) .Cast<IncreaseDecrease>();
```

Conditional operations and “optimistic synchronization”

A normal “put” operation adds an object or updates an existent one using the primary key as object identity.

More advanced use cases are implemented:

- 1) **Add an object only if it is not already there and tell me if it was really added**
- 2) **Update an existent object only if the current version in the database satisfies a condition**

The first one is available through the **TryAdd** operation on the **Data Source** object. If the object was already there it is not modified, and it returns false. The test on the object existence and the insertion are executed as an atomic operation. The object cannot be updated or deleted by another client in-between.

That can be useful for data initialization, creating singleton objects, distributed locks etc.

The second use case is especially useful for, but not limited to, the implementation of “optimistic synchronization”.

If we need to be sure that nobody else modified an object while we were editing it (manually or algorithmically), there are two possibilities

- Lock the object during the edit operation. This is not the best option for a modern distributed system. A distributed lock is not suitable for massively parallel processing and if it is not released automatically (due to client or network failure) manual intervention by an administrator is required
- Use “optimistic synchronization” also known as “optimistic lock”: do not lock but require that, when saving the modified object, the one in the database did not change since it was loaded. Otherwise the operation fails, and we must retry (load + edit + save).

This can be achieved in different ways:

- Having a version on an object. When we save version n+1 we require that the object in the database is still at version n. In Cachalot DB the syntax is `datastore.UpdateIf(item, i=> i.Version == n-1)`
- Having a timestamp on an object. When we save a modified object, we require that the timestamp of the version in the database is identical to the one of the object before our update.

```
var oldTimestamp = item.Timestamp;  
item.Timestamp = DateTime.Now;  
datastore.UpdateIf(item, I => i.Timestamp == oldTimestamp);
```

This can be even more useful when committing multiple object modifications in a transaction. If a condition is not satisfied on one object, rollback the whole transaction. See next section...

Two Stage Transactions

The most important thing to understand about two stage transactions is when you really need them. Most of the time you don't.

An operation that involves one single object (Put, TryAdd, UpdateIf, Delete) is always transactional.

It is durable (operations are synchronously written to an append-only transaction log), and it is atomic. An object will be visible to the rest of the world only fully updated or fully inserted.

On a single-node cluster, operations on multiple objects (PutMany, DeleteMany) are also transactional.

You need two stage transactions only if you must transactionally manipulate multiple objects on a multi-node cluster.

As usual, let's build a small example: a toy banking system that allows money to be transferred between accounts. There are two types of business objects: **Account** and **AccountOperation**

```
public class Account
{
    [PrimaryKey(KeyDataType.IntKey)]
    public int Id { get; set; }

    [Index(KeyDataType.IntKey, true)]
    public decimal Balance { get; set; }
}

public class AccountOperation
{
    [PrimaryKey(KeyDataType.IntKey)]
    public int Id { get; set; }

    [Index(KeyDataType.IntKey)]
    public int SourceAccount { get; set; }

    [Index(KeyDataType.IntKey)]
    public int TargetAccount { get; set; }

    [Index(KeyDataType.IntKey, ordered:true)]
    public DateTime Timestamp { get; set; }

    public decimal TransferredAmount { get; set; }
}
```

Let's create two accounts. No need for transactions at this stage.

```
var accountIds = connector.GenerateUniqueIds("account_id", 2);
var accounts = connector.DataSource<Account>();

var account1 = new Account {Id = accountIds[0], Balance = 100};
var account2 = new Account {Id = accountIds[1], Balance = 100};

accounts.Put(account1);
accounts.Put(account2);
```

When we transfer money between the accounts we would like to simultaneously (atomically) update the balance of both accounts and to create a new instance of **AccountOperation**.

This is how the business logic could be implemented

```
private static void MoneyTransfer(Connector connector, Account sourceAccount,
Account targetAccount, decimal amount)
{
    sourceAccount.Balance -= amount;
    targetAccount.Balance += amount;

    var tids = connector.GenerateUniqueIds("transaction_id", 1);
    var transfer = new AccountOperation
    {
        Id = tids[0],
        SourceAccount = sourceAccount.Id,
        TargetAccount = targetAccount.Id,
        TransferredAmount = amount
    };
    var transaction = connector.BeginTransaction();
    transaction.Put(sourceAccount);
    transaction.Put(targetAccount);
    transaction.Put(transfer);
    // this is where the two stage transaction happens
    transaction.Commit();
}
```

The operations allowed inside a transaction are:

- Put
- Delete
- UpdateIf

If a conditional update (**UpdateIf**) is used and the condition is not satisfied by one object, the whole transaction is rolled back.

In the previous example we could, to prevent “monetary creation”, allow the transaction to be committed only if the balance of the source account did not change since the account object was loaded.

```
private static void MoneyTransfer(Connector connector, Account sourceAccount,
Account targetAccount, decimal amount)
{
// store the old balance before updating it
    var oldBalance = sourceAccount.Balance;
    sourceAccount.Balance -= amount;
    targetAccount.Balance += amount;

    var tids = connector.GenerateUniqueIds("transaction_id", 1);
    var transfer = new AccountOperation
    {
        Id = tids[0],
        SourceAccount = sourceAccount.Id,
        TargetAccount = targetAccount.Id,
        TransferredAmount = amount
    };
    var transaction = connector.BeginTransaction();
// if another operation changed the balance, the transaction is
// rolled-back and an exception is thrown
    transaction.PutIf(sourceAccount, acc=>acc.Balance == oldBalance);
    transaction.Put(targetAccount);
    transaction.Put(transfer);
    transaction.Commit();
}
```

A full example-application be found in the release package: “DemoClients/Accounts”.

More on client configuration

At this point it is important to stress a significant design choice:

The nodes of a cluster do not know each other. Only the client has a view of the whole cluster.

In the previous sections we mentioned the **ClientConfig** class which is the one and only parameter required to instantiate a **Connector**.

You already know it contains the list of servers in the cluster and in the previous example we filled it manually. It can do much more and usually it is loaded from an XML file.

```
var cfg = new ClientConfig();  
cfg.LoadFromFile("two_nodes_cluster.xml");
```

In all the previous examples the index fields were described by using attributes on .NET properties.

Alternatively, indexes can be described in the configuration file. Both ways are perfectly acceptable, but you need to choose one of them. If you use tagged properties on your objects they will override the parameters from the configuration file.

On the next page, an example of configuration file that contains an equivalent description for the **Home** type.

The connection-pool

Cachalot DB is usually used in server environments (REST backends for example).

Lots of clients can make requests in the same time. Without a connection pool, their requests would be either queued or a new connection would be required for each client.

A high-performance connection pool is used transparently by the client code. It allows for simultaneous connections from multiple clients but ensures that a reasonable quantity of TCP connections is used.

Two parameters of the connection pool can be set in the configuration file

Capacity: the maximum number of connections in the pool; it is also the maximum number of client requests that are processed in parallel. If this limit is reached, requests are queued. Given the average request time which is in the low milliseconds (mostly network latency) the number of clients that can effectively be connected to the backend without any visible latency is much more than the capacity of the connection pool. For example, if you have one thousand connected clients, ten simultaneous connections to the database are probably enough. It depends, of course, of the kind of request and on their frequency.

Preloaded: by default, the pool is empty, and it is dynamically filled to scale up with the client activity. This parameter allows connections to be preloaded in the pool thus improving the response time for the very first requests.

The connection pool allows also for the transparent recreation of connections if one or more nodes restart. The client does not need to reconnect explicitly.

```
<?xml version="1.0" encoding="utf-8" ?>
<clientConfig>
  <servers>
    <server>
      <host>SRVPRD100</host>
      <port>4848</port>
    </server>
    <server>
      <host>SRVPRD200</host>
      <port>4568</port>
    </server>
  </servers>
  <typeDescriptions>
    <type fullName="BookingMarketplace.Home" assembly="BookingMarketplace">
      <property name="Id" dataType="int" keyType="primary" ordered="false"/>
      <property name="CountryCode" dataType="string" keyType="index" ordered="false"/>
      <property name="Town" dataType="string" keyType="index" ordered="false"/>
      <property name="Rooms" dataType="int" keyType="index" ordered="true"/>
      <property name="Bathrooms" dataType="int" keyType="index" ordered="true"/>
      <property name="ValueDate" dataType="int" keyType="index" ordered="true"/>
      <property name="PriceInEuros" dataType="int" keyType="index" ordered="true"/>
      <property name="AvailableDates" dataType="int" keyType="list" ordered="false"/>
    </type>
  </typeDescriptions>
  <connectionPool capacity="10" preloaded="2"/>
</clientConfig>
```

In-process server

In some cases, especially if the quantity of data is bounded and it can be stored on a single node, you can instantiate a Cachalot server directly inside your server process. This will give blazing fast responses as there is no more network latency involved.

To do this, pass an empty client configuration to the **Connector** constructor. A database server will be instantiated inside the connector object and communications will be done by simple in-process calls, not a TCP channel.

```
var connector = new Connector(new ClientConfig());
```

Connector implements **IDisposable**. Disposing the **Connector** will gracefully stop the server. You need to instantiate the Connector once when the server process starts and dispose it once when the server process stops.

Using Cachalot as a distributed cache with unique features

Serving single objects from a cache

The most frequent use-case for a distributed cache is to store objects identified by one or more unique keys.

A database contains the persistent data and, when an object is accessed, we first try to get it from the cache and, if not available, load it from the database. Most usually if the object is loaded from the database it is also stored in the cache for later use.

```
Item = cache.TryGet(itemKey)
If Item found
    return Item
Else
    Item = database.Load(itemKey)
    cache.Put(Item)
    return Item
```

By using this simple algorithm, the cache is progressively filled with data and its “hit ratio” improves over time.

This cache usage is usually associated with an “eviction policy” to avoid excessive memory consumption. When a threshold is reached (either in terms of memory usage or object count) some of the objects from the cache are removed.

The most frequently used eviction policy is “Least Recently Used” abbreviated **LRU**. In this case, every time an object is accessed in the cache, its associated timestamp is updated. When eviction is triggered, we remove the objects with the oldest timestamp.

Using cachalot as a distributed cache of this type is very easy.

First disable persistence (by default it is enabled). On every node in the cluster there is a small configuration file called **node_config.json**. It usually looks like this

```
{
  "IsPersistent": true,
  "ClusterName": "test",
  "TcpPort": 6666,
  "DataPath": "root"
}
```

To switch a cluster to pure cache mode, simply set **IsPersistent** to false on all the nodes. **DataPath** will be ignored in this case

Example of client code with LRU eviction activated

```
public class TradeProvider
{
    private Connector _connector;
    public void Startup(ClientConfig config)
    {
        _connector = new Connector(config);
        var trades = _connector.DataSource<Trade>();
        // remove 500 items every time the limit of 500_000 is reached
        trades.ConfigEviction(EvictionType.LessRecentlyUsed, 500_000, 500);
    }
    public Trade GetTrade(int id)
    {
        var trades = _connector.DataSource<Trade>();
        var fromCache = trades[id];
        if (fromCache != null)
        {
            return fromCache;
        }
        var trade = GetTradeFromDatabase(id);
        trades.Put(trade);
        return trade;
    }
    public void Shutdown()
    {
        _connector.Dispose();
    }
}
```

Eviction is configured by data type. Each data type can have a specific eviction policy (or none).

Every decent distributed cache on the market can do this. But Cachalot can do much more.

Serving complex queries from a cache

The single-object access mode is useful in some real-world cases like storing session information for web sites, partially filled forms, blog articles and much more.

But sometimes we need to retrieve a collection of objects from a cache with a SQL-like query.

And we would like the cache to return a result only if it can guarantee that all the data concerned by the query is available in the cache.

The obvious issue here is: **How do we know if all data is available in the cache?**

First case: all data in the database is loaded into the cache

In the simplest (but not the most frequent) case, we can guarantee that all data in the database is also in the cache. This requires that RAM is available for all the data in the database.

The cache is either preloaded by an external component (for example each morning) or it is lazily loaded when we first access it.

Pseudocode for lazy loading all data for a data-type in the cache.

```
Items = cache.GetIfAllAvailable(query)
If Items available
    return Items
Else
    Items = database.Load(query)
    cache.Put(Items)
    return Items
```

Two new methods are available on the **DataSource** class to manage this use-case.

- 1) A LINQ extension: **OnlyIfComplete**. When we insert this method in a LINQ command pipeline it will modify the behavior of the data source. It returns an **IEnumerable** only if all data is available and it throws an exception otherwise.
- 2) A new method to declare that all data is available for a given data type: **DeclareFullyLoaded** (member of **DataSource** class)

Here is a code example extracted from a unit test

```
var dataSource = connector.DataSource<ProductEvent>();
dataSource.PutMany(events);

// here an exception will be thrown
Assert.Throws<CacheException>(() =>
    dataSource.Where(e => e.EventType == "FIXING").OnlyIfComplete().ToList()
);

// declare that all data is available
dataSource.DeclareFullyLoaded();

// here it works fine
var fixings = dataSource.Where(e => e.EventType == "FIXING").OnlyIfComplete().ToList();
Assert.Greater(fixings.Count, 0);

// declare that data is not available again
dataSource.DeclareFullyLoaded(false);

// an exception will be thrown again
Assert.Throws<CacheException>(() =>
    dataSource.Where(e => e.EventType == "FIXING").OnlyIfComplete().ToList()
);
```

Second case: a subset of the database is loaded into the cache

For this use-case Cachalot provides an inventive solution:

- Describe preloaded data as a query (expressed as LINQ expression)
- When data is queried from the cache, determine if the query is a subset of the preloaded data

The two methods (of class **DataSource**) involved in this process are:

- The same **OnlyIfComplete** LINQ extension
- **DeclareLoadedDomain** method. Its parameter is a LINQ expression that defines a subdomain of the global data

Some examples

- 1) In the case of a renting site like Airbnb we would like to store in cache all properties in the most visited cities.

```
homes.DeclareLoadedDomain(h=>h.Town == "Paris" || h.Town == "Nice");
```

Then this query will succeed as it is a subset of the specified domain

```
var result = homes.Where( h => h.Town == "Paris" && h.Rooms >= 2)  
    .OnlyIfComplete().ToList();
```

But this one will throw an exception

```
result = homes.Where(h => h.CountryCode == "FR" && h.Rooms == 2)  
    .OnlyIfComplete().ToList()
```

If we omit the call to **OnlyIfComplete** it will simply return the elements in the cache that match the query

- 2) In a trading system we want to cache all the trades that are alive (maturity date \geq today) and all the ones that have been created in the last year (trade date $>$ one year ago)

```
var oneYearAgo = DateTime.Today.AddYears(-1);
var today = DateTime.Today;

trades.DeclareLoadedDomain(
    t=>t.MaturityDate >= today || t.TradeDate > oneYearAgo
);
```

Then this query will succeed as it is a subset of the specified domain

```
var res =trades.Where(
    t=>t.IsDestroyed == false && t.TradeDate == DateTime.Today.AddDays(-1)
).OnlyIfComplete().ToList();
```

This one too

```
res = trades.Where(
    t => t.IsDestroyed == false && t.MaturityDate == DateTime.Today
).OnlyIfComplete().ToList();
```

But this one will throw an exception

```
trades.Where(
    t => t.IsDestroyed == false && t.Portfolio == "SW-EUR"
).OnlyIfComplete().ToList()
```

Domain declaration and eviction policy are of course mutually exclusive on a datatype. Automatic eviction would make data incomplete.

What is Cachalot DB good at?

Cachalot DB is designed to be blazing fast and transactional. As always, there is a trade-off in terms of what it is not designed for.

The infamous [CAP Theorem](#) proves that a distributed system cannot be in the same time fault tolerant and transactionally consistent. We made the choice of full transactional consistency.

To be as fast as possible, Cachalot is fully memory cached. It means you need enough memory to store all your data. Each node loads everything in memory when it starts (Cachalot is a contraction of “Cache a lot” 😊)

Tests have been done up to 200 GB of data and one hundred million, medium size, objects. It can scale even more but, if you need to store more than 1 TB of data, it is probably not the right technology to choose.

It is designed to manipulate objects, as complex as you want. But it will always retrieve full objects. So, if you need to extract fragments of big data structures or if your data can not be naturally represented as collections of typed objects it is probably not the right technology to choose.

Administration

In this section we introduce a new member of the Cachalot ecosystem. The “Administration Console”. It is the **AdminConsole.exe** in your distribution

It can be used to:

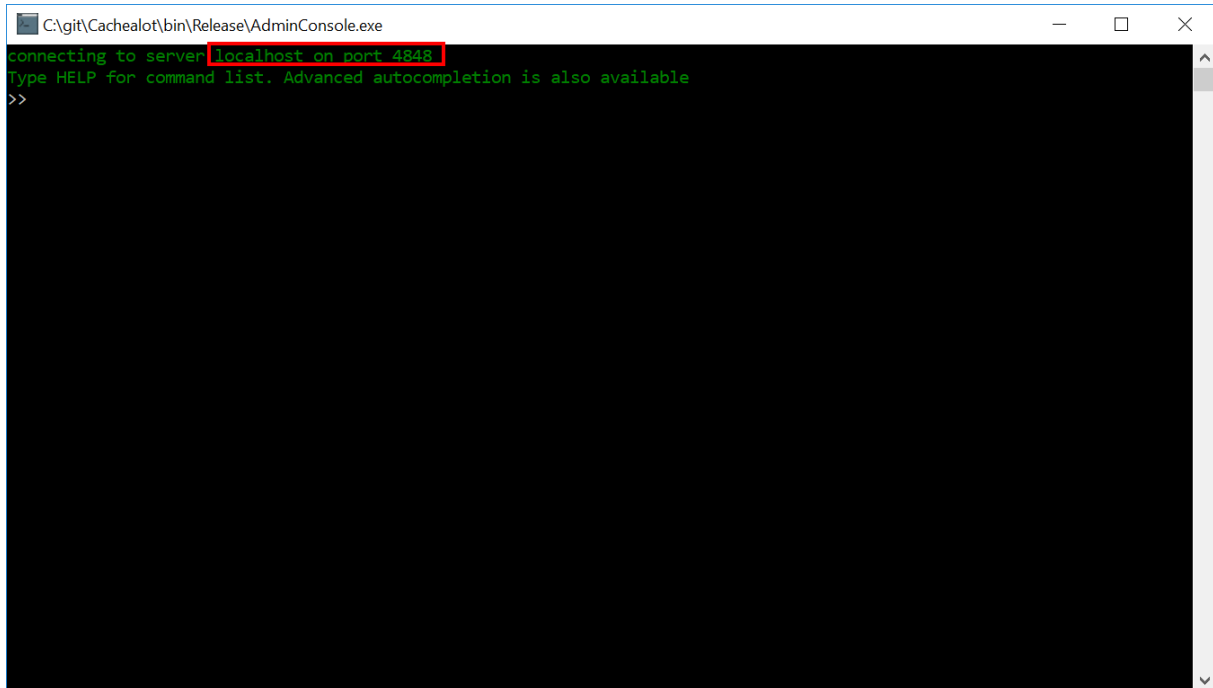
- visualize data
- create backup
- restore from backup
- change cluster configuration
 - o schema modification: add or delete indexes on an object
 - o add nodes to a cluster
- extract data to json files
- import data from json files
- switch database to read-only/read-write mode
- delete data
 - o drop a whole database
 - o delete a whole table
 - o delete items that match a condition

The first command to know is “help” which will display a list of available commands. “help” + “command name” will display a detailed explanation and examples.

A very powerful autocompletion feature is also available. Use TAB to activate it in almost every context

Connecting to the database

If you double-click the AdminConsole you will see this

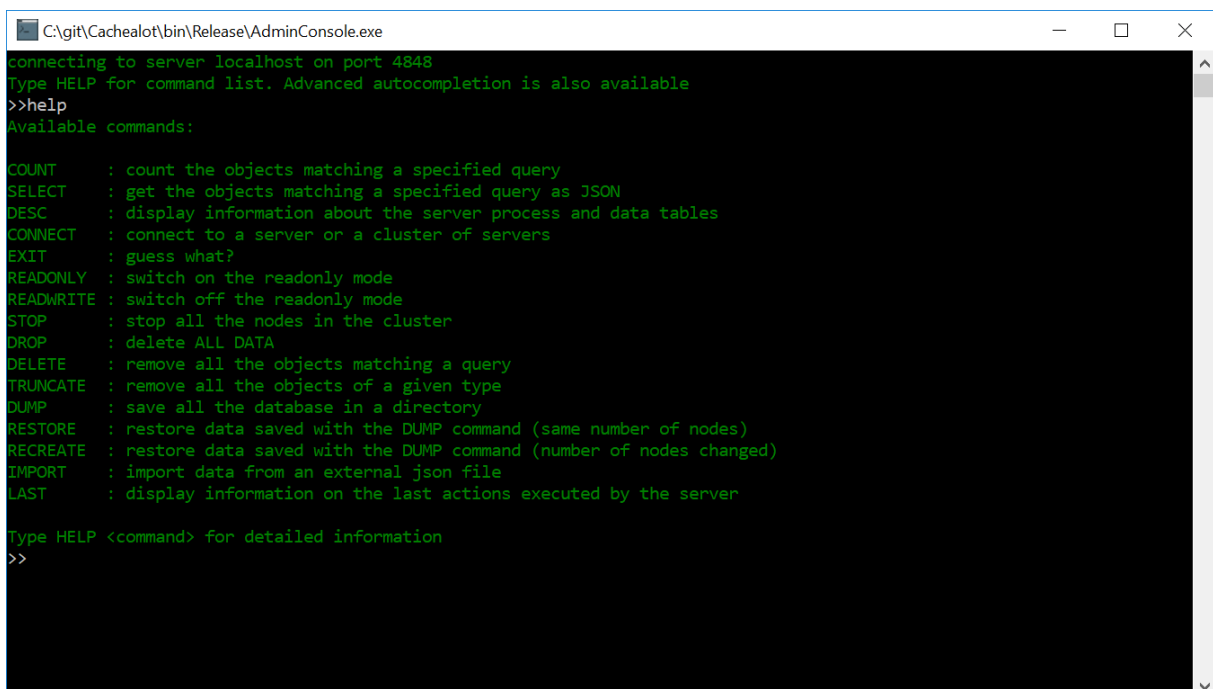


A screenshot of a Windows command prompt window titled "C:\git\Cachealot\bin\Release\AdminConsole.exe". The window shows the following text: "connecting to server localhost on port 4848", "Type HELP for command list. Advanced autocompletion is also available", and a prompt ">>". The text "localhost on port 4848" is highlighted with a red rectangle.

By default, it will try to connect to a single node on the current machine using the port 4848.

You can use **connect <hostname or IP address> <port>** to connect to a single node or **connect <client config.xml>** to connect to a cluster

Type h + TAB; this will be autocompleted to "help" and will display this



A screenshot of a Windows command prompt window titled "C:\git\Cachealot\bin\Release\AdminConsole.exe". The window shows the following text: "connecting to server localhost on port 4848", "Type HELP for command list. Advanced autocompletion is also available", ">>help", and a list of available commands. The text ">>help" is highlighted with a red rectangle.

```
Available commands:
COUNT      : count the objects matching a specified query
SELECT      : get the objects matching a specified query as JSON
DESC        : display information about the server process and data tables
CONNECT     : connect to a server or a cluster of servers
EXIT        : guess what?
READONLY    : switch on the readonly mode
READWRITE   : switch off the readonly mode
STOP        : stop all the nodes in the cluster
DROP        : delete ALL DATA
DELETE      : remove all the objects matching a query
TRUNCATE    : remove all the objects of a given type
DUMP        : save all the database in a directory
RESTORE     : restore data saved with the DUMP command (same number of nodes)
RECREATE    : restore data saved with the DUMP command (number of nodes changed)
IMPORT      : import data from an external json file
LAST        : display information on the last actions executed by the server

Type HELP <command> for detailed information
>>
```

Typing “help connect” will display this

```
>>help connect
Connect to a single node or a Cachalot cluster
connect (no parameter): by default connect to localhost 4848
connect server port : connect to a specific node
connect config.xml : connect to a cluster described by a configuration file
>>_
```

Visualizing data

The **desc** command will display all the data types that are known by the server/cluster and some information about each server process:

```
>>desc

Server process
-----
image type = 64 bits
started at = 01/11/2018 11:08:16
active clients = 1
threads = 7
physical memory = 148 MB
virtual memory = 878 MB
software version = 1.1.0.0

Tables
-----
|      Name |      Zip |
-----
|      Home |      False |
-----

The call took 6,3539339036821 milliseconds
>>
```

desc <type name> displays all the indexes on a data type.

```
>>desc home

HOME (BookingMarketplace.Home)
-----
|      property |      index type |      data type |      ordered |
-----
|      Id |      Primary |      IntKey |      False |
|      CountryCode |      ScalarIndex |      StringKey |      False |
|      Town |      ScalarIndex |      StringKey |      False |
|      Rooms |      ScalarIndex |      IntKey |      True |
|      Bathrooms |      ScalarIndex |      IntKey |      False |
|      PriceInEuros |      ScalarIndex |      IntKey |      True |
|      AvailableDates |      ListIndex |      IntKey |      False |
-----

The call took 7,92693655656655 milliseconds
>>_
```

A SQL-like query language is available in the admin console. It is a subset of the commands available through the LINQ provider. Only AND logical operator is supported and the CONTAINS operator is not supported.

“help select” will display a pretty good description of the query language:

```
>>help select
SELECT <table> WHERE <query> [INTO file.json]
Queries are expressed in a SQL-like language.
Index names are NOT query sensitives.

Examples:
priceineuros <= 100., Town = Paris
ValueDate = 2018-09-01 , IsDestroyed=0

Comma symbol stands for AND

Dates          as yyyy-mm-dd. No quotes
Strings        as is. No quotes. They ARE case sensitive
Integers       as is
Booleans       as 0 or 1
Enumerations   as integer value
Floating point values with a mandatory '.' decimal separator like 200. or 200.0
If INTO is used the data is saved as a json array in an external file
>>
```

As explained in the output of the help command, be aware of some pitfalls: no quotes are required for strings and dates, and the decimal point “.” is mandatory for float values. Dates are always formatted as **yyyy-mm-dd**.

The query language can be used with the **count**, **select**, **delete** commands. For example, count the apartments of at least three rooms in Paris France

```
>>count home where rooms > 3, countrycode=FR, town=Paris
Found 15000 items. The call took 33,6440 milliseconds
>>
```

Select will display a well formatted JSON array containing the result of your query

```
>>select home where id=1000
[
{
  "$type": "BookingMarketplace.Home, BookingMarketplace",
  "Id": 1000,
  "CountryCode": "FR",
  "Town": "Paris",
  "Adress": "14 rue du chien qui fume",
  "Rooms": 4,
  "Bathrooms": 1,
  "PriceInEuros": 248.0,
  "AvailableDates": []
}
]
```

Updating data

The most radical of the data modification commands is **drop**. It will completely delete all data from the database, including the schema (type definitions).

To completely delete all items of a type: **truncate <table name>**

To delete all the items matching a query: **delete <query>**

```
C:\git\Cachealot\bin\Release\AdminConsole.exe
connecting to server localhost on port 4848
Type HELP for command list. Advanced autocompletion is also available
>>count home
Found 92501 items. The call took 9,8043 milliseconds
>>delete home where rooms > 3
Deleted 35000 items. The call took 312,506088494426 milliseconds
>>count home
Found 57501 items. The call took 0,3353 milliseconds
>>truncate home
Deleted 57501 items. The call took 64,2907253256448 milliseconds
>>count home
Found 0 items. The call took 0,5339 milliseconds
```

After the execution of **truncate** command, the table does not contain data any more but the schema (type definition) is still present.

The **drop** command deletes all data from all tables and the type definitions

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

Server process
-----
image type = 64 bits
started at = 01/11/2018 11:08:16
active clients = 1
threads = 7
physical memory = 191 MB
virtual memory = 878 MB
software version = 1.1.0.0

Tables
-----
| Name | Zip |
-----
```

Data update has three steps:

- use a variant of **select** that allows to specify an external JSON file; the query result will be stored in this file instead of being displayed in the console
- update the json file with an external application (text editor or script for example)
- **import** the data from the external file into the database

```
>>count home where countrycode=FR
Found 92501 items. The call took 30,7321 milliseconds
>>select home where countrycode=FR into c:\dump\homes_in_france.json
Found 92501 items. The call took 3398,91626687127 milliseconds
>>import c:\dump\homes_in_france.json
Data successfully imported
>>
```

Backup and Restore

The backup/restore functions allow to

- save and recover a complete database
- transfer data to another database cluster
- change the type definitions (schema)

The **restore** command is designed to be very fast: data is restored in parallel by all the servers. But to be as fast as possible it does not allow data to be restored to a cluster which has a different number of nodes than the one who saved the backup.

A specific command is available to import data into a cluster with a different number of nodes. See next section.

The syntax of dump is **dump <existent root directory>**. The directory is usually a network drive as it needs to be accessible by all the servers in the cluster. Each dump will be saved in a subdirectory whose name is the current date in yyyy-mm-dd format. This subdirectory is created automatically if it does not exist.

Restoring data:

- restore <root directory> will restore the most recent dump available
- restore <root directory>/yyyy-mm-dd will restore a specific dump

The type definitions may be changed by editing a special file in the dump: **schema.json**. This is the only file from a dump who can be manually edited without compromising the dump integrity.

Editing it allows to:

- add indexes
- remove indexes
- change index definition (ordered or not)
- activate or deactivate compression.

An example of schema.json issued from a dump.

```
{
  "TypeDescriptions": [
    {
      ...
      "IndexFields": [
        {
          "KeyDataType": "StringKey",
          "KeyType": "ScalarIndex",
          "Name": "CountryCode",
          "IsOrdered": false
        },
        ...
      ],
      "ListFields": [
        {
          "KeyDataType": "IntKey",
          "KeyType": "ListIndex",
          "Name": "AvailableDates",
          "IsOrdered": false
        }
      ],
      "FullTypeName": "BookingMarketplace.Home",
      "TypeName": "Home",
      "UseCompression": false
    }
  ]
}
```

Change cluster configuration

To transfer complete databases between clusters with different numbers of nodes (or add nodes to an existing cluster) we use the usual dump command (from the previous section) but we restore data with a specific function: **recreate** with the same syntax as **restore**.

This function is significantly slower than **restore**. Instead of having each server restoring its data in parallel it reads the dump file on the client and feeds the data to all nodes in the new cluster. The target database needs to be empty (new or after drop) before executing this command.

Other commands

- readonly → switch data to read-only mode
- readwrite → switch off the read-only mode
- stop → stops all the servers in a cluster
- last <n> → display information on the last n commands executed on the server: server-side execution time, items affected, execution plan (primary index used). The execution plan may be used to optimize your queries and detect inefficient indexes