Cachalot DB - version 2

Quick Start Guide

What is Cachalot DB? 3
How fast is it? 4
Show me some code first 5
Server-side values 8
More code 9
Indexing collection properties 11
Coffee break 12
The connector 12
Collections and schemas 13
Data sources 14
Full-text search 15
Fine-tuning the full-text search 16
Computing pivot tables server-side 18
Other methods of the API 20
Deleting items from the database 20
Inserting or updating many objects 20
Deleting the whole content of the collection 20
Precompiled queries 21
Compressing object data 22
Storing polymorphic collections in the database 23
Conditional operations and "optimistic synchronization." 24
Two-Stage Transactions 26
Consistent read context 29
In-process server 30
Using Cachalot as a distributed cache with unique features 31
Serving single objects from a cache 31
Serving complex queries from a cache 33
First case: all data in the database is available into the cache 33
Second case: a subset of the database is available into the cache 35
What is Cachalot DB good at? 38

What is Cachalot DB?

An open-source in-memory database for dotnet applications

All data is available in memory and distributed on multiple nodes, allowing for blazing-fast queries. Like Redis but with some significant differences:

- Persistence is **transactional**; all update operations are written into a persistent transaction log before being applied to memory.
- It has a **full query model** available through SQL and LINQ. It supports all usual operators like projections, distinct, order by, and, of course, a complete where clause. Everything is done server-side.
- It supports both dictionary and ordered indexes. A highly optimized query processor chooses the best index, combining multiple indexes for one query.
- It can compute **pivot tables** server-side even on a cluster of many nodes
- A high-speed **full-text search**. You can do either full-text search, traditional queries, or combine the two.
- Very fast, **fully ACID, two-stage transactions** on multiple nodes
- **Consistent read context** allows executing a sequence of queries in a context that guarantees that data is not modified meanwhile.
- **Bulk inserts**: when feeding with large quantities of data, ordered indexes are sorted only once at the end to ensure optimum performance.
- When used as a distributed cache (persistence disabled), an inventive mechanism allows the description of the cached data, thus enabling complex queries to be served from cache only if **all the concerned data is available**.

How fast is it?

Very fast

Two demo applications are available in the release package:

- **BookingMarketplace** is testing feeding data and query capabilities
- **Accounts** is testing the transactional capabilities

Feel free to check by yourself. Here are some typical results on a reasonably powerful machine.

These results are for a cluster with two nodes. Most operations are faster as the number of nodes increases.

- Feeding one million objects into the database
  - 2678 milliseconds

- Reading 1000 objects (out of one million) by primary key
  - 219 milliseconds

- Reading 6000 objects (out of one million) with this query

```
select from home where town = Paris
```

  - 113 milliseconds

- Running this query on one million objects

```
select from home where town=Paris and AvailableDates
contains 2021-10-19 order by PriceInEuros descending
take 10
```

  - 22 milliseconds

- Computing a full pivot table (no filter) with two axes and two aggregations
  - 28 milliseconds

- Running a transaction with a conditional update
  - Less than two milliseconds

Show me some code first

Deep dive first, details after

Let's prepare our business objects for database storage.

We start with a toy website 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 Address { 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; }
}
```

To store a business object in a database, it needs a primary key. As there is no "natural" one, in this case, we will add a numeric Id.

```
public class Home
{
    [ServerSideValue(IndexType.Primary)]
    public int Id { get; set; }
    ...
}
```

Any simple type can be used for the primary key, for example, Guid, string, DateTime.

The first step is to instantiate a **Connector** that needs a connection string.

```
using var connector = new Connector("localhost:48401");
connector.DeclareCollection<Home>();
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.

In the case of a GUID primary key, we can generate the value locally; there is no collision risk.

For numeric primary keys, we can ask the database to generate the unique key.

It can produce multiple unique values at once.

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.

Three ways to do it:

```
var h = homes[property.Id];
// Or with a LINQ expression.
var h = homes.First(p => p.Id == home.Id);
// Or with SQL
var h = homes.SqlQuery($"select from home where
id={ids[i]}")
.First();
```

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

"Put" will insert new items (new primary key) and update the existing ones by default.

We will see later how to do conditional updates or "insert only if new."

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

Server-side values

An important design choice

In Cachalot DB, an object can be as complex as you want, but we can apply query expressions only on the root level properties tagged as server-side visible.

Both simple types and collections of simple types at the root level can be server-side visible.

That allows for a very efficient query processing and avoids any constraints on the serialization format that can be as compact as possible.

Server-side values can be indexed or not. We can query non-indexed server-side properties, but the queries will generally be more efficient on indexed ones.

Two types of indexes are available:

- **Dictionary**: very fast update and search, but only equality operators can exploit them
- **Ordered**: fast search, slower update but can be used efficiently for comparison operators and sorting

Massive insert/update operations (**DataSource.PutMany** method) are well optimized. After the size reaches a threshold (50 items by default), the action is treated as a **bulk insert**. In this case, ordered indexes are sorted only once at the end.

More code.

Adding indexes to the "Home" class

```
public class Home
{
    [ServerSideValue(IndexType.Primary)]
    public int Id { get; set; }

    [ServerSideValue(IndexType.Dictionary)]
    public string CountryCode { get; set; }

    [ServerSideValue(IndexType.Dictionary)]
    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; }

    [ServerSideValue(IndexType.Ordered)]
    public int Rooms { get; set; }

    [ServerSideValue(IndexType.Dictionary)]
    public int Bathrooms { get; set; }

    [ServerSideValue(IndexType.Ordered)]
    public decimal PriceInEuros { get; set; }
}
```

With these new indexes, we 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 cross the network to the client.

The "Contains" extension method is also supported

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

The previous LINQ expression 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 next.

Indexing collection properties

Let's enrich our business object. It would be helpful to have a list of available dates for each home.

Adding this new property enables some exciting features.

```
[ServerSideValue(IndexType.Dictionary)]
public List<DateTime> AvailableDates { get; set; }
     = new List<DateTime>();
```

It is a collection property, and it can be indexed the same way as the scalar properties.

Now we can search for homes available at a specific date

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

This method has no direct equivalent in the classical SQL databases, and it conveniently replaces some of the uses for the traditional JOIN operator.

Coffee break

Some explanations about what we have done so far

The connector

The first thing to do when connecting to a database is to instantiate a **Connector** class. The only parameter is a connection string.

```
var connector = new
Connector("localhost:48401+localhost:48402");
```

This line will connect to a cluster of two nodes on the local machine. Multiple nodes are separated by "+" and are specified as "machine: port." You can use hostnames or IP addresses to specify a machine.

Data is uniformly distributed on all the nodes in the cluster by using a sharding algorithm applied to the primary key.

The connector contains a connection pool for each node in the cluster. The connection pool has three responsibilities:

- Limit the number of simultaneous connections for a client
- Preload connections to speed up the first call
- Detect if the connections are still valid and try to open new ones otherwise
    - If a node in the cluster restarts, the connections are reestablished graciously

By default, the pool has a capacity of 4 connections, and one is preloaded. You can change this by adding `"; capacity, preloaded"` at the end of the connection string.

```
var connector = new Connector("SRVPRD1040:48401; 10, 4");
```

Instantiating a connector is quite expensive operation. It should be done only once in the application lifecycle. Disposing of the connector closes all the connections in the pool.

Collections and schemas

Once we have an instance of the connector, we need to declare the collections.

A name and a **CollectionSchema** define a collection. A schema contains all the information needed to convert an object from .NET to server representation and index it server-side.

```
connector.DeclareCollection<Home>();
```

This line is shorthand for declaring a collection with the same name as the type and a schema created automatically from the attributes on the properties in the class.

It is equivalent to:

```
var schema = TypedSchemaFactory.FromType(typeof(Home));
var name = typeof(Home).Name;
connector.DeclareCollection(name, schema);
```

When using the simplified version, you can also specify a different collection name.

```
connector.DeclareCollection<Home>("homes");
```

In all the examples, we have used attributes to define the properties that are visible server-side. There is another way: we can explicitly specify a schema.

```
var schema = SchemaFactory.New("heroes")
            .PrimaryKey("id")
            .WithServerSideCollection("tags")
            .WithServerSideValue("name", IndexType.Dictionary)
            .WithServerSideValue("age", IndexType.Ordered)
            .EnableFullTextSearch("tags", "name")
            .Build();
```

Multiple collections (with different names) can share the same schema.

When declaring a collection, if already defined on the server with a different schema, all data will be reindexed. This is a costly operation.

It is useful when deploying a new version of a client application, but otherwise, all clients should use the same schema.

Data sources

A data source is the client-side view of a server-side collection. It implements **IQueryable,** thus enabling direct use with LINQ expressions.

To get a data source from the connector, specify the type and eventually the collection name if different from the type name.

```
var homes = connector.DataSource<Home>();
var homes = connector.DataSource<Home>("homes");
```

Full-text search

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

First, we need to prepare the business objects for full-text indexation. We do it the usual way with a specific tag. Let's index as full text the **address**, the **town**, and the **comments**.

```
public class Home
{
…
        [FullTextIndexation]
        [ServerSideValue(IndexType.Dictionary)]
        public string Town { get; set; }

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

…

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

You notice that full-text indexation can be applied to ordinarily indexed properties and to properties that are not available to LINQ queries.

We can apply it to scalar and collection properties.

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

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

```
// pure full-text search
var result = homes.FullTextSearch("Paris close metro").ToList();

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

In both cases, the full-text query gives the order (most pertinent items first).

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.

```
{
  "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, and you may need to add other words depending on your business case. For example, it is good to avoid indexing "road" or "avenue" if you enable a full-text search on addresses.

Computing pivot tables server-side

A pivot table is a hierarchical aggregation of numeric values. A pivot definition consists of:

- An optional filter to restrict the calculation to a subset of data.
  - We can use any query, but operators like DISTINCT, ORDER BY or TAKE, make no sense for a pivot table calculation
- An ordered list of axes.
  - They are optional, too; if no one is specified, the aggregation is done on the whole collection, and the result contains a single level of aggregation
  - The axis must be server-side visible (indexed or not)
- A list of numeric values to aggregate (at least one must be specified)
  - They must be server-side visible (indexed or not)

For example, an **Order** class:

```
public class Order
{

      [ServerSideValue(IndexType.Primary)]
      public Guid Id { get; set; }

      // indexed value to aggregate
      [ServerSideValue(IndexType.Ordered)]
      public double Amount { get; set; }

      // non indexed value to aggregate
      [ServerSideValue]
      public int Quantity { get; set; }

      [ServerSideValue(IndexType.Dictionary)]
      public string Category { get; set; }

      [ServerSideValue(IndexType.Dictionary)]
      public int ProductId { get; set; }

      ...

}
```

We can aggregate the whole collection on **Amount** and **Quantity** and use **Category**, **ProductId** as axes.

```
var pivot = dataSource.PreparePivotRequest()
            .OnAxis(o => o.Category, o => o.ProductId)
            .AggregateValues(o=>o.Amount, o=>o.Quantity)
            .Execute();
```

To specify a filter, add a LINQ query as a parameter to the method **PreparePivotRequest**.

Calling pivot.ToString() will return:

```
ColumnName: Amount, Count: 100000, Sum: 1015000.00
ColumnName: Quantity, Count: 100000, Sum: 200000
    Category = science
    ColumnName: Amount, Count: 20000, Sum: 203000.00
    ColumnName: Quantity, Count: 20000, Sum: 40000
        ProductId = 1006
        ColumnName: Amount, Count: 10000, Sum: 101500.00
        ColumnName: Quantity, Count: 10000, Sum: 20000
        ProductId = 1001
        ColumnName: Amount, Count: 10000, Sum: 101500.00
        ColumnName: Quantity, Count: 10000, Sum: 20000
    Category = sf
    ColumnName: Amount, Count: 20000, Sum: 203000.00
    ColumnName: Quantity, Count: 20000, Sum: 40000
        ProductId = 1000
        ColumnName: Amount, Count: 10000, Sum: 101500.00
        ColumnName: Quantity, Count: 10000, Sum: 20000
        ProductId = 1005
        ColumnName: Amount, Count: 10000, Sum: 101500.00
        ColumnName: Quantity, Count: 10000, Sum: 20000
```

Color usage: Aggregate Axis Name Axis Value

Sum and **Count** are available as aggregation functions. We let you as an exercise to compute the **Average** 😊.

Other methods of the API

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

Deleting items from the database

```
// delete one object
homes.Delete(home);

// delete multiple objects
homes.DeleteMany(p => p.Town == "Paris");
```

Inserting or updating many objects

```
dataSource.PutMany(collection);
```

This method is very optimized for vast collections of objects

- Packets of objects are sent together in the network
- For massive groups, 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 to generation of data while inserting it into the database dynamically. The complete collection does not need to be present in the client's memory.

Deleting the whole content of the collection

```
dataSource.Truncate();
```

Precompiled queries

In general, the query parsing time is a tiny percentage of the data retrieval time.

Surprisingly the SQL parsing is faster than LINQ expression processing.

For the queries that return a small number of items and that are executed often, we can squeeze some more processing speed by precompiling them:

```
// you can replace this
var resultWithLinq = dataSource.Where(
      o => categories.Contains( o.Category)).ToList();

// by this (if the same query is used many times)
var query = dataSource.PredicateToQuery(
      o => categories.Contains( o.Category));

var resultWithPrecompiled =
      dataSource.WithPrecompiledQuery(query).ToList();
```

Compressing object data

The business objects are stored internally in a type-agnostic format.

Server-side properties are stored in a binary format that is an excellent compromise between memory consumption and comparison speed, and all the object data is stored as UTF-8 encoded JSON.

"Packing" is the process of transforming a .NET object in the internal format.

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.

By default, the JSON data is not compressed, but compression may benefit objects that take more than a few kilobytes.

For an object that takes 10 KB in JSON, the compression ratio is around 1:10.

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

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

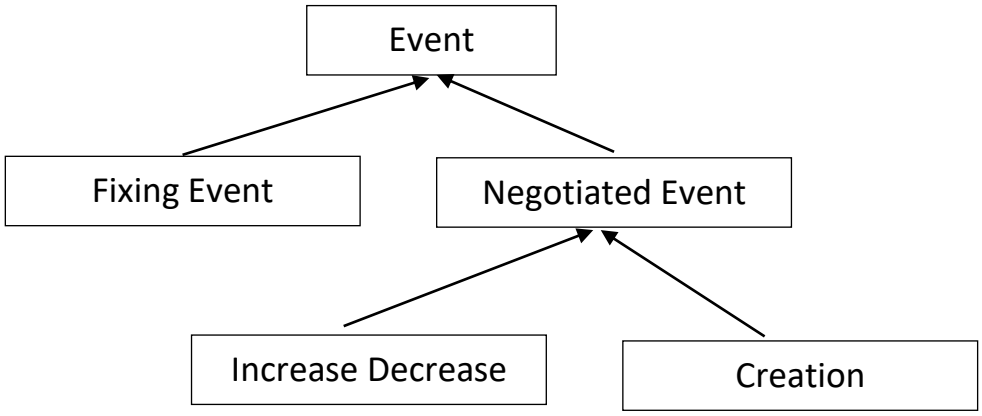
Using compressed objects is transparent for the client code. However, it has an impact on packing time. When objects are retrieved, they are unpacked (which may imply decompression).

In conclusion, compression may be beneficial, starting with medium-sized 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 (as $type property), and the client API uses it to deserialize the proper concrete type. The base type can be abstract.

A small example from a trading system



To store a polymorphic collection, we must expose all required server-side values on the base type.

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

Example of code that queries a collection having an abstract class as type.

```
var events = connector.DataSource<Event>();
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 existing one using the primary key as object identity.

More advanced use cases may arise:

1) Add an object only if it is not already there and tell me if it was effectively added
2) Update an existing 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** class. If the object was already there, it is not modified, and the return value is 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 handy 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. Otherwise, the operation fails, and we must retry (load + edit + save).

We can achieve this 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.

```
item.Version = n + 1;
data.UpdateIf(item, i=> i.Version == n);
```

- Add a timestamp on an object. When we save a modified object, we require the timestamp of the version in the database is identical to that of the object before our update.

```
var oldTimestamp = item.Timestamp;
item.Timestamp = DateTime.Now;

data.UpdateIf(item, I => i.Timestamp == oldTimestamp);
```

This feature can be even more helpful when committing multiple object modifications in a transaction. If a condition is not satisfied with one object, roll back the whole transaction.

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.

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

It is durable (operations are synchronously saved to an append-only transaction log) and 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.**

The complete example is included in the release package (**Accounts** application).

Imagine lots of money transfers happening in parallel.

We would like to:

- Subtract an amount of money from the source account only if the balance is superior to the amount
- Add the same amount to the target account
- Create a money-transfer object to keep track of accounts history

All this should happen (or fail) as an atomic operation.

The business object definition:

```
// this is an account containing only it's current balance
public class Account
{

      [ServerSideValue(IndexType.Primary)]
      public int Id { get; set; }

      [ServerSideValue(IndexType.Ordered)]
      public decimal Balance { get; set; }
}

// this is a money transfer between accounts
public class MoneyTransfer
{

      [ServerSideValue(IndexType.Primary)]
      public int Id { get; set; }

      [ServerSideValue(IndexType.Ordered)]
      public decimal Amount { get; set; }

      [ServerSideValue(IndexType.Dictionary)]
      public DateTime Date { get; set; }

      [ServerSideValue(IndexType.Dictionary)]
      public int SourceAccount { get; set; }


      [ServerSideValue(IndexType.Dictionary)]
      public int DestinationAccount { get; set; }

}
```

The transfer as a transaction:

```
srcAccount.Balance -= transferred;
dstAccount.Balance += transferred;

var transaction = connector.BeginTransaction();
transaction.UpdateIf(srcAccount,
      account => account.Balance >= transferred
      );
transaction.Put(dstAccount);
transaction.Put(new MoneyTransfer
{
      Amount = transferred,
      Date = DateTime.Today,
      SourceAccount = src,
      DestinationAccount = dst,
      Id = ids[i]
});


transaction.Commit();
```

The operations allowed inside a transaction are:

- Put
- Delete
- DeleteMany
- UpdateIf

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

Consistent read context

Consistent read is new functionality available in version 2.

It enables multiple queries to be executed in a context that guarantees that data do not change during the execution of all the queries. Multiple **ConsistentRead** methods are available on the Connector class.

A simplified version of the method is available for up to four collections when using default collection names.

```
connector.ConsistentRead<MoneyTransfer, Account>(ctx =>
{
      var myAccounts = ctx.Collection<Account>().ToList();

      Assert.AreEqual(2, myAccounts.Count);

      // The sum of the accounts balance should always be 1000
      Assert.AreEqual(1000, myAccounts.Sum(acc => acc.Balance));

      var transfers = ctx.Collection<MoneyTransfer>();

      var tr = transfers
            .Where(t => t.SourceAccount == myAccounts[0].Id).ToList();

      //check consistency between transfer and balance
      var sumTransferred = tr.Sum(tr => tr.Amount);

      Assert.AreEqual(sumTransferred, myAccounts[1].Balance);


});
```

When using explicit collection names, this version of the method should be used

```
void ConsistentRead(Action<ConsistentContext> action,
 params string[] collections)
```

In-process server

In some cases, if the quantity of data is bounded and a single node has enough memory to keep all the data, 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 string as a connection string to the **Connector** constructor.

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

This will instantiate a server inside the connector object, and communications will be done by simple in-process calls, not a TCP channel.

The **Connector** class implements **IDisposable**. Disposing of the **Connector** will graciously stop the server.

The connector should be instantiated and disposed of only once in the application lifetime.

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 the primary key.

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. Usually, when we load an object from the database, we also store it into the cache.

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

The cache progressively fills with data when using this simple pattern, and its hit-ratio improves over time.

This cache usage is usually associated with an "eviction policy" to avoid excessive memory consumption.

An eviction policy is an algorithm used to decide which objects to remove.

- 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 items with the oldest timestamp.
- Another supported policy is "Time To Live," abbreviated **TTL**. The objects have a limited lifespan, and ze remove them 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

```
{
  "IsPersistent": true,
  "ClusterName": "test",
  "TcpPort": 48401,
  "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

Each collection 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 helpful in some real-world cases like storing session information for websites, 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.

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 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 methods are available in the **DataSource** class to manage this use case.

- 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.
- A method used to declare that all data is available for a given data type: **DeclareFullyLoaded.**

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, the cache will 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 all houses in the most visited cities in the cache.

```
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()
```

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()
```

In both cases, If we omit the call to **OnlyIfComplete,** it will merely return the elements in the cache that match the query.

Domain declaration and eviction policy are, of course, mutually exclusive on a collection. 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, and we chose transactional consistency.

To achieve high-speed data access, it loads all data in memory.

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 up to 200 GB of data and one hundred million medium-sized objects per collection. It can scale even more, but it is probably not the right technology to choose if you need to store more than 1 TB of data.

We can use it as a very efficient cache for big data applications but not the golden source.