vyhodb

Getting started

Copyright © 2015 Igor Vykhodtsev

Copyright © 2015 Igor Vykhodtsev

Introduction

Vyhodb is a database management system, which uses network data model, supports ACID transactions, written on Java and intended to be used by Java applications.

This document aim is to give brief info about vyhodb API and to show how to use it.

The audience of this document is software developers and architects who are going to use vyhodb in their projects. Knowledge of Java language is essential. Knowledge of any other technologies (like J2EE, Spring) isn't required.

Almost all sections in this document give brief information about particular vyhodb API and show usage example:

API	Description
Server API	Starting/stopping vyhodb server in embedded mode; transaction management.
Space API	Reading/modification vyhodb data.
Indexes	Using indexes.
RSI	Remote service invocation. Services are objects which are executed inside vyhodb and have access to Space API.
Functions API	Traversing over vyhodb records.
ONM API	Mapping between java classes and vyhodb records. Reading/writing graph of java records from/to vyhodb.

This guide is included in the vyhodb documentation package which consists of the following documents:

Document	Description
Getting Started	Fast start. Document gives idea what is vyhodb API about using simple code examples.
Developer Guide	Describes different vyhodb APIs and how to use them.
Functions Reference	Functions API Reference. Describes functions with usage examples.
Administrator Guide	Describes vyhodb architecture, configuring and administration.

Examples

You can download example source code from here http://www.vyhodb.com/doku.php?id=docs.

For running examples, you need to do the following actions:

1) Configure your IDE and include into classpath all jar archives from vyhodb **lib** directory.
2) Change in each example static fields LOG, DATA, which point to the storage files. Storage files are located in vyhodb **storage** directory.

Copyright © 2015 Igor Vykhodtsev

Vyhodb distribution package, installation and starting

Download JRE

vyhodb requires JRE version not lower than 7u64. The latest version, as well as instructions for installing JRE can be found here http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Download vyhodb

The latest version of the vyhodb distribution package can be downloaded from download page. The distribution package is a zip file with directory **vyhodb-0.9.0** inside.

Configure path to JRE

After unzipping **vyhodb-0.9.0** directory, you need to specify path to JRE:

Windows:

1) Open file **bin-cmd\set-env.cmd**
2) Set **JRE_HOME** variable. For example: SET JRE_HOME=C:\Program Files\Java\jdk1.7.0_60\jre

Linux:

1) Open file **bin-sh/set-env.sh**
2) Set **JRE_HOME** variable. For example: JRE_HOME=/home/jdk1.7.0_79/jre

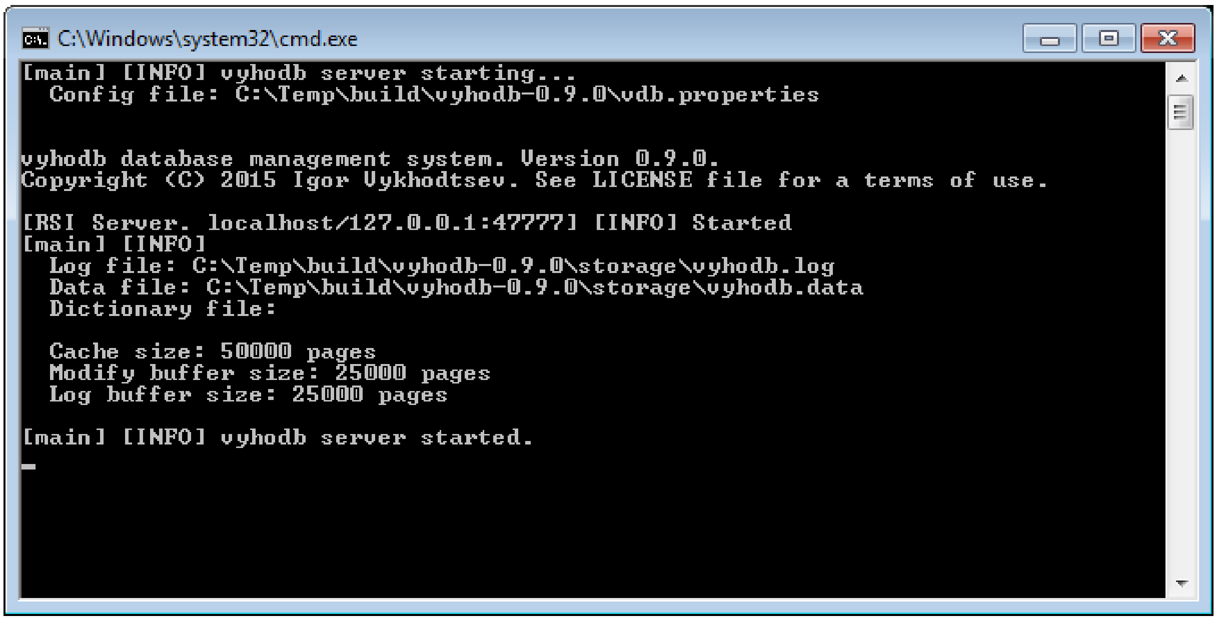
Run

Now you can start vyhodb server in so called stand-alone mode. Run one of the following scripts:

Windows: **vdb-start.cmd**

Linux: **vdb-start.sh**

Picture below shows console output of running vyhodb server:



```
[main] [INFO] vyhodb server starting...
  Config file: C:\Temp\build\vyhodb-0.9.0\vdb.properties


vyhodb database management system. Version 0.9.0.
Copyright (C) 2015 Igor Vykhodtsev. See LICENSE file for a terms of use.

[RSI Server. localhost/127.0.0.1:47777] [INFO] Started
[main] [INFO]
  Log file: C:\Temp\build\vyhodb-0.9.0\storage\vyhodb.log
  Data file: C:\Temp\build\vyhodb-0.9.0\storage\vyhodb.data
  Dictionary file:

  Cache size: 50000 pages
  Modify buffer size: 25000 pages
  Log buffer size: 25000 pages

[main] [INFO] vyhodb server started.
_
```

To stop vyhodb server use **Ctrl+C** combination or **vdb-close-remote** script (can be found in **bin-cmd** or **bin-sh** directories).

Copyright © 2015 Igor Vykhodtsev

Vyhodb directory layout

File	Description
bin-cmd	windows command line utilities
bin-sh	linux command line utilities
lib	vyhodb system jar files
services	Directory for RSI Services jar files
storage	Default directory for storage files (data file and log file)
LICENSE	vyhodb license agreement
vdb.properties	vyhodb configuration file
vdb-start.cmd	windows start script
vdb-start.sh	linux start script

Copyright © 2015 Igor Vykhodtsev

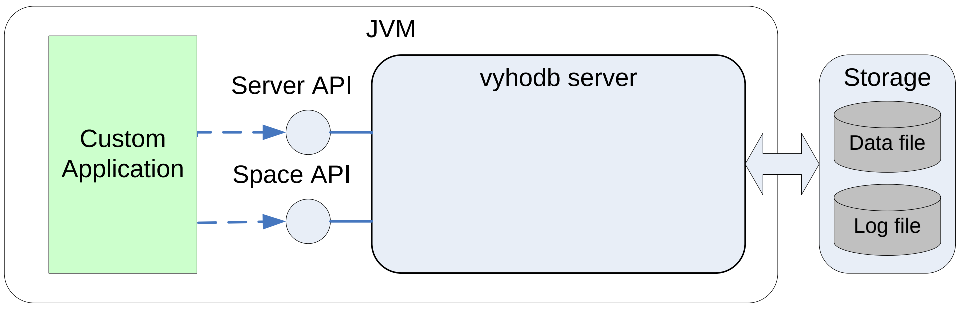
Running modes

vyhodb supports the following running modes[1]:

1) Embedded
2) Standalone

Embedded

In this mode, custom application and vyhodb server are running in the same JVM. Custom application uses **Server API** for starting vyhodb server and transaction management. **Space API** is used by custom application for reading and modifying vyhodb data.

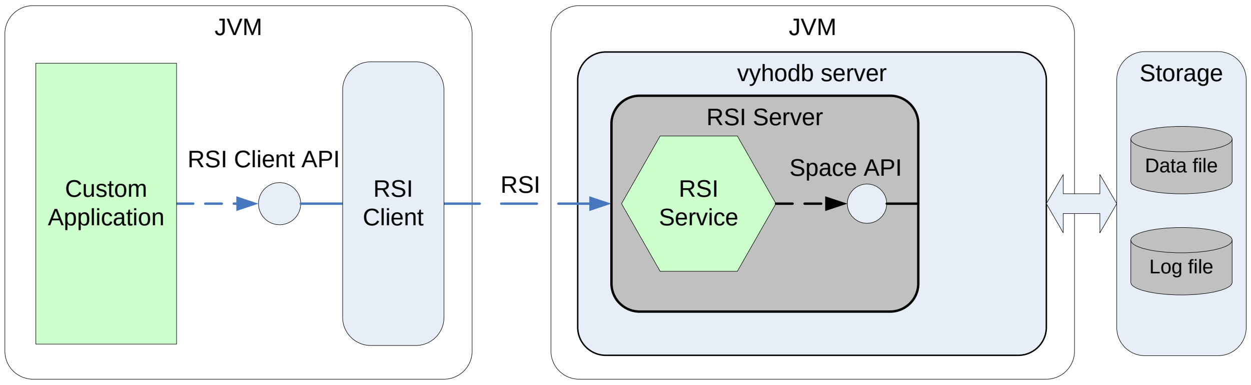


In all code examples (except RSI section) we use this mode for starting vyhodb server.

Standalone

In this mode, custom application and vyhodb server are physically separated and running in different JVMs. Remote Service Invocation (RSI) mechanism is used for communication between them. Section **RSI** gives an example of RSI Service developing and using it.

There are scripts in vyhodb root directory for starting vyhodb in standalone mode: **vdb-start.cmd, vdb-start.sh.**



Custom application in this mode invokes methods of RSI Services, which implement business logic and are running inside vyhodb server (in RSI Server component of vyhodb server).

RSI Server component manages lifecycle of RSI Services. It opens new transaction for each remote method invocation and closes it after method completion. Custom application uses RSI Client API for establishing connections to remote vyhodb server.

[1] There is one more running mode exists - “Local”. In this mode, custom application can start vyhodb server in its JVM, but use RSI technology to access RSI Services. See “Developer Guide” and “Administrator Guide” documents for more information about this mode.

Copyright © 2015 Igor Vykhodtsev

Server API

Server API in intended for starting vyhodb server in embedded mode and transaction management.

In example below, we start vyhodb server in embedded mode, open Modify transaction, change root record (about records and vyhodb data model see next section **Space API**) and commit transaction. After that we open Read transaction and read modified data.

```
package com.vyhodb.started.server;

import java.io.IOException;
import java.util.Date;
import java.util.Properties;

import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;

public class ServerAPI {

    public static final String LOG = "C:\\vyhodb-0.9.0\\storage\\vyhodb.log";
    public static final String DATA = "C:\\vyhodb-0.9.0\\storage\\vyhodb.data";

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.setProperty("storage.log", LOG);
        props.setProperty("storage.data", DATA);

        try(Server server = Server.start(props)) {
            modifyTrx(server);
            readTrx(server);
        }
    }

    private static void modifyTrx(Server server) {
        TrxSpace space = server.startModifyTrx();   // Starts modify transaction

        Record root = space.getRecord(0L);          // Retrieves root record
        root.setField("Current Time", new Date());  // Changes field

        space.commit();                             // Commits transaction
    }

    private static void readTrx(Server server) {
        TrxSpace space = server.startReadTrx();     // Starts read transaction

        Record root = space.getRecord(0L);          // Retrieves root record
        Date date = root.getField("Current Time");  // Gets field value
        System.out.println(date);

        space.rollback();                           // Rolls back transaction
    }
}
```

Note, that you need to change values of LOG, DATA constants so that they point to vyhodb storage files (by default shipped with vyhodb distributive package and can be found in **storage** directory). Other code examples in this document should be changed the same way.

Output:

```
vyhodb database management system. Version 0.9.0.
Copyright (C) 2015 Igor Vykhodtsev. See LICENSE file for a terms of use.

[main] [INFO]
  Log file: C:\vyhodb-0.9.0\storage\vyhodb.log
  Data file: C:\vyhodb-0.9.0\storage\vyhodb.data
  Dictionary file:

  Cache size: 50000 pages
  Modify buffer size: 25000 pages
  Log buffer size: 25000 pages

[main] [INFO] vyhodb server started.
Mon Sep 07 03:34:02 BRT 2015
[main] [INFO] vyhodb server closed.
```

Most part of output are logging messages. In next examples, they will be omitted and only valuable output will be shown. For instance for our example it is:

```
Thu Jul 09 07:28:10 BRT 2015
```

Copyright © 2015 Igor Vykhodtsev

Space API

Space API is intended for reading and modifying vyhodb data.

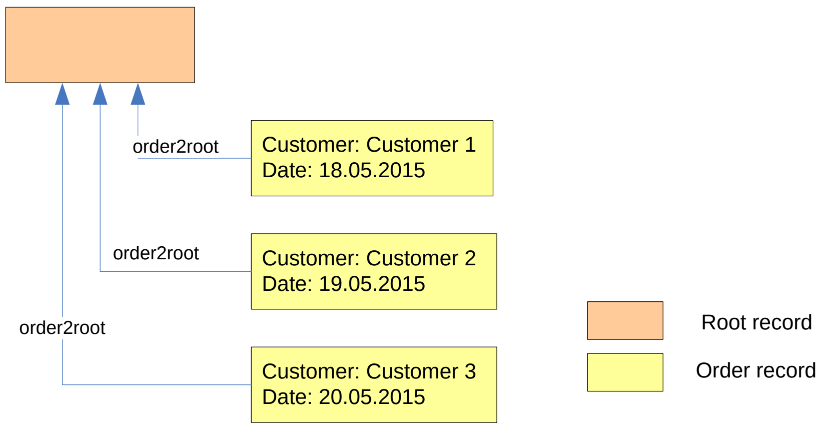
Vyhodb uses network data model, which is comprised of records, fields and links:

- **Record** – is a basic concept of vyhodb data model. It is a container for fields. Each record has unique identifier of long type, which is automatically assigned by vyhodb. Record with id == 0 is a so called root record. It is automatically created at the new storage creation time (this record we changed in previous example).
- **Field** – is named value stored inside record. Field names are unique within particular record.
- **Link** connects two records and has a name. Link has a direction from source (child) record to destination (parent) record. Link names from child record must be unique (no outbound links with identical names). From the parent record there is an ability to iterate over its child records.

All functionalities for working with records, fields and links are concentrated in two interfaces: **com.vyhodb.space.Space** and **com.vyhodb.space.Record**.

com.vyhodb.server.TrxSpace interface plays two roles at the same time: space of records (storage) and active transaction.

In example below we create records structure shown on next diagram:



Example:

```
package com.vyhodb.started.space;

import java.io.IOException;
import java.util.Date;
import java.util.Properties;

import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;

public class SpaceAPI {
    public static final String LOG = "C:\\vyhodb-0.9.0\\storage\\vyhodb.log";
    public static final String DATA = "C:\\vyhodb-0.9.0\\storage\\vyhodb.data";

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.setProperty("storage.log", LOG);
        props.setProperty("storage.data", DATA);
```

Copyright © 2015 Igor Vykhodtsev

```
        try(Server server = Server.start(props)) {
            TrxSpace space = server.startModifyTrx();
            example(space);
            space.rollback();
        }
    }

    private static void example(Space space) {
        Record root = space.getRecord(0L);                  // Retrieves record by id

        Record order1 = space.newRecord();                  // Creates new record
        order1.setField("Customer", "Customer 2");          // Sets field
        order1.setField("Date", new Date("05/19/2015"));    // Sets field
        order1.setParent("order2root", root);               // Creates link to root record

        Record order2 = space.newRecord();
        order2.setField("Customer", "Customer 3");
        order2.setField("Date", new Date("05/20/2015"));
        order2.setParent("order2root", root);

        Record order3 = space.newRecord();
        order3.setField("Customer", "Customer 1");
        order3.setField("Date", new Date("05/18/2015"));
        order3.setParent("order2root", root);

        // Retrieves child records and iterates over them
        for (Record order : root.getChildren("order2root")) {
            System.out.println(order);
        }
    }
}
```

Similar to previous example, we start vyhodb server in embedded mode and open Modify transaction.

After that, in **example()** method, we create three records, which logically correspond to orders. We set **"Customer"**, **"Date"** fields on each one. Then, we create links from each created record to root record with a link name **"order2root"**. So created records are child records from root record's perspective.

Finally, we get child records from root records, iterate over ones and print to console.

Output (record ids can differ):

```
{Customer="Customer 2", Date="Tue May 19 00:00:00 BRT 2015"} id=523
{Customer="Customer 3", Date="Wed May 20 00:00:00 BRT 2015"} id=534
{Customer="Customer 1", Date="Mon May 18 00:00:00 BRT 2015"} id=545
```

Copyright © 2015 Igor Vykhodtsev

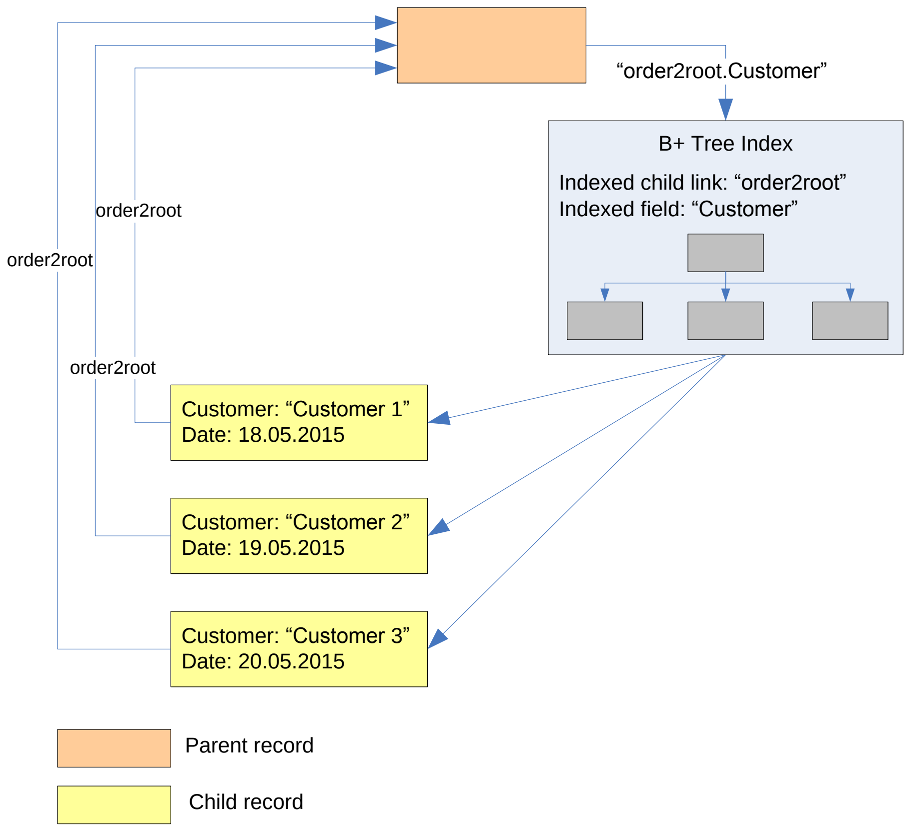
Indexes

Indexes are used for fast search of child records by their field's values.

They are created from parent record side and indexing fields of child records.

There is an example below, where we create index with name **"order2root.Customer"** on parent record. This index is built upon **"Customer"** field values from child records which in turns reference to root record by links with **"order2root"** name.

Records and index, created in example are illustrated on diagram below:



Example (main() method and import directives are the same as in previous example):

```
package com.vyhodb.started.index;

. . .

public class Indexes {

. . .

    private static void example(Space space) {
        Record root = space.getRecord(0L);

        // Creates records
        {
            Record order1 = space.newRecord();
            order1.setField("Customer", "Customer 2");
            order1.setField("Date", new Date("05/19/2015"));
            order1.setParent("order2root", root);

            Record order2 = space.newRecord();
            order2.setField("Customer", "Customer 3");
            order2.setField("Date", new Date("05/20/2015"));
            order2.setParent("order2root", root);

            Record order3 = space.newRecord();
            order3.setField("Customer", "Customer 1");
            order3.setField("Date", new Date("05/18/2015"));
            order3.setParent("order2root", root);
        }

        // Creates index which indexes "Customer" field on child records
        createIndex(root);

        // Creates search criteria
        Criterion criterion = new Equal("Customer 3");

        // Searches records and iterates over search result
        for (Record order : root.searchChildren("order2root.Customer", criterion)) {
            System.out.println(order);
        }
    }

    private static void createIndex(Record record) {
        String fieldName = "Customer";
        String linkName = "order2root";
        String indexName = "order2root.Customer";

        // Creates indexed field descriptor
        IndexedField indexedField = new IndexedField(
                fieldName,
                String.class,
                Nullable.NOT_NULL
        );

        // Creates index descriptor
        IndexDescriptor indexDescriptor = new IndexDescriptor(
                indexName,
                linkName,
                Unique.DUPLICATE,
                indexedField
        );

        // Creates index
        record.createIndex(indexDescriptor);
    }
}
```

Output:

```
{Customer="Customer 3", Date="Wed May 20 00:00:00 BRT 2015"} id=534
```

For more details about indexes see section “Indexes” in “Developer Guide” document.

Copyright © 2015 Igor Vykhodtsev

RSI

Remote Service Invocation (RSI) – is a technology provides ability to call methods of RSI Services remotely.

RSI Service is a java object which is instantiated and lives inside vyhodb server and has access to Space API.

In this section we implement simple RSI Service and client application for invoking its methods remotely. For more information about RSI developing see “Developer Guide”, for information about RSI configuring see “Administrator Guide”.

Firstly, we define Service’s contract:

```
package com.vyhodb.started.rsi;

import java.util.Collection;
import java.util.Date;

import com.vyhodb.rsi.Implementation;
import com.vyhodb.rsi.Modify;
import com.vyhodb.rsi.Read;

@Implementation(className="com.vyhodb.started.rsi.ServiceImpl")
public interface Service {

    @Modify
    public long newOrder(String customerName, Date date);

    @Read
    public Collection<String> listCustomers();
}
```

Now we implement RSI Service class, which objects are instantiated by vyhodb server and lives inside it:

```
package com.vyhodb.started.rsi;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;

import com.vyhodb.space.Record;
import com.vyhodb.space.ServiceLifecycle;
import com.vyhodb.space.Space;

public class ServiceImpl implements Service, ServiceLifecycle {

    private Space _space;

    @Override
    public void setSpace(Space space) {
        _space = space;
    }

    @Override
    public long newOrder(String customerName, Date date) {
        Record root = _space.getRecord(0L);

        Record order = _space.newRecord();
        order.setField("Customer", customerName);
        order.setField("Date", date);
        order.setParent("order2root", root);

        return order.getId();
    }
```

Copyright © 2015 Igor Vykhodtsev

```
    @Override
    public Collection<String> listCustomers() {
        ArrayList<String> result = new ArrayList<>();

        Record root = _space.getRecord(0L);
        for (Record order : root.getChildren("order2root")) {
            result.add((String) order.getField("Customer"));
        }

        return result;
    }
}
```

Finally, we create client application, which creates network connection to vyhodb server and invokes methods of RSI Service:

```
package com.vyhodb.started.rsi;

import java.io.IOException;
import java.net.URISyntaxException;
import java.util.Collection;
import java.util.Date;

import com.vyhodb.rsi.Connection;
import com.vyhodb.rsi.ConnectionFactory;
import com.vyhodb.rsi.RsiClientException;

public class Client {

 public static final String URL = "tcp://localhost:47777";

    public static void main(String[] args) throws RsiClientException, IOException,
URISyntaxException {

        try(Connection connection = ConnectionFactory.newConnection(URL)) {

            Service service = connection.getService(Service.class);

            service.newOrder("Customer 3", new Date("05/20/2015"));
            service.newOrder("Customer 1", new Date("05/18/2015"));
            service.newOrder("Customer 2", new Date("05/19/2015"));

            Collection<String> customers = service.listCustomers();
            System.out.println(customers);
        }
    }
}
```

When all classes are ready, the following steps should be done to run our example:

1) Pack Service and ServiceImpl classes into jar archive.
2) Stop vyhodb server, if it runs.
3) Copy jar archive into **services** directory.
4) Start vyhodb serve in standalone mode. Use the following scripts for this: vdb-start.cmd/vdb-start.sh
5) Run **com.vyhodb.started.rsi.Client** class.

Output of Client class:

```
[Customer 3, Customer 1, Customer 2]
```

Vyhodb doesn’t support “hot” deployment that is why it should be restarted.

Copyright © 2015 Igor Vykhodtsev

For your convenience, **vdb-started-rsi.jar** file is shipped alone with documentation package. This archive contains compiled classes from this section’s examples. For running it you need to do the following steps:

1) Copy **vdb-started-rsi.jar** into **services** directory.
2) Start/restart vyhodb server in standalone mode.
3) Change current directory to vyhodb root directory and run one of the commands (according to your platform):
    a. Windows: **java -cp lib/*;services/* com.vyhodb.started.rsi.Client**
    b. Linux: **java –cp lib/*:services/* com.vyhodb.started.rsi.Client**

Copyright © 2015 Igor Vykhodtsev

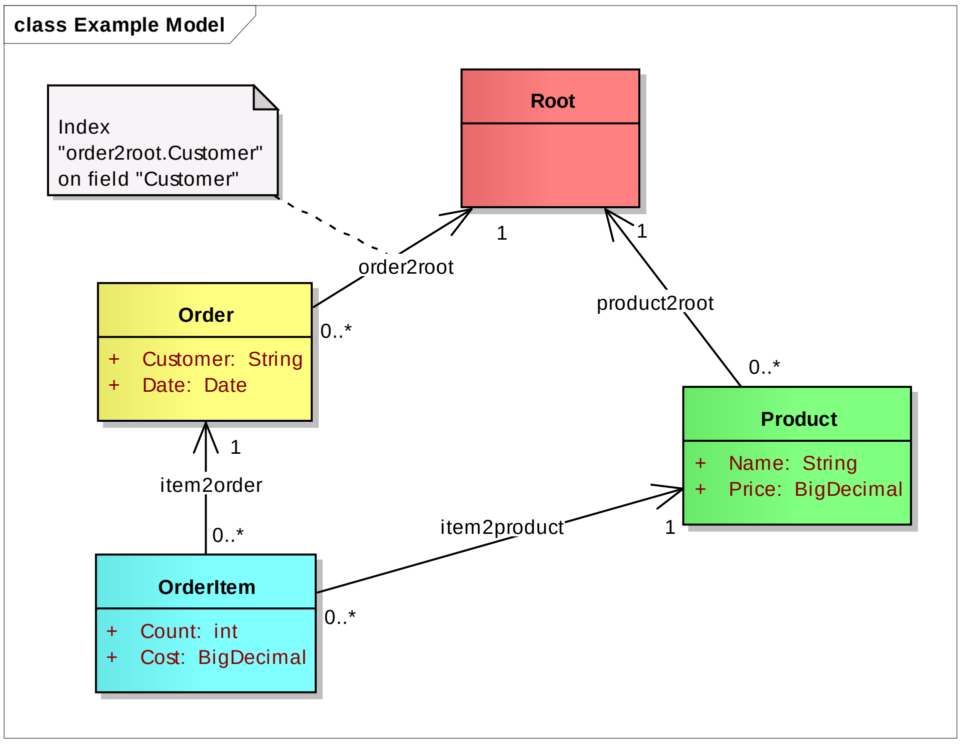
Functions API

Functions API is used for traversing over records.

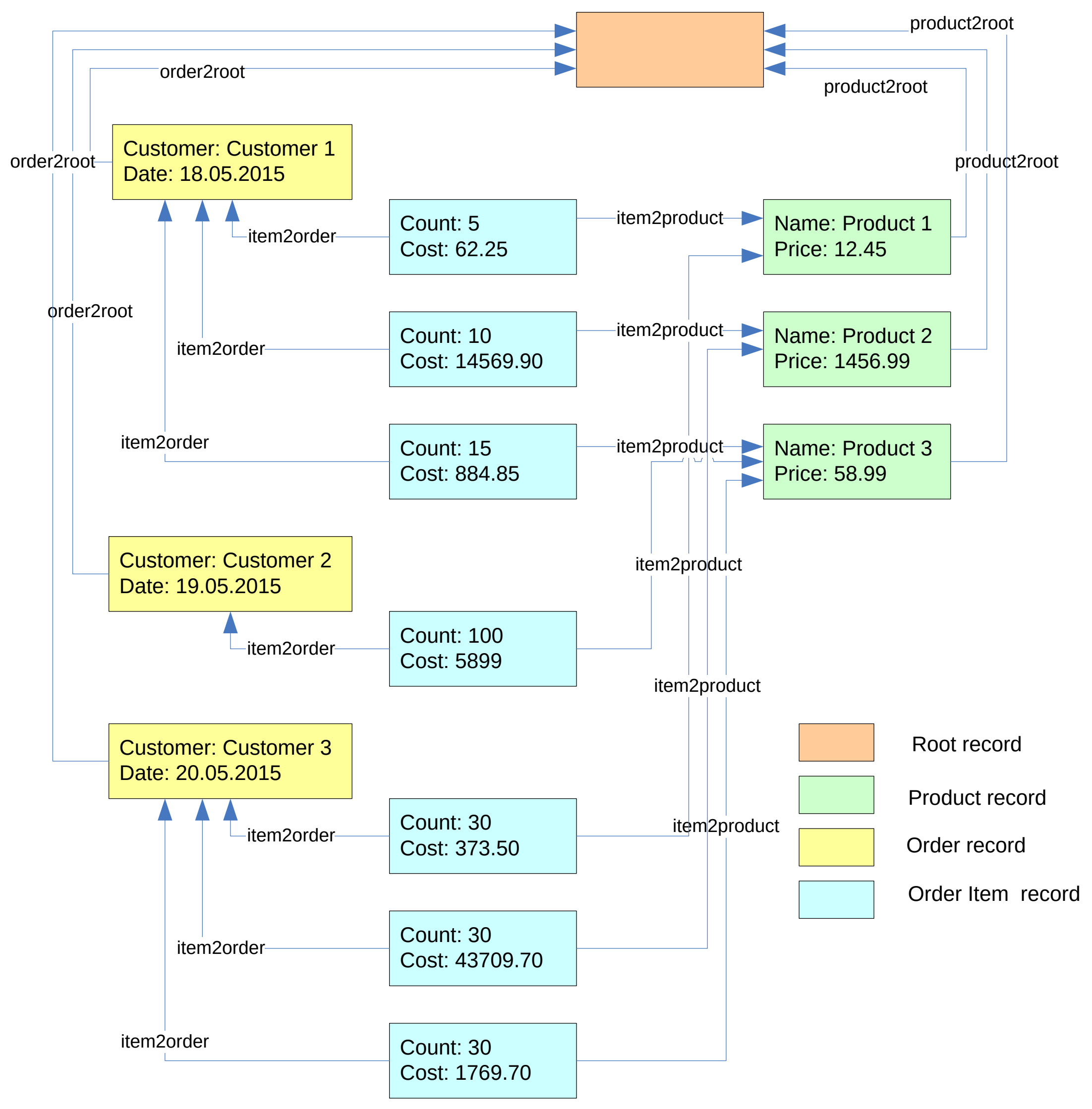
Idea of Functions API is following: your application creates tree of functions. Function is any object, which class is inherited from **com.vyhodb.F** class. Each function can perform particular action. After that, function tree is evaluated by invoking **eval()** method on root function and passing record object as it's parameter. Approach is similar to functional programming.

For more theory, please address "Developer Guide" and "Functions Reference" documents.

In this and next section we will use for sophisticate data model, illustrated on diagram below:



To create sample data we use **com.vyhodb.utils.DataGenerator#generate()** method. Method creates data according to data model above. Diagram below shows records, fields and links which are created by this method:



Assume that we need to print “Order Item” records, which refer to the “Product” record with specified name.

Firstly consider example, using usual approach with **for-each** iteration (main() method, import directives and constants are omitted because they are common for each example):

Copyright © 2015 Igor Vykhodtsev

```
package com.vyhodb.started.functions;

. . .
public class IntroForeach {

. . .

    private static void example(Space space) {
        String productName = "Product 2";

        // Generates sample data
        Record root = space.getRecord(0L);
        DataGenerator.generate(root);

        for (Record product : root.getChildren("product2root")) {
            if (productName.equals(product.getField("Name"))) {
                for (Record item : product.getChildren("item2product")) {
                    System.out.println(item);
                }
            }
        }
    }
}
```

Output:

```
{Cost=14569.90, Count=10} id=600
{Cost=43709.70, Count=30} id=688
```

As you can see, example() method is hard to read and understand because of inner for-each cycles.

Let's have a look how we can do the same things using Functions API:

```
package com.vyhodb.started.functions;

import static com.vyhodb.f.CommonFactory.*;
import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.f.PredicateFactory.*;

. . .
public class IntroFunctions {

. . .

    private static void example(Space space) {
        String productName = "Product 2";

        // Generates sample data
        Record root = space.getRecord(0L);
        DataGenerator.generate(root);

        // Builds function tree
        F printf =
        childrenIf("product2root", fieldsEqual("Name", productName),
            children("item2product",
                printCurrent()
            )
        );

        // Evaluates function
        printf.eval(root);
    }
}
```

Output:

```
{Cost=14569.90, Count=10} id=600
{Cost=43709.70, Count=30} id=688
```

Copyright © 2015 Igor Vykhodtsev

Pay attention to static import directives:

```
import static com.vyhodb.f.factories.CommonFactory.*;
import static com.vyhodb.f.factories.NavigationFactory.*;
import static com.vyhodb.f.factories.PredicateFactory.*;
```

We use them for hiding class names, which contains static methods for function creation (childrenIf(), children(), current(), etc). Because of this, our function building code looks like functional programming code.

Below there is one more example, where we calculate sales amount for particular product:

```
package com.vyhodb.started.functions;

import static com.vyhodb.f.AggregatesFactory.*;
import static com.vyhodb.f.CommonFactory.*;
import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.f.PredicateFactory.*;
import static com.vyhodb.f.RecordFactory.*;

. . .

public class Sum {

. . .

    private static void example(Space space) {
        String productName = "Product 2";

        // Generates sample data
        Record root = space.getRecord(0L);
        DataGenerator.generate(root);

        // Builds function tree
        F sumF =
            composite(
                childrenIf("product2root", fieldsEqual("Name", productName),
                    children("item2product",
                        sum(getField("Cost"))
                    )
                ),
                getSum()
            );

        // Evaluates function
        BigDecimal sum = (BigDecimal) sumF.eval(root);
        System.out.println(sum);
    }
}
```

Output:

```
58279.60
```

Copyright © 2015 Igor Vykhodtsev

ONM API

Object to Network model Mapping (ONM) API is a library for mapping between Java classes and vyhodb records.

ONM API provides the following functionalities to application developer:

1) **ONM Reading**. Traverses over vyhodb records and creates graph of Java objects according to traversal route.
2) **ONM Writing**. Updates records (creates new, changes or deletes existed) by Java object graph.
3) **ONM Cloning.** Traverses over Java object graph and creates copy of its sub-graph according to traversal route.

In this document we only cover ONM Reading and ONM Writing. For more information about ONM API, please address “Developer Guide” document.

Mapping between java class fields and record fields/links is specified by annotations or by external xml file. In this document we use annotations.

Java classes

Let’s create java classes, which satisfy to our domain model (described in **Functions API** section). Source code of these classes can be found in **com.vyhodb.started.onm** package. We need the following classes:

1) **Root**
2) **Order**
3) **OrderItem**
4) **Product**

Root

```
package com.vyhodb.started.onm;

import java.util.ArrayList;
import java.util.List;

import com.vyhodb.onm.Children;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Record;

@Record
public class Root {

    @Id
    private long id = 0;

    @Children(linkName="order2root")
    private ArrayList<Order> orders = new ArrayList<>();

    public long getId() {
        return id;
    }

    public List<Order> getOrders() {
        return orders;
    }
}
```

Order

```
package com.vyhodb.started.onm;
```

```java
import java.util.ArrayList;
import java.util.List;

import com.vyhodb.onm.Children;
import com.vyhodb.onm.Field;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Record;

@Record
public class Order {

    @Id
    private long id = -1;

    @Children(linkName="item2order")
    private ArrayList<OrderItem> items = new ArrayList<>();

    @Field(fieldName="Customer")
    private String customerName;

    public long getId() {
        return id;
    }

    public List<OrderItem> getItems() {
        return items;
    }

    public String getCustomerName() {
        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
}
```

OrderItem

```java
package com.vyhodb.started.onm;

import com.vyhodb.onm.Field;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Parent;
import com.vyhodb.onm.Record;

@Record
public class OrderItem {

    @Id
    private long id = -1;

    @Parent(linkName="item2order")
    private Order order;

    @Parent(linkName="item2product")
    private Product product;

    @Field(fieldName="Count")
    private int count;

    public long getId() {
        return id;
    }

    public Order getOrder() {
        return order;
    }
```

Copyright © 2015 Igor Vykhodtsev

```
    public void setOrder(Order order) {
        this.order = order;
    }

    public Product getProduct() {
        return product;
    }

    public void setProduct(Product product) {
        this.product = product;
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }
}
```

Product

```
package com.vyhodb.started.onm;

import com.vyhodb.onm.Field;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Record;

@Record
public class Product {

    @Id
    private long id = -1;

    @Field(fieldName="Name")
    private String name;

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

ONM Reading

Functions API is used for reading java object graph. Function is built and during evaluation it traverse over vyhodb records. For each visited record it creates appropriate java object, sets object fields and references between objects.

Information about Mapping is contained in **com.vyhodb.onm.Mapping** object, which should be explicitly created.

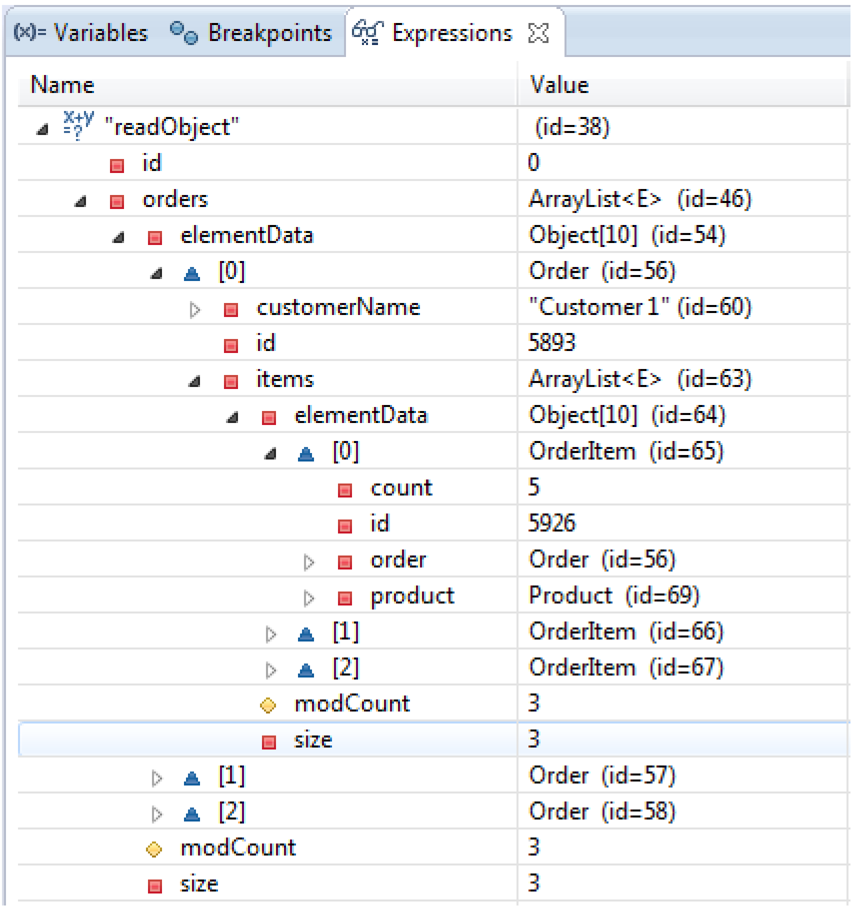
For traversing over vyhodb records, ONM Reading uses the same functions which were used in **Functions API** section (so called record navigation functions): children(), parent(), index(), etc.

The difference is that tree of navigation functions should be wrapped by **startRead()** function from **com.vyhodb.onm.OnmFactory** factory class. This function does some actions behind the scene and adds additional logic for handling visited records. This additional logic is java object creation and reference setting.

ONM Reading example:

```java
package com.vyhodb.started.onm;

import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.onm.OnmFactory.*;

import java.io.IOException;
import java.util.Properties;

import com.vyhodb.f.F;
import com.vyhodb.onm.Mapping;
import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;
import com.vyhodb.utils.DataGenerator;

public class OnmRead {

. . .

    private static void example(Space space) {
        // Generates sample data
        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        // Creates Mapping
        Mapping mapping = Mapping.newAnnotationMapping();

        // Builds ONM Reading function
        F readF =
            startRead(Root.class, mapping,
                children("order2root",
                    children("item2order",
                        parent("item2product")
                    )
                )
            );

        // Evaluates and reads object graph
        Root readObject = (Root) readF.eval(rootRecord);

        System.out.println(readObject);
    }
}
```

To see ONM Reading result it is recommended to run this class in Debug mode and inspect **readObject** variable. Below there is an Eclipse screenshot:



Name	Value
"readObject"	(id=38)
id	0
orders	ArrayList<E> (id=46)
elementData	Object[10] (id=54)
[0]	Order (id=56)
customerName	"Customer 1" (id=60)
id	5893
items	ArrayList<E> (id=63)
elementData	Object[10] (id=64)
[0]	OrderItem (id=65)
count	5
id	5926
order	Order (id=56)
product	Product (id=69)
[1]	OrderItem (id=66)
[2]	OrderItem (id=67)
modCount	3
size	3
[1]	Order (id=57)
[2]	Order (id=58)
modCount	3
size	3

ONM Writing

To update vyhodb records according to java object graph, the following method of **com.vyhodb.onm.Writer** class is used:

```
public static void write(Mapping mapping, Object rootObject, Space space)
```

This method traverses over objects in java graph, starting from **rootObject**. Method uses java reflection and **Mapping** object to traverse from one object to another, using fields, annotated by @Parent and @Children. For each visited java object, it does one of the following actions:

1) Creates new record
2) Changes record
3) Delete record

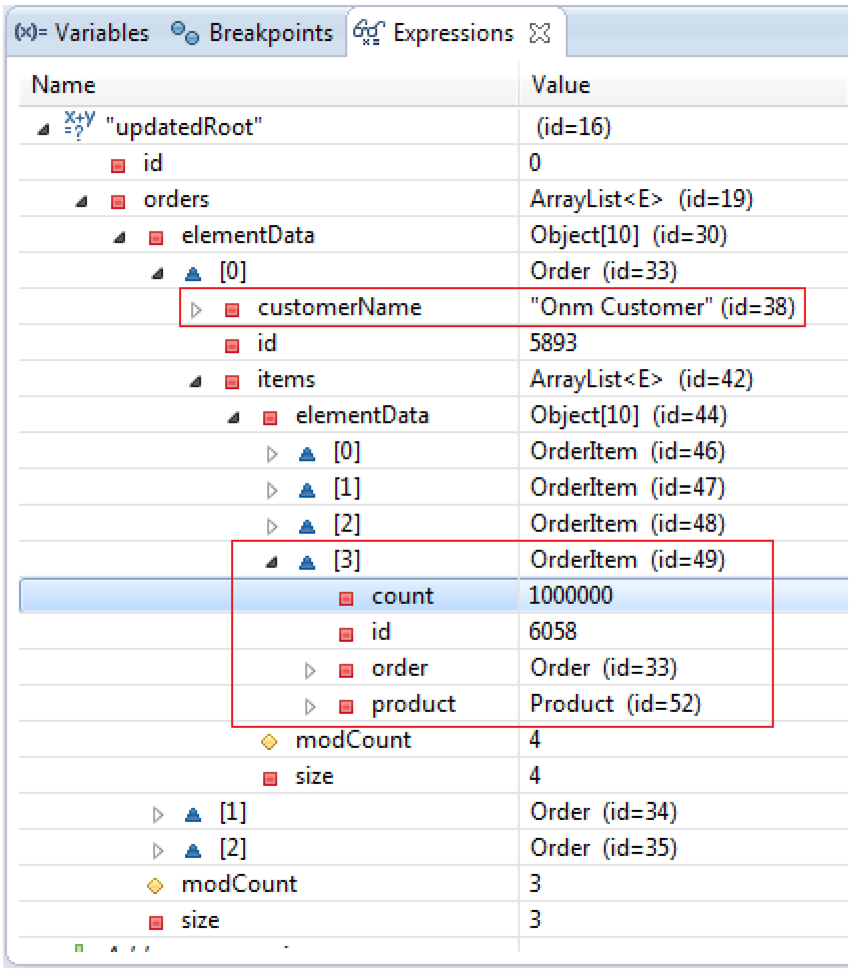
For more information about ONM Writing traversal algorithm and updating records see “ONM API” section in “Developer Guide” document.

In next example we read graph of java objects, modify it and save changed object graph. Changes consist of modifying “CustomerName” on first Order object and adding new OrderItem object. Finally we read java graph again to show that our changes have been successfully saved before.

```
package com.vyhodb.started.onm;

import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.onm.OnmFactory.*;

import java.io.IOException;
import java.util.Properties;

import com.vyhodb.f.F;
import com.vyhodb.onm.Mapping;
import com.vyhodb.onm.Writer;
import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;
import com.vyhodb.utils.DataGenerator;

public class OnmWrite {

. . .

    private static void example(Space space) {
        // Generates sample data
        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        // Builds ONM Read function
        Mapping mapping = Mapping.newAnnotationMapping();
        F readF =
            startRead(Root.class, mapping,
                children("order2root",
                    children("item2order",
                        parent("item2product")
                    )
                )
            );

        // Reads object graph
        Root readRoot = (Root) readF.eval(rootRecord);

        // Changes objects in read graph
        modify(readRoot);

        // ONM Writing
        Writer.write(mapping, readRoot, space);

        // Reads object graph again to illustrate ONM Writing result
        Root updatedRoot = (Root) readF.eval(rootRecord);
        System.out.println(updatedRoot);
    }

    private static void modify(Root root) {
        Order order = root.getOrders().get(0);
        Product product = order.getItems().get(0).getProduct();

        // Changes order's customer
        order.setCustomerName("Onm Customer");

        // Adds item
        OrderItem item = new OrderItem();
        item.setCount(1000000);
        item.setOrder(order);
        item.setProduct(product);
        order.getItems().add(item);
    }
}
```

Result can be seen by inspecting **updatedRoot** graph in debug mode. See Eclipse screenshot below:

Copyright © 2015 Igor Vykhodtsev



```
Name                                  Value
◢ "updatedRoot"                        (id=16)
      id                              0
    ◢ orders                          ArrayList<E>  (id=19)
        ◢ elementData                 Object[10]  (id=30)
            ◢ [0]                     Order  (id=33)
                ▷ customerName        "Onm Customer" (id=38)
                  id                  5893
                ◢ items               ArrayList<E>  (id=42)
                    ◢ elementData     Object[10]  (id=44)
                        ▷ [0]         OrderItem  (id=46)
                        ▷ [1]         OrderItem  (id=47)
                        ▷ [2]         OrderItem  (id=48)
                        ◢ [3]         OrderItem  (id=49)
                              count   1000000
                              id      6058
                            ▷ order   Order  (id=33)
                            ▷ product Product  (id=52)
                      modCount        4
                      size            4
            ▷ [1]                     Order  (id=34)
            ▷ [2]                     Order  (id=35)
          modCount                    3
          size                        3
```

Copyright © 2015 Igor Vykhodtsev