Cachalot DB - version 1.2.1

Quick Start and Administration Guide

Index

What is new in this version

It improves the full-text search: faster and more pertinent. See the corresponding section here.

All the executables target the platform netcore 3.1. The Nugget package containing the client code is netstandard 2.1. It can be used both in core and classical applications.

The admin console is now supported on Linux too. The Windows service was ported to netcore 3.1 (it will only run on Windows as a service).

We added a new eviction policy (in pure cache mode): TTL (Time To Live). It automatically removes those that are older than a given age (specified for each type of data).

A simple connection string can be used to connect to a cluster instead of the configuration file.

What is Cachalot DB?

Cachalot DB is more than one solution.

It is a distributed cache with unique features. It can do the usual things distributed caches do. Retrieve items by one or more unique keys and remove them according to different eviction policies. But it can do much more. Using a smart LINQ extension, you can run SQL-like queries, and they return a result set if and only if the whole subset of data is available in the cache.

Cachalot is also a fully-featured in-memory database. Like REDIS but with a full query model, not only a key-values dictionary. And unlike REDIS, it is entirely transactional.

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

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 that allows renting 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; }

        ooo
}
```

Now the object can be stored in the database.

The first step is to instantiate a **Connector** that 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.

We can read the configuration from an external file or specify it as a connection string. 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
}
```

There is one last step before storing an object in the database. We need to generate a unique value for the primary key. We can produce multiple unique values with a single call.

Unlike other databases, you do not need to create a unique value generator explicitly. The 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 the 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 the existing ones.

You probably have higher expectations from a modern database than merely 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

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

Using integer type is 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: we can define zero or more for each data type
- Index key: we can specify zero or more for each data 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 action is treated as 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 an 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 exciting 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 to 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 method has no direct equivalent in the classical SQL databases. It conveniently replaces most of the uses for the traditional JOIN operator.

You may need to create a query dynamically. For example, in a search screen, you add criteria to restrict your results. We can do it 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();
```

Dynamically building a query has the same effect as joining multiple ones with the && (AND) operator.

Full-text search

A very efficient and customizable full-text indexation is available starting version 1.1.3.

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 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 common 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, some words 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 us to specify them. This part should be identical for all nodes in a cluster.

```json
{
  "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. <u>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.</u>

Other methods of the API

In addition to querying and putting single items, the **DataSource** class exposes other important methods.

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 vast 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 choice allows us to generate data that will be inserted in the database dynamically. 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 beneficial. For an object that takes 10 KB in JSON, the compression ratio is around 1:10.

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

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

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

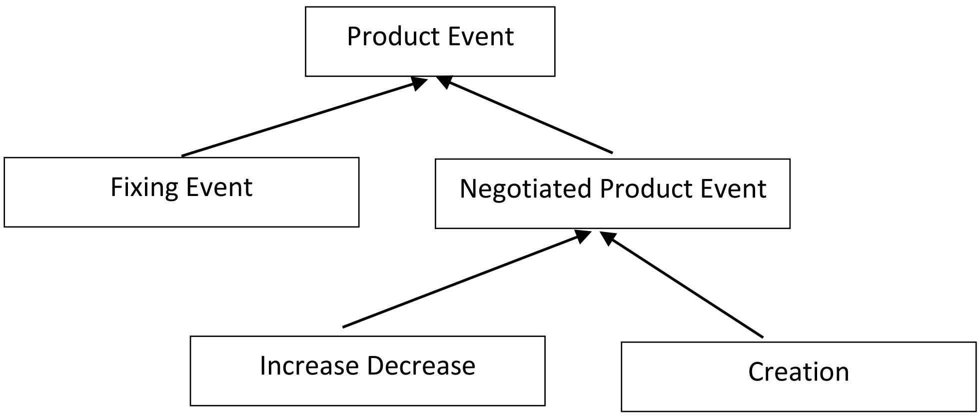
Compression only has an impact on the client-time

In conclusion, compression may be beneficial, 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 supported. Type information is stored internally in the JSON, and the client API uses it 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 us to expose indexed properties that 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; }

        ooo
}

public abstract class NegotiatedProductEvent: ProductEvent
{
        ooo
}

public class IncreaseDecrease : NegotiatedProductEvent
{
        ooo
        public override string EventType => "IncreaseDecrease";
}
```

Please find below an example of a code that 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 typical “put” operation adds an object or updates an existent one using the primary key as object identity.

More advanced use cases may be required:

1) **Add an object only if it is not already there and tell me if it was effectively 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 availability 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 choice 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).

We can achieve it in different ways:

  - Add 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)`**
  - Add 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 feature can be even more useful when committing multiple object modifications in a transaction. If a condition is not satisfied with one object, rollback the whole transaction.

See the next section.

Two-Stage Transactions

The most important thing to understand about two-stage transactions is when you need them.

Most of the time, you don't need them.

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 transfers between accounts. There are two types of business objects: **Account** and **AccountOperation**

```csharp
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 TransferedAmount { get; set; }

}
```

Let's create two accounts. There is 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**.

You could implement the business logic like this

```
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,
                        TransferedAmount = 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 we use conditional update (**UpdateIf**) and the condition is not satisfied by one object, the whole transaction rolls back.

In the previous example, we could, to prevent “monetary creation”, commit the transaction 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,
                    TransferedAmount = 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 is available 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 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, we load from an XML file.

```
var cfg = new ClientConfig();

cfg.LoadFromFile("two_nodes_cluster.xml");
```

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

Alternatively, we can describe the indexes 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 a 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 at the same time. Without a connection pool, their requests would be 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 the use of a reasonable quantity of TCP connections.

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 connect to the backend without any visible lag 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 also allows for the transparent recreation of connections if one or more nodes restart. The client does not need to reconnect explicitly.

```xml
<?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>
```

Starting with version 1.2, we can connect to a cluster or a single node with a simple connection string passed as a parameter to the **Connector** or the **ClientConfig.** An example of a connection string:

host1:port1+host2:port2;max_connections_in_pool, preloaded_connections

In-process server

In some cases, mainly 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 configuration 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 is done by simple in-process calls, not a TCP channel.

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

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

An external 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 pattern, the cache progressively fills 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 we reach a threshold, the cache automatically removes some of the objects.

An **eviction policy** is an algorithm used to decide which objects are removed.

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

Another supported policy is “Time To Live” abbreviated TTL. The objects have a limited lifespan, and they are removed when too old.

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

```json
{
  "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

```csharp
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();
    }
  }
```

Each data type can have a specific eviction policy (or none). The possible values in the current version are: **None**, **LeastRecentlyUsed** and **TimeToLive**

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 available 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. It 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 lazily loaded when we first access it.

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** (a 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 available into the cache

For this use-case Cachalot provides an inventive solution:

- Describe preloaded data as a query (expressed as LINQ expression)
- When querying data 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 merely 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 >= 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?

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

The infamous CAP Theorem proves that a distributed system cannot be at the same time fault-tolerant and transactionally consistent. We chose full transactional consistency.

To be as fast as possible, Cachalot stores all data in memory (and on disk if the persistent mode is activated). 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” 😊)

We have tested it 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.

We 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

We can use it to:

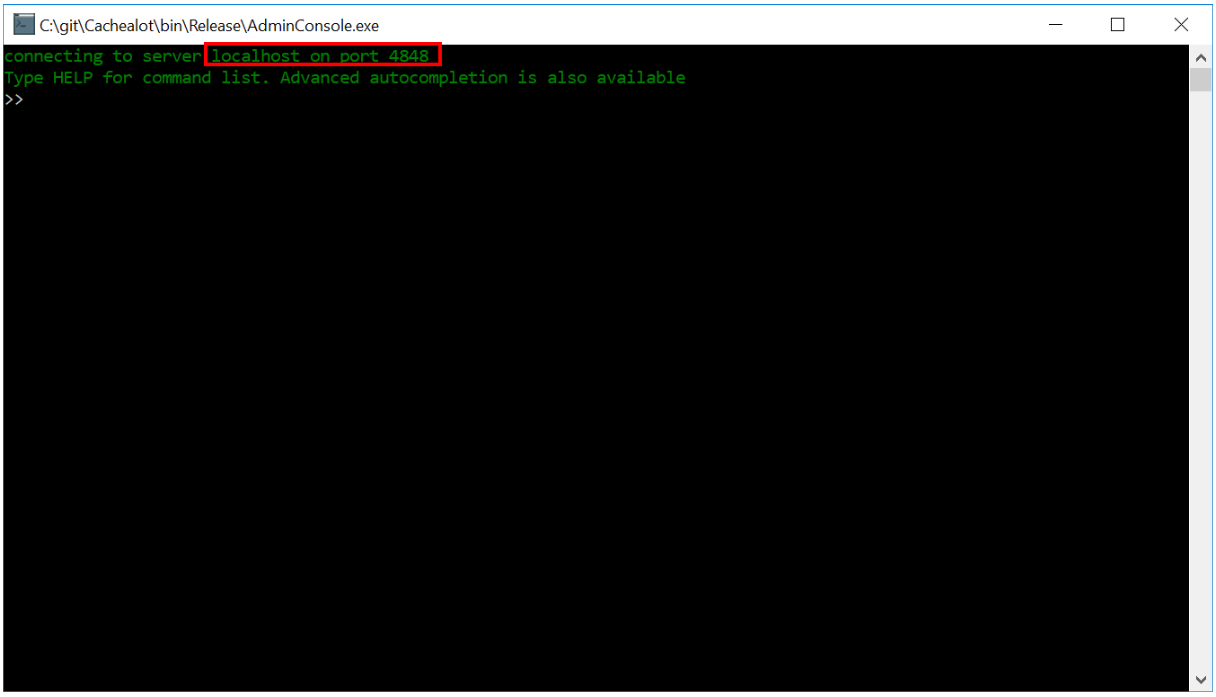
- visualize data
- create backups
- restore from backups
- change cluster configuration
    - schema modification: add or delete indexes on an object
    - add nodes to a cluster
- extract data to json files
- import data from json files
- switch database to read-only/read-write mode
- delete data
 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

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

A compelling 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



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. You can now also directly use a connection string like "host1:port1+host2:port2".

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

C:\git\Cachealot\bin\Release\AdminConsole.exe

```
connecting to server localhost on port 4848
Type HELP for command list. Advanced autocompletion is also available
>>help
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 accessible through the LINQ provider. Only AND logical operator is supported, and the CONTAINS operator is not supported (yet).

“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 miliseconds
>>_
```

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 thoroughly 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 miliseconds
>>delete home where rooms > 3
Deleted 35000 items. The call took 312,506088494426 miliseconds
>>count home
Found 57501 items. The call took 0,3353 miliseconds
>>truncate home
Deleted 57501 items. The call took 64,2907253256448 miliseconds
>>count home
Found 0 items. The call took 0,5339 miliseconds
```

After the execution of the **truncate** command, the table does not contain data anymore, 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 specifying 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 miliseconds
>>select home where countrycode=FR into c:\dump\homes_in_france.json
Found 92501 items. The call took 3398,91626687127 miliseconds
>>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)

We designed the **restore** command 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 restoration to a cluster that 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 the next section.

The syntax of the dump command is **dump <existent root directory>.** The directory is usually a network drive as it needs to be accessible by al 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

We can change the type definitions by editing a special file in the dump: **schema.json**. It is the only file from a dump that 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.

This is qn 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 is 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 on 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 previous 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