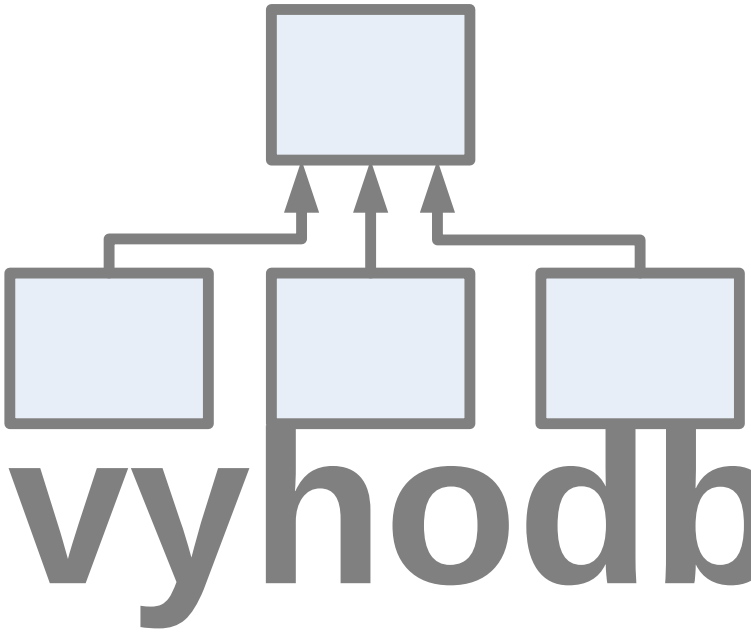


Developer's Guide

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.

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.

This document describes vyhodb APIs, available to software developer. Each chapter dedicated to particular API:

Chapter	Description
Server API	Start/stop vyhodb server in embedded mode. Transaction management (opening/completion).
Space API	Reading/modifying vyhodb data: records, fields, links
Indexes	Creation and using indexes
RSI API (Remote service invocation)	RSI Service implementation and remote invocation of their methods. RSI Services are java objects which "live" inside vyhodb server and have access to **Space API.**
Functions API	Traversing vyhodb records using functional programming approach.
ONM API	Mapping framework between Java classes and vyhodb records. Allows reading, writing and cloning.
Admin API	Administration tasks on running vyhodb and closed storage.

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.

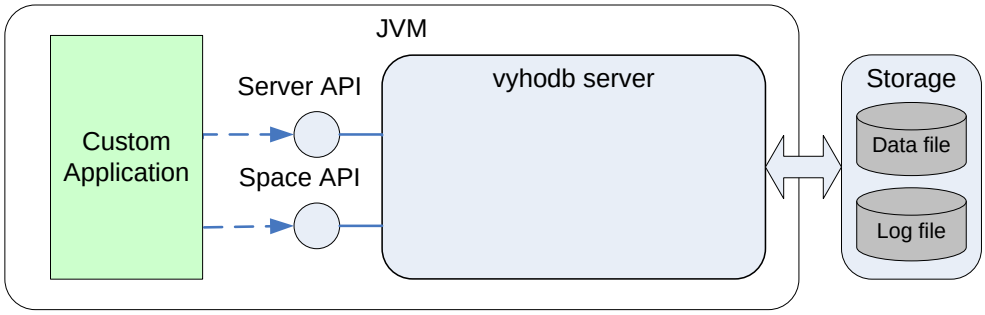
Running modes

vyhodb supports the following running modes:

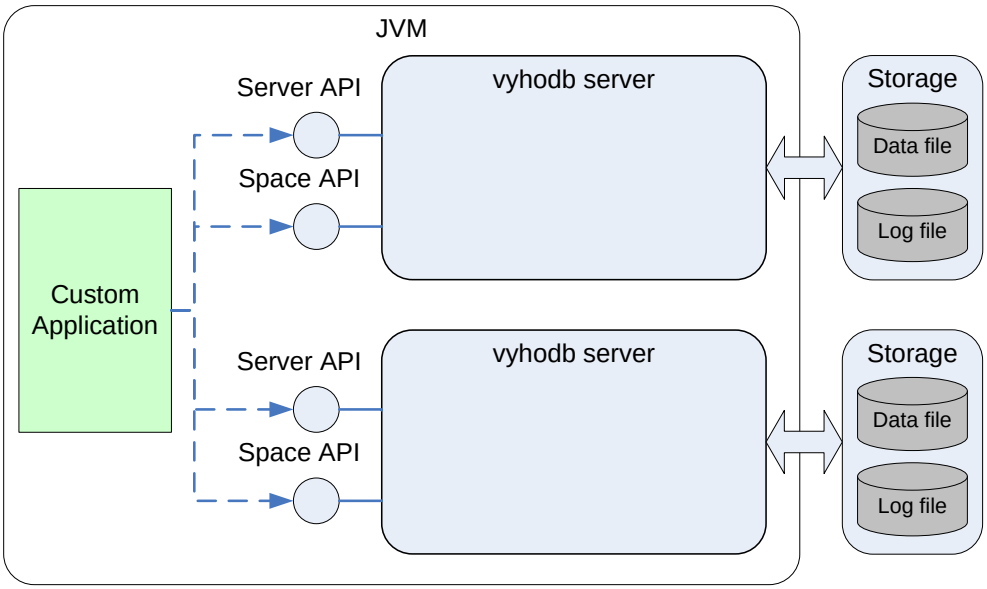
1) Embedded
2) Standalone
3) Local

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.



Custom application can start many vyhodb servers in its JVM.



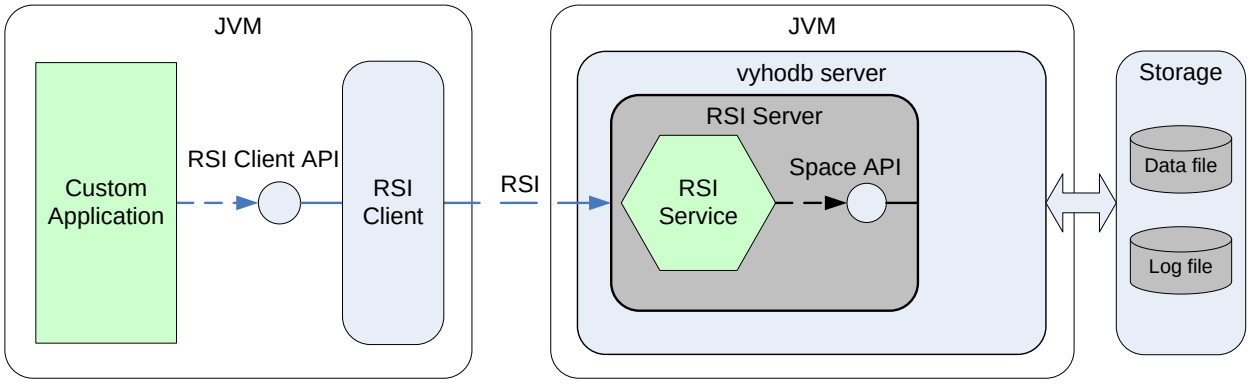
For starting vyhodb server in this mode, custom application's classpath should include all jar archives from lib directory.

If vyhodb server in embedded mode has active RSI Server component, then classpath should also include jar archives with RSI Service classes (they are placed in services directory). For more information about RSI Server component see "Administrator Guide" document.

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 (see section **RSI API (Remote service invocation)**).

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). Development of RSI Services is covered in **RSI Service implementation**.

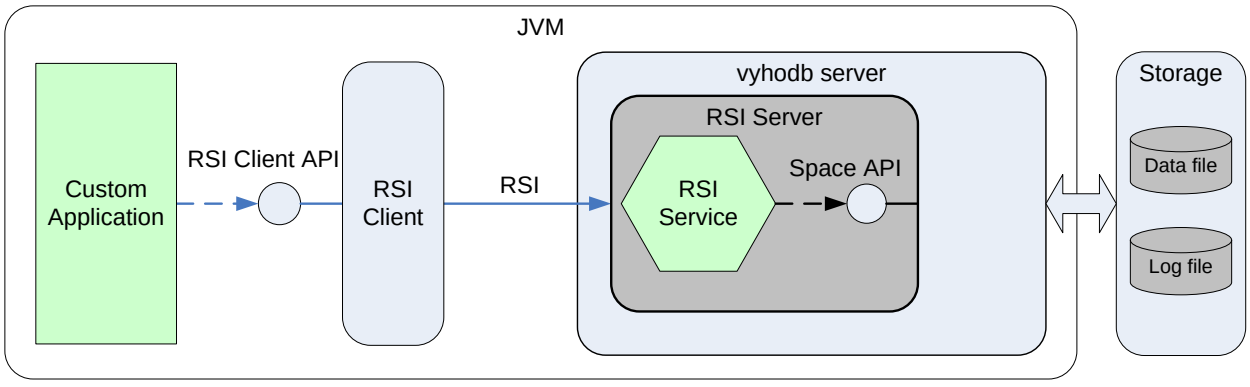
RSI Server component manages lifecycle of RSI Services. It opens new transaction for each remote method invocation and closes it after method completion.

Custom application uses RSI Client API for establishing connections to remote vyhodb server. RSI Client API is a part of RSI technology and is described in section **Using RSI Client API**.

Local

This mode is quite similar to Standalone mode with the exception that custom application and vyhodb server are running within the same JVM. They use RSI to communicate to each other; however no network connections are established.

To start vyhodb server, custom application creates RSI connection of special type (so called Local connection). Closing this connection leads to stopping vyhodb server. For more information see **Using RSI Client API.**



For starting vyhodb in Local mode, JVM's classpath should contain references to all jar files from lib directory as well as classes (or jar archives) of RSI Services.

Server API

Server API is intended for:

1) Starting vyhodb in embedded mode.
2) Transaction management and getting reference to **Space API.**

Server API's classes and interfaces can be found in package **com.vyhodb.server** (**vdb-core-0.9.0.jar**).

Note, that in order to start vyhodb in embedded mode, all jar files from **lib** directory should be included into classpath.

Example

Have a look at the example:

```
package com.vyhodb.guide.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
    }
}
```

Firstly, we prepare Properties object, by putting vyhodb configuration properties. These properties are the same as in vyhodb configuration file for standalone mode: **vdb.properties** (for more information about vyhodb configuring see "Administrator Guide" document).

In our case we specify only two properties: path to data file (**storage.data**) and path to log file (**storage.log**).

Secondly, we start vyhodb and receive reference to its object:

```
…
Server server = Server.start(props)
…
```

Thirdly, we call modifyTrx(), readTrx() methods, which read and write data (by opening and closing transactions).

Finally, we have to close vyhodb server explicitly by calling its close() method. This leads to free all resources used by vyhodb server. In out example, close() method is invoked automatically after leaving try(){} block.

Console 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.
Wed Sep 09 02:52:18 BRT 2015
[main] [INFO] vyhodb server closed.
```

In further examples we won't show vyhodb logging messages in console output. In current example, program output is done by invocation:

```
System.out.println(root.getField("Current Time"));
```

and its output is:

```
Thu Jul 02 10:44:41 BRT 2015
```

Transactions

vyhodb support two transaction types:

- Read
- Modify

Read transactions allow only reading date, whereas Modify can be used for both reading and modification.

For open new transactions, **com.vyhodb.server.Server** class provides the following methods:

```
    public abstract TrxSpace startReadTrx();
    public abstract TrxSpace startModifyTrx();
```

These methods returns TrxSpace object, which is an active transaction and **Space API** at the same time:

```
public interface TrxSpace extends Space {
    public void commit();
    public void rollback();
    public boolean isActive();
}
```

Transactions locks and isolation

Lock granularity in vyhodb is the whole storage (the whole Space of records in terms of vyhodb).

Vyhodb provides SERIALIZABLE isolation level.

Below there are rules, which describes how Read and Modify transactions are operating concurrently:

1) Many Read transactions are running in parallel.
2) Only one Modify transaction can be running at particular time.
3) Many Read transactions and one Modify transaction are running in parallel.
4) In case of Modify transaction commit, vyhodb waits for active Read transactions completion. It also suspends new Read transactions. Once Modify transaction has completed, vyhodb continues operating of new Read transactions.
5) Modify transactions are waiting their execution according to FIFO rule.

Deadlocks

Deadlocks are impossible if custom application follows approach: one active transaction per one Thread.

This approach is used by RSI Server, which manages transaction opening and closing during remote method invocations.

Closing Server

Object **com.vyhodb.server.Server** (which represents vyhodb server running in embedded mode) become subject of garbage collection only after closing. It continues consuming resources and keeping network connections until it explicitly closed.

That is why method close() must be invoked explicitly by custom application.

Vyhodb server automatically closes itself in case of stopping JVM, except situations when JVM is destroyed (for instance by sending kill -9 signal).

Space API

This chapter is dedicated to Space API which is used for reading and manipulating vyhodb data. Space API includes working with indexes, but they are covered in separated chapter **Indexes**.

Vyhodb uses quite simple data model, which consists of Records, Fields and Links. Records are containers for fields. Records are connected by named links forming child->parent relationships.

Vyhodb is schemaless. This means that records might have fields and links with any names. There are neither types for records nor rules about how records can be connected by links. Custom application can build data structure whichever it needs.

Space API classes and interfaces can be found in **com.vyhodb.space** package (**vdb-core-0.9.0.jar**).

Space

Space object (**com.vyhodb.space.Space**) is a vyhodb storage representation. It serves as a container for vyhodb records.

Reference to Space object (to TrxSpace actually) is retrieved at transaction opening time (see **com.vyhodb.server.Server** class).

Space interface definition:

```
package com.vyhodb.space;

public interface Space {
    public Record getRecord(long id);
    public Record newRecord();
    public boolean isReadOnly();
}
```

So, Space object allows to create new records and retrieve existed ones by id.

Records and fields

com.vyhodb.space.Record interface represents record stored in vyhodb.

Record belongs to one and only one Space and can be moved between Spaces. It doesn't support serialization and can't participate in RSI or any other RMI framework.

Each record has identifier of long type. This identifier is automatically assigned by vyhodb at record creation time and can't be changed.

During new storage creation, vyhodb automatically creates "root" record with 0 id. Root record is usually used as start point in record's traversing.

Record is a container for fields. Each field has name (String type) and value. List of supported classes which can be used as a field's values can be found in **Appendix A. Supported field value classes**. When assigned value isn't supported by vyhodb, current transaction is rolled back with throwing **com.vyhodb.server.TransactionRolledbackException** exception.

Record deletion leads to removing all links where deleted record is participated as a child or a parent. In current vyhodb implementation, space or deleted records isn't recycled, so deleted records occupy disk space.

Code below illustrates how to work with records:

```
package com.vyhodb.guide.space;

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

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

public class RecordExample {

    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)) {
            example(server);
        }
    }

    private static void example(Server server) {
        TrxSpace space = server.startModifyTrx();

        // Creates new record
        Record product = space.newRecord();

        // Sets fields
        product.setField("Name", "Product 1");
        product.setField("Price", new BigDecimal("12.45"));

        System.out.println(product.getId());
        System.out.println(product.getField("Name"));
        System.out.println(product.getField("Price"));

        // Deletes record
        product.delete();
        System.out.println(product.isDeleted());

        space.commit();
    }
}
```

Output:

```
5293
Product 1
12.45
true
```

Links

Links are used to connect records between each other. Because of this, vyhodb allows create and support sophisticated models, which represents real world domain areas.

Link creates two records, one of which is child, while the other is a parent. Bot records should belongs to the same Space. Each link has a name. Name is unique from child perspective and non-unique from parent (in other words, child record can have only one link to parent record with particular name).

Link creation, child record iterating

Links are created from child record side. The following methods on **com.vyhodb.space.Record** interface are used for link creation/modification and parent record retrieval:

```
    public Record setParent(String linkName, Record parent);

    public Record getParent(String linkName);
```

setParent() method is also used for changing link to another parent.

Child records (which refer to current parent record using the same link name) can be iterated from parent record. Methods below are intended for this:

```
    public Iterable<Record> getChildren(String linkName);

    public Iterable<Record> getChildren(String linkName, Order order);
```

Interface Iterable<Record> returns lazy iterator: records are read from disk/cache only at calling next() method.

Let's have a look at example, where we create records, links (from new records to root record) and, after that, iterate over child records (method main() as well as constants LOG, DATA are omitted because they are the same as in previous example):

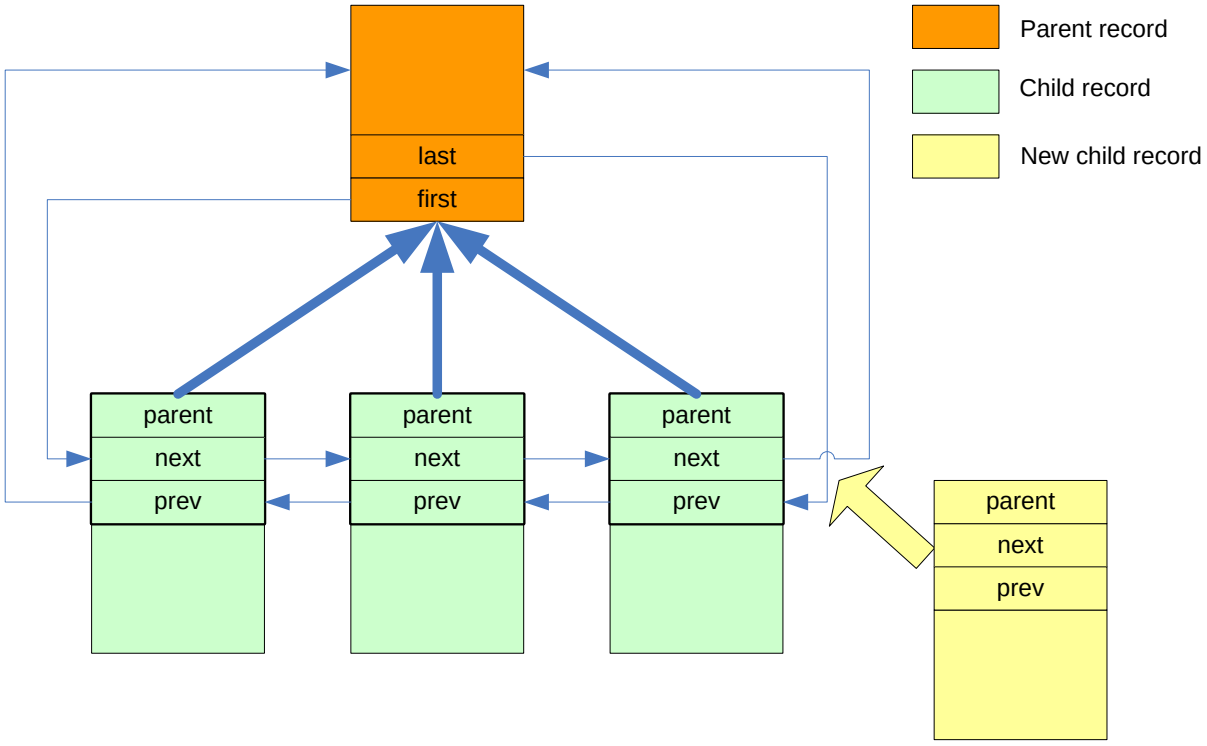
```
package com.vyhodb.guide.space;

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

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

public class LinksCreation {

. . .

    private static void example(Server server) {
        TrxSpace space = server.startModifyTrx();

        // Retrieves root record and removes all children for links "product2root"
        Record root = space.getRecord(0L);
        root.removeChildren("product2root");

        // Creates new records
        Record product1 = space.newRecord();
        Record product2 = space.newRecord();

        // Sets fields
        product1.setField("Name", "Product 1");
        product2.setField("Name", "Product 2");

        // Creates links to root record
        product1.setParent("product2root", root);
        product2.setParent("product2root", root);

        // Iterates over child records
        for (Record product : root.getChildren("product2root")) {
            System.out.println(product);
        }

        space.commit();
    }
}
```

Output:

```
{Name="Product 1"} id=534
{Name="Product 2"} id=545
```

Links internal structure

Links are implemented as a linked list, nodes of which are stored inside records.



Creation of new link to particular parent record creates new child record to the tail of linked list.

Method for retrieving child records **getChildren(String linkName, Order order)** supports additional parameter which specifies iterating direction: forward (from beginning to tail – **Order.ASC**) and backward (**Order.DESC**):

```
package com.vyhodb.space;

public enum Order {
    ASC,
    DESC
}
```

Forward direction returns records ordered by link creation time.

Link removal

There are two approaches for link removal.

In first approach, to remove existed link, you need to create a new one where parent record is null. You just clear parent record on particular child record:

```
…
product1.setParent("product2root", null);
…
```

Example below shows this approach:

```
package com.vyhodb.guide.space;
```

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

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

public class LinksRemoving1 {

    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)) {
            createLinks(server);
            removeLink(server);
        }
    }

    private static void createLinks(Server server) {
        TrxSpace space = server.startModifyTrx();

        // Retrieves root record and removes all children for links "product2root"
        Record root = space.getRecord(0L);
        root.removeChildren("product2root");

        // Creates new records
        Record product1 = space.newRecord();
        Record product2 = space.newRecord();

        // Sets fields
        product1.setField("Name", "Product 1");
        product2.setField("Name", "Product 2");

        // Creates links to root record
        product1.setParent("product2root", root);
        product2.setParent("product2root", root);

        space.commit();
    }

    private static void removeLink(Server server) {
        TrxSpace space = server.startModifyTrx();

        Record root = space.getRecord(0L);

        // Retrieves child record and removes link to root
        Record product1 = root.getChildFirst("product2root");
        product1.setParent("product2root", null);

        // Iterates over children and prints them
        for (Record product : root.getChildren("product2root")) {
            System.out.println(product);
        }

        space.commit();
    }
}
```

Output:

```
{Name="Product 2"} id=1101
```

In second approach, you retrieve child record iterator and use it's remove() method. See example below (methods main(), createLinks() as well as constants LOG, DATA are omitted because identical to previous example):

```
package com.vyhodb.guide.space;

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

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

public class LinksRemoving2 {

. . .

    private static void removeLink(Server server) {
        TrxSpace space = server.startModifyTrx();

        Record root = space.getRecord(0L);

        // Retrieves children iterator and removes link
        Iterator<Record> childIterator = root.getChildren("product2root").iterator();
        childIterator.next();
        childIterator.remove();

        // Iterates over children and prints them
        for (Record product : root.getChildren("product2root")) {
            System.out.println(product);
        }

        space.commit();
    }
}
```

Output:

```
{Name="Product 2"} id=1379
```

Sibling records iterating

Space API allows retrieve iterator for sibling records, which (with current one) refer to the same parent using the same link name.

It can be useful in case when child record's processing is split into many transactions. So application can store last processed child record and continue iterating from it in next transaction by using sibling iterator.

You can iterate over sibling records in forward (Order.ASC) and backward (Order.DESC) directions. The following methods return sibling iterators:

```
    public Iterable<Record> getSiblings(String linkName);

    public Iterable<Record> getSiblings(String linkName, Order order);
```

Iterable<Record> object returns lazy iterator: sibling records are read from disk/caches only at next() invocation.

Example of iterating over sibling record:

```
package com.vyhodb.guide.space;

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

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

public class Siblings {

. . .

    private static void example(Server server) {
        TrxSpace space = server.startModifyTrx();

        // Retrieves root record and removes all children for links "product2root"
        Record root = space.getRecord(0L);
        root.removeChildren("order2root");

        // Creates new records
        Record order1 = space.newRecord();
        Record order2 = space.newRecord();
        Record order3 = space.newRecord();

        // Sets fields
        order1.setField("Customer", "Customer 1");
        order2.setField("Customer", "Customer 2");
        order3.setField("Customer", "Customer 3");

        // Creates links to root record
        order1.setParent("order2root", root);
        order2.setParent("order2root", root);
        order3.setParent("order2root", root);

        // Iterates over sibling in ASC order
        System.out.println("Ascend order:");
        for (Record order : order2.getSiblings("order2root")) {
            System.out.println(order);
        }

        // Iterates over sibling in DESC order
        System.out.println("\nDescend order:");
        for (Record order : order2.getSiblings("order2root", Order.DESC)) {
            System.out.println(order);
        }

        space.commit();
    }
}
```

Output:

```
Ascend order:
{Customer="Customer 3"} id=1668

Descend order:
{Customer="Customer 1"} id=1646
```

Virtual fields

Each time when new link is created, new virtual field on child record is created as well. Virtual link is a read-only field of Long type. It name equals to added link name and value is parent record's identifier.

One of virtual field usage is create indexes on them. It allows creation of indexes based on links someway. Example:

```
package com.vyhodb.guide.space;

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

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

public class VirtualField {

. . .

    private static void example(Server server) {
        TrxSpace space = server.startModifyTrx();

        // Retrieves root record and removes existed children for links "product2root"
        Record root = space.getRecord(0L);
        root.removeChildren("product2root");

        // Creates new records
        Record product1 = space.newRecord();
        Record product2 = space.newRecord();

        // Sets fields
        product1.setField("Name", "Product 1");
        product2.setField("Name", "Product 2");

        // Creates links to root record
        product1.setParent("product2root", root);
        product2.setParent("product2root", root);

        // Prints virtual fields' values.
        // In current example they print root record's id: 0.
        System.out.println(product1.getField("product2root"));
        System.out.println(product2.getField("product2root"));

        space.commit();
    }
}
```

Dictionary

By default, names of fields, links, and indexes are stored inside each record. It leads to increase disk space usage and decrease in database performance.

For reducing used disk space and increasing overall performance it is recommended to use Dictionary mechanism.

Dictionary allows specify map between strings (which are names of fields, links, and indexes) and Integer codes in separate java properties file. When such file is created and configured, vyhodb starts writing Integer codes instead of corresponding Strings.

Dictionary file example:

```
Name = 1
Price = 2
product2root = 3
```

```
order2root = 4
```

Path to dictionary file is specified by using property **storage.dictionary**. For information about vyhodb configuring see document "Administrator Guide".

Fragment of vyhodb configuration file: **vdb.properties** (for standalone mode):

```
# Path to log file
storage.log = storage/vyhodb.log

# Dictionary property file
storage.dictionary = storage/dictionary.properties

# Durable flag
#storage.durable = false
```

Indexes

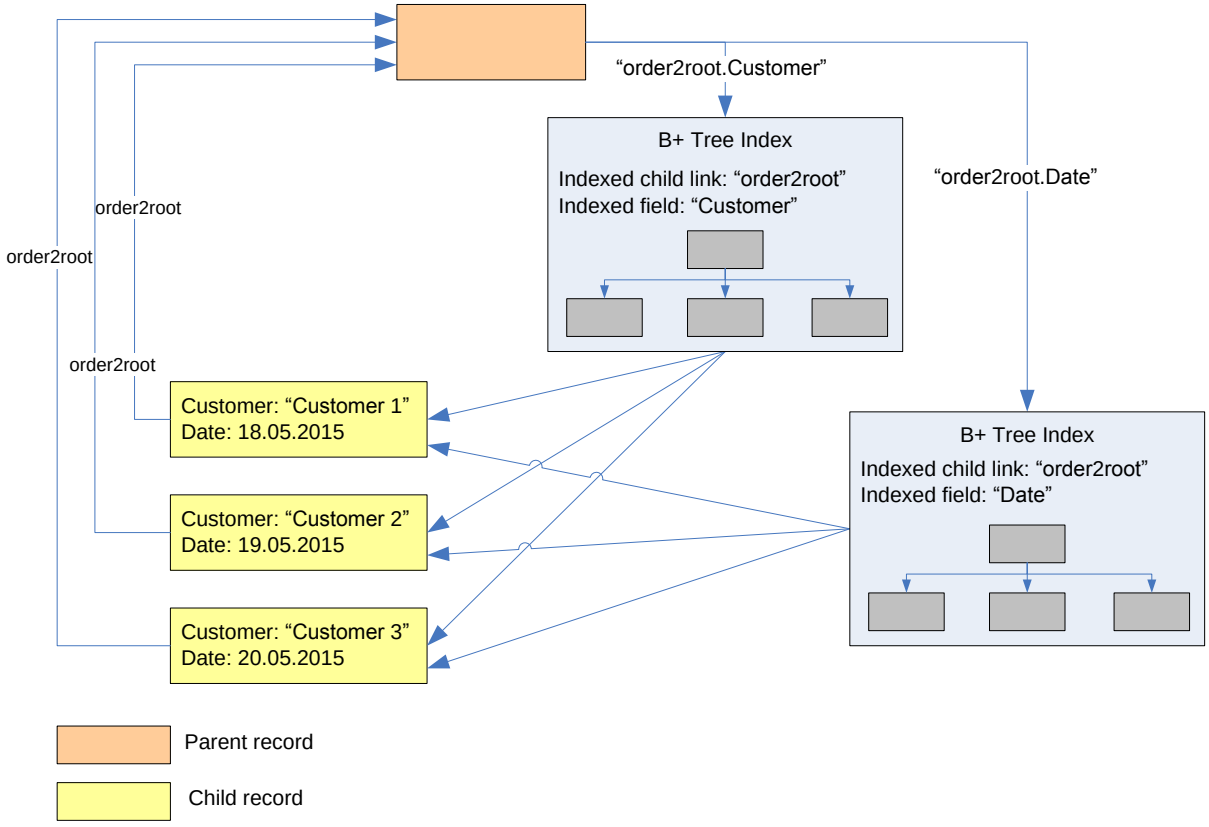
Search records by their field values is a very often task. Search by iterating over all huge count of children records might lead to performance decrease. Indexes facilitate this task by speeding up search of child records.

Indexes are part of Space API.

Index is created for particular "parent" record and indexes fields of its child records. "Parent" record can have as many indexes as it needs on any child record's fields. Each index has name which should be unique within "parent" record.

Internally, index is a B+Tree structure, which is hidden from application developer. Nodes and leafs of B+Tree stores field values of child records in sorted order (ascending), and references to child records themselves. It allows fast searching and sorting of child records.

Diagram below illustrate indexes:



On this diagram you can see one parent and three child records. Child records have "order2root" links to parent record. Two indexes are created on parent record for indexing its children: **"order2root.Customer"** and **"order2root.Date**". First one indexes **"Customer"** field, the second one indexes **"Date"** field.

Index is automatically updated each time, when:

1) New link is created to "parent" record.
2) Existed link is removed.
3) Indexed field value is changed on child record.

Depending on count of indexed fields, indexes are considered as:

- Simple – one indexed field
- Composite - two and more indexed fields

All methods for working with indexes (creation, removal, search) are defined on **com.vyhodb.space.Record** interface and will be covered further in this chapter.

Index creation

To create new index application have to specify:

1) Index name (should be unique within record)
2) Link name of indexed child records.
3) Index uniqueness (whether it is allowed or not to have child records with duplicate field values).
4) List of indexed field names.

In example below, we do the following actions:

1) Create records
2) Create index with name "order2root.Customer" on field "Customer"
3) Search child records using index

```
package com.vyhodb.guide.index;

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.Criterion;
import com.vyhodb.space.IndexDescriptor;
import com.vyhodb.space.IndexedField;
import com.vyhodb.space.Nullable;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;
import com.vyhodb.space.Unique;
import com.vyhodb.space.criteria.Equal;

public class IndexCreation {
    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)) {
            TrxSpace space = server.startModifyTrx();
            example(space);
            space.rollback();
        }
    }

    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);
    }
}
```

Example is quite large, so let's focus on createIndex() method, which actually creates index:

```
    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);
    }
```

This method:

1) Creates **IndexedField** object, which describes indexed field.
2) Creates **IndexDescriptor** object, which describes the whole index.
3) Creates index, using **com.vyhodb.space.Record#createIndex()** method.

IndexedField and **IndexDescriptor** objects are not connected to particular record; they just describe index metadata and can be reused to create many indexes with the same parameters.

IndexedField

Objects of **com.vyhodb.space.IndexedField** class describe indexed fields. They contains:

1) Name of indexed field.
2) Class of indexed field's values.
3) Nullable – whether field can be empty or not.

Once index created, it puts constraint on all indexed child records. For instance, if added/changed child record contains field value with a class differs from specified in IndexedField, then transaction will be rolled back and **com.vyhodb.server.TransactionRolledbackException** will be thrown.

List of supported classes which objects can be used as a indexed field values can be found in **Appendix B. Supported indexed field value classes**.

Enumeration **com.vyhodb.space.index.Nullable** specifies whether indexed fields can be empty (with null value) or not:

```
package com.vyhodb.space.index;

public enum Nullable {
    NULL,
    NOT_NULL
}
```

For instance, if empty values are not supported (Nullable.NOT_NULL) and a new link is creating from child record which doesn't have indexed field, then transaction will be rolled back and **com.vyhodb.server.TransactionRolledbackException** exception is thrown.

IndexDescriptor

Objects of **com.vyhodb.space.IndexDescriptor** class describes index. To create this object application should specify:

1) Index name.
2) Link name of indexed child records.
3) Index uniqueness (whether it is allowed or not to have child records with duplicate field values).
4) List of indexed field names.

To define whether index can support unique/duplicate values the following enumeration is used:

```
package com.vyhodb.space.index;

public enum Unique {
    DUPLICATE,
    UNIQUE
}
```

If adding new child record or field modification violates uniqueness, then transaction will be rolled back and **com.vyhodb.server.TransactionRolledbackException** will be thrown.

Search

For searching child records using index, Record interface has the following methods:

```
public Iterable<Record> searchChildren(String indexName, Criterion criterion);

public Iterable<Record> searchChildren(String indexName, Criterion criterion, Order order);
```

Object Iterable<Record> returns lazy iterator: when iterating over search result records, B+Tree nodes as well as records itself are read from disk/cache only at next() method invocations.

Returned child records are sorted according to indexed field(s) values. Order of iteration is ascending by default (Order.ASC), but can be changed.

Search criterion is specified by **com.vyhodb.space.Criterion** interface**.** There are many implementations of Criterion interface for different search types in **com.vyhodb.space.criteria** package.

Table below shoes criteria classes which can be used to search in a **simple index**:

Criteria class (package name is omitted)	Description
All	Returns all records, sorted by indexed field.
Equal	Returns records which indexed field values are equal to key specified in criterion.
Null	Returns records with empty indexed field.
NotNull	Returns records which indexed field is not empty.
In	Returns records which indexed field values are contained in collection specified in criterion.
Less	Returns records which indexed field < criterion key.
LessEqual	Returns records which indexed field <= criterion key.
More	Returns records which indexed field > criterion key.
MoreEqual	Returns records which indexed field >= criterion key.
Between	Returns records which indexed field value is within specified range.
BetweenExclusive	Returns records which indexed field value is within specified range (excludes range boundaries).
StartsWith	Returns records which indexed field values starts with specified String prefix. Criterion can be used only for String fields.

Table below shows criteria for **composite index**:

Criteria class (package name is omitted)	Description
All	Returns all records sorted by indexed fields.

EqualComposite	Returns records which indexed fields are equal to values specified in criterion.

Sorting, minimal, maximum

Indexed records inside index are sorted by their field values. So index can be used to retrieve record with minimal and maximum indexed field value. The following methods are defined in Record interface for these purposes:

```
public Record searchMaxChild(String indexName);

public Record searchMinChild(String indexName);
```

In addition to it, custom application can iterate over child records, sorted by their indexed field values. **com.vyhodb.space.criteria.All** criterion class is used for this. Order enumeration paremeter defines sorting direction.

Index deletion

Index is deleted by method:

```
public void deleteIndex(String indexName);
```

Index deletion doesn't delete indexed child records. Parent record deletion lead to deletion of all its indexes.

Composite index

Composite indexes (index two or more fields) are created and used when there is a need for searching child records by many field values.

There is an example below of composite index creation and searching child records using it (method main() as well as constants LOG, DATA are omitted, because they identical to previous example):

```
package com.vyhodb.guide.index;

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

import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Criterion;
import com.vyhodb.space.IndexDescriptor;
import com.vyhodb.space.IndexedField;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;
import com.vyhodb.space.criteria.EqualComposite;

public class IndexComposite {

. . .

    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 composite index
        createCompositeIndex(root);

        // Creates search criteria
        HashMap<String, Comparable> fields = new HashMap<>();
        fields.put("Customer", "Customer 3");
        fields.put("Date", new Date("05/20/2015"));
        Criterion criterion = new EqualComposite(fields);

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

    private static void createCompositeIndex(Record record) {
        IndexedField customerField = new IndexedField(
                "Customer",
                String.class
        );

        IndexedField dateField = new IndexedField(
                "Date",
                Date.class
        );

        IndexDescriptor indexDescriptor = new IndexDescriptor(
                "order2root.CustomerDate",
                "order2root",
                customerField,
                dateField
        );

        record.createIndex(indexDescriptor);
    }
}
```

Output:

```
{Customer="Customer 3", Date="Wed May 20 00:00:00 BRT 2015"} id=2074
```

Composite indexes can include **Virtual fields** as well.

Only two search criterion classes are available for searching in composite indexes:

- All
- EqualComposite

EqualComposite criterion

Let's return to the code fragment from previous example, where we create EqualComposite criterion:

```
```

```
        // Creates search criteria
        HashMap<String, Comparable> fields = new HashMap<>();
        fields.put("Customer", "Customer 3");
        fields.put("Date", new Date("05/20/2015"));
        Criterion criterion = new EqualComposite(fields);
```

EqualComposite criterion is used for searching records by many indexed fields. Key field names and their values are specified as Map<String, Comparable> object, where key – is field name, value –field value.

EqualComposite criterion can be used for searching by all indexed fields or by only some of them (**partial search**). Example below shows how to construct criterion for searching by "Customer" field only:

```
        // Creates search criteria
        HashMap<String, Comparable> fields = new HashMap<>();
        fields.put("Customer", "Customer 3");
        Criterion criterion = new EqualComposite(fields);

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

Main rule of **partial search** is gap absence. We will illustrate this on example.

Consider we have composite index which is based on fields: A, B, C, D. **Order of fields specified in IndexDescriptor is vital.**

Then the following field combinations in EqualComposite are possible:

- A, B, C, D
- A, B, C
- A, B
- A

However the following combinations are wrong (because first fields are absent or there are gaps between fields):

- B, C, D
- A, B, D
- C
- B
- A, D

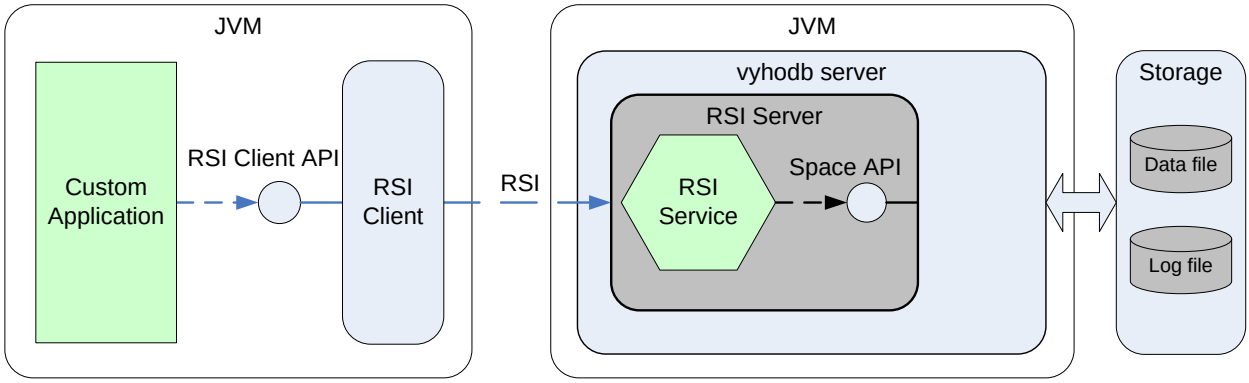
An attempt to use error combination of search fields in EqualComposite criterion leads to rolling back of current transaction and throwing **com.vyhodb.server.TransactionRolledbackException** exception.

RSI API (Remote service invocation)

RSI API is a technology for remote invocation of service methods. Services implements business logic and are executed within RSI Server component.

By default RSI Server component is switched off in embedded mode and switched on in standalone mode. RSI Server component's parameters as well as its availability is configured by Property object or **vdb.properties** file.

Custom application invokes service methods by using RSI Client API.



RSI Client API classes can be found in package **com.vyhodb.rsi** (packed in **vdb-rsi-0.9.0.jar** archive). Note, that RSI Client API doesn't depend on other vyhodb APIs. This allows developing custom application in service-oriented way, without any connection to vyhodb data model (only RSI Services are aware about vyhodb data model and APIs). For more information about jar dependencies see **Appendix C. Jar's dependencies**.

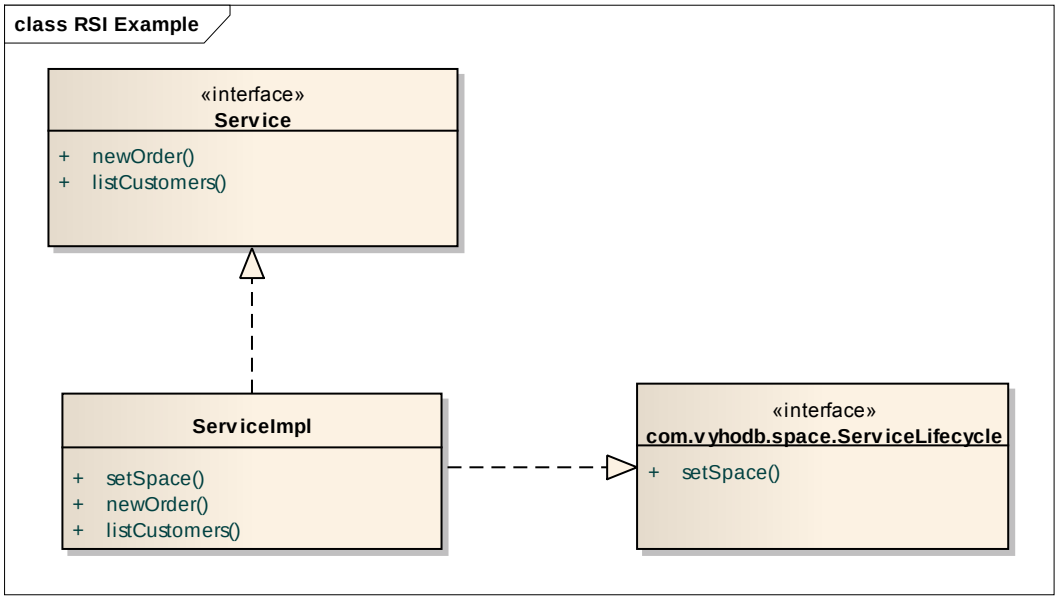
RSI Service implementation

To create RSI Service the following steps should be done:

1) Create service contract (java interface actually).
2) Implement service class. This class should implement methods, defined in service contract.
3) Pack RSI Service classes (service contract, implementation class, other referenced classes) into jar archive and deploy it on vyhodb server.

Further in this section we show how to implement RSI Service in details. Examples are the same as in "Getting Started" document; however we are considering them with more details.

Diagram below shows classes of our RSI Service:



Service 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();
}
```

@Implementation annotation defines class name of service implementation. Implementation class name is sent to RSI Server side with invocation request and is used by RSI Server to create new object which is used in turn to invoke method. In our example, service implementation class name is **com.vyhodb.started.rsi.ServiceImpl.**

Each contract's method is annotated by either @Read or @Modify. These annotations specify transaction type opened by RSI Server before method invocation. By default (if annotation is absent) @Read is used.

Service implementation class

```
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();
    }

    @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;
    }
}
```

Every service implementation class must implement **com.vyhodb.space.ServiceLifecycle** interface:

```
public interface ServiceLifecycle {

    /**
     * Initializes service implementation object.
     * <p>Service implementation object must save passed transaction space
     * somewhere in its field and use it in subsequent service method invocation.
     *
     * @param space transaction space
     */
    public void setSpace(Space space);
}
```

setSpace() method is used for passing **Space API** into service object. Service implementation should store it inside and use in subsequent method invocation.

Service implementation class must have public no-argument constructor.

RSI Server performs the following actions when new RSI request is arrived:

1) Creates object of service class, specified in RSI request.
2) Opens transaction. Transaction type (Read/Modify) is specified in RSI request and is read from service contract on client side (by RSI Client API actually).
3) Passes Space API object into service implementation object by using setSpace() method.
4) Invokes required method (specified in RSI request).
5) Commits transaction and sends method invocation result back to RSI Client API.

If invoked method throws any exception, than RSI Server rollback current transaction and sends exception stack trace back to RSI Client API (where it is wrapped by **com.vyhodb.rsi.RSIServerException** and throws to custom application).

Deployment

Deployment is quite simple: just put jar files with RSI Services into **services** directory and restart vyhodb server.

RSI Server doesn't support "hot" deployment, so it has to be restarted after copying new classes (jar files) into **services**.

RSI Server doesn't create "special" class loaders for loading RSI Service classes. It uses the same class loader which had been used to load vyhodb system classes.

In case of Local mode, RSI Service classes (jar files) should be included into JVM classpath or be available for custom application's class loader.

Using RSI Client API

Example below shows client application and how it uses RSI Client API for opening new connection with RSI Server and invoking methods of RSI Service, created in previous section:

```
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);
        }
    }
}
```

Pretty simple, isn't it?

In this example we create **com.vyhodb.rsi.Connection** object using ConnectionFactory. We pass **URL connection string** into ConnectionFactory#newConnection() method in order to create new Connection object. **URL connection string** describes connection type, RSI Server location and other configuration parameters.

com.vyhodb.rsi.Connection object represents logical connection to RSI Server. It is logical, because for physical network connection might not establish at all for particular type, while the others establishes many TCP connections.

com.vyhodb.rsi.Connection object is intended to create proxy objects for RSI Services (**service** variable in our example). Those proxy objects create invocation requests and send them through parent connection.

Table below shows supported connection types. For more information about connection types and their configuring see "RSI Client API" section in "Administrator Guide" document.

Connection type	Description	URL examples

Local	Starts vyhodb server in Local mode.
	
	URL starts from "local:" prefix followed by vyhodb configuration file URL.
		`local:file:c:/vdb-0.9.0/vdb.properties`
		
		`local:http://195.168.10.1/vdb.properties`

TCP Single	Establishes TCP connection to RSI Server. URL starts with "tcp:" prefix, followed by RSI Server's host name and port.	`tcp://localhost:47777/`
TCP Pooled	Creates pool of TCP connections to specified RSI Server. URL is similar to previous one. Differences are in URL parameters. For pooled connection "pool=true". Additional pool parameters can be specifies as well: poolSize, poolTTL, debug.	`tcp://localhost:47777/?pool=true&poolSize=10&poolTTL=360000`

TCP Balancer	Creates many connection pools to different RSI Servers for load balancing and distributing @Read invocations.
	
	URL starts with "balancer:" prefix followed by balancer configuration file URL.
	
	Balancer configuration file defines RSI Server locations as well as the other balancer parameters.
		`balancer:file:c:/vdb-0.9.0/balancer.properties`
		
		`balancer:ftp://195.68.10.1/balancer.properties`

Threads synchronization

All classes of RSI Client API are thread-safe (including service's proxy objects): they are synchronized and can be shared by many threads.

From the other hand, physical TCP connection to RSI Server can handle only one invocation per time. Because of this, threads synchronize their access to physical connection. As a result of this, count of simultaneously running invocations over RSI Connection object is equal to count of underlying physical connections.

In case of Local RSI Connection (in local running mode), physical connections are not created, and there is no synchronization between threads for access to underlying connections. All threads invokes RSI Service methods without delays, however they are synchronized inside RSI Server as part of transaction synchronization and locking.

Serialization requirements for classes

RSI uses **kryo** framework (https://github.com/EsotericSoftware/kryo) for methods' arguments/result serialization.

Classes participated in RSI communication (as method arguments and/or result) must have public no-argument constructor.

To speed up serialization/deserialization, application developer and implement **KryoSerializable** on classes participated in RSI.

Versioning

To prevent getting out of sync between versions of service contract on client application and service implementation class on RSI Server, application developer can use versioning mechanism.

For using it, both service contract and service implementation classes should be annotated by **@Version** with specifying version as string.

When @Version is presented on service contract, RSI Service API includes this version into RSI request. RSI Server validates each RSI request and if it has version information RSI Server compares it with version on implementation class. If they are differ, then remote invocation is interrupted and custom application receives **com.vyhodb.rsi.RsiServerException**.

Examples below show RSI Service from previous examples with @Version annotations.

Versioned service contract:

```
package com.vyhodb.guide.rsi.version;

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

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

@Version(version="1.42")
@Implementation(className="com.vyhodb.guide.rsi.version.ServiceImpl")
public interface Service {

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

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

Versioned service implementation class:

```
package com.vyhodb.guide.rsi.version;

import java.util.ArrayList;
import java.util.Collection;
```

```
import java.util.Date;

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

@Version(version="1.42")
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();
    }

    @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;
    }
}
```

Functions API

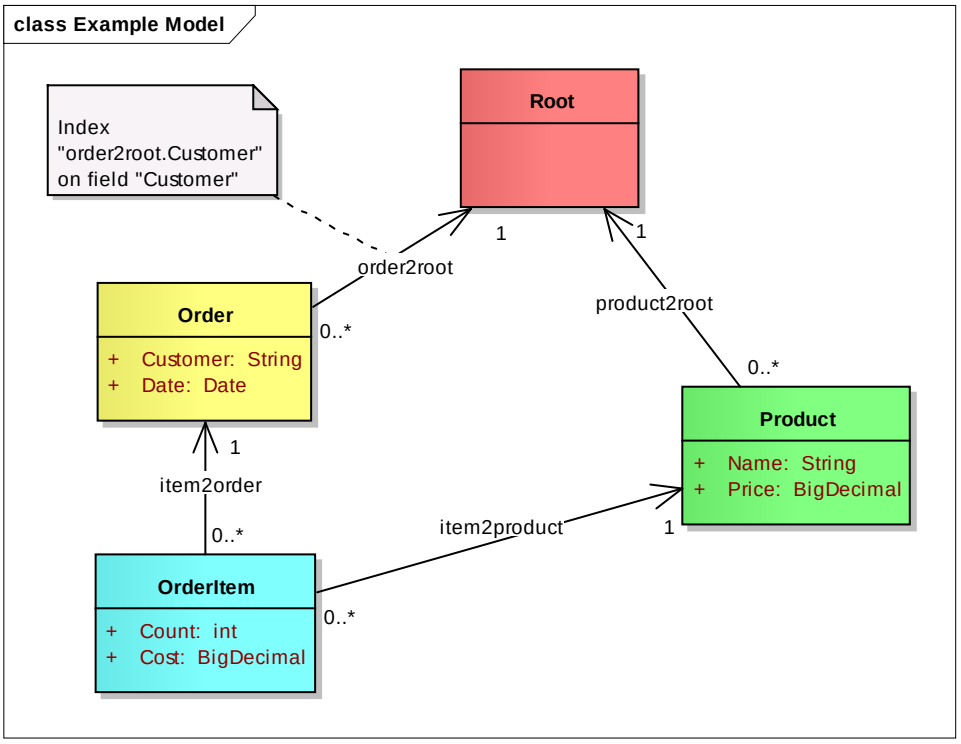
"Functions API" (or simply - functions) is an approach and framework which simplifies writing code for traversing over vyhodb records.

This chapter is an introduction to Functions API. For more information about functions it is recommended to address "Functions Reference" document and API source code.

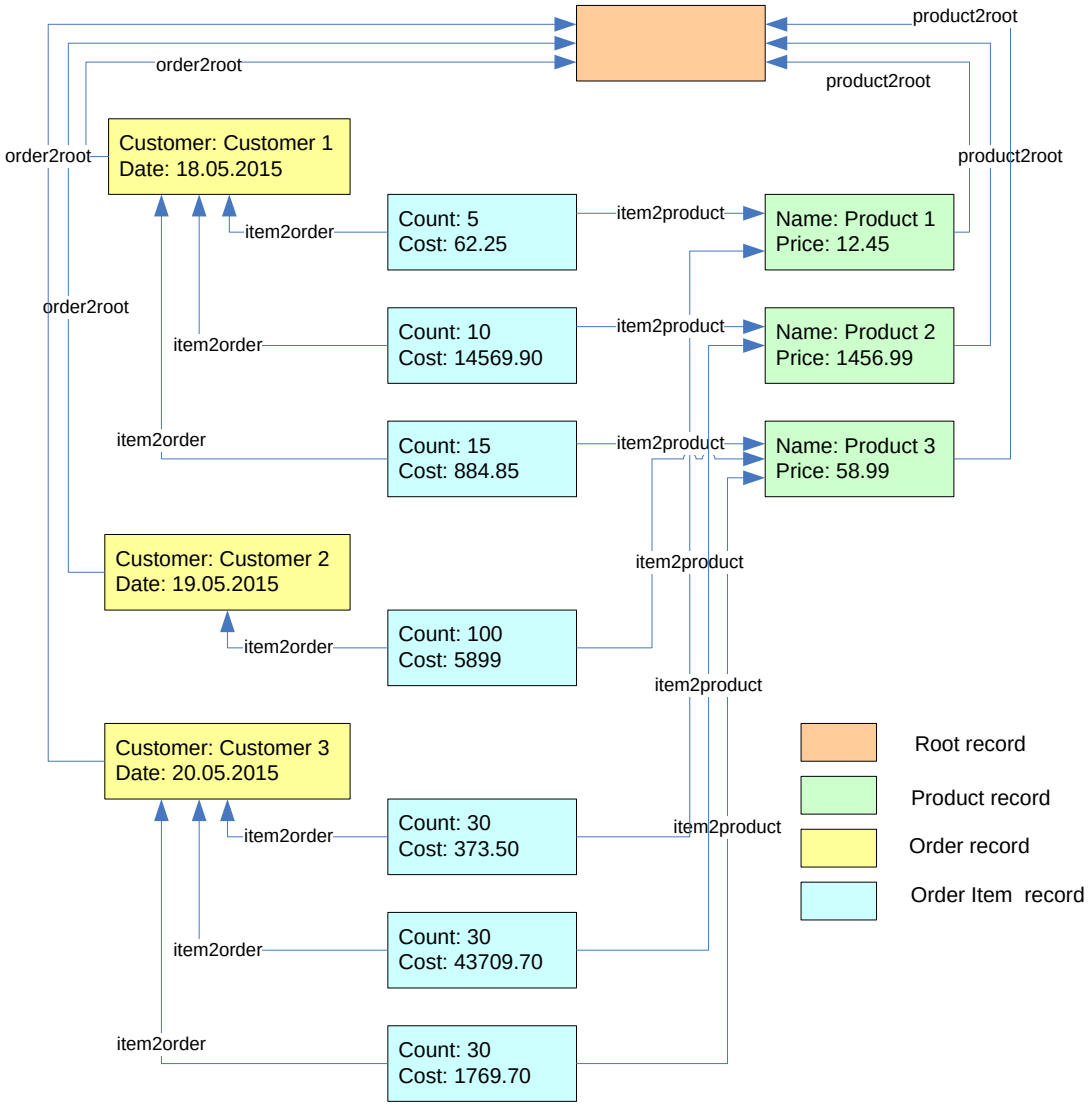
Functions API classes and interfaces are located in **com.vyhodb.f** package(**vdb-core-0.9.0.jar** system archive).

Example's data model

In current and next chapters we are going to use the following data model for our examples:



Method **com.vyhodb.utils.DataGenerator#generate()** creates sample data according to this model. We will use this method in each code example to create sample data. Diagram below shows records, fields and links which are created by **com.vyhodb.utils.DataGenerator#generate():**



Introduction

Example

We are going to show how to implement some algorithm using both traditional approach and Functions API.

Example below prints all OrderItem records, which parent Order record has particular Customer name. We use traditional approach with for-each cycle:

```
package com.vyhodb.guide.functions;

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

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 IntroForeach {

    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)) {
            TrxSpace space = server.startModifyTrx();
            example(space);
            space.rollback();
        }
    }

    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=2140
{Cost=43709.70, Count=30} id=2228
```

The same algorithm but using Functions API (method main(), constants LOG, DATA and some import directives are omitted):

```
package com.vyhodb.guide.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);
    }
}
```

It is much simpler and better for reading, isn't?

Output:

```
{Cost=14569.90, Count=10} id=2140
{Cost=43709.70, Count=30} id=2228
```

Function building

Let's examine the following fragment of code where we assembling functions' tree:

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

Functions' tree is made up of function objects, which classes are inherited from base function class **com.vyhodb.f.F**:

```
public abstract class F implements Serializable {
    /**
     * Starts function tree evaluation.
     * <p>
     * Method creates new context and invokes
     * {@linkplain #evalTree(Object, Map)}
     *
     * @param current
     *            current object
     * @return evaluation result
     */
    public final Object eval(Object current) {
        return evalTree(current, new HashMap<String, Object>());
    }
    /**
     * Evaluates function as part of function tree evaluation.
     *
     * @param current
     *            current object
     * @param context
     *            evaluation context
     * @return evaluation result
     */
    public abstract Object evalTree(Object current, Map<String, Object> context);
}
```

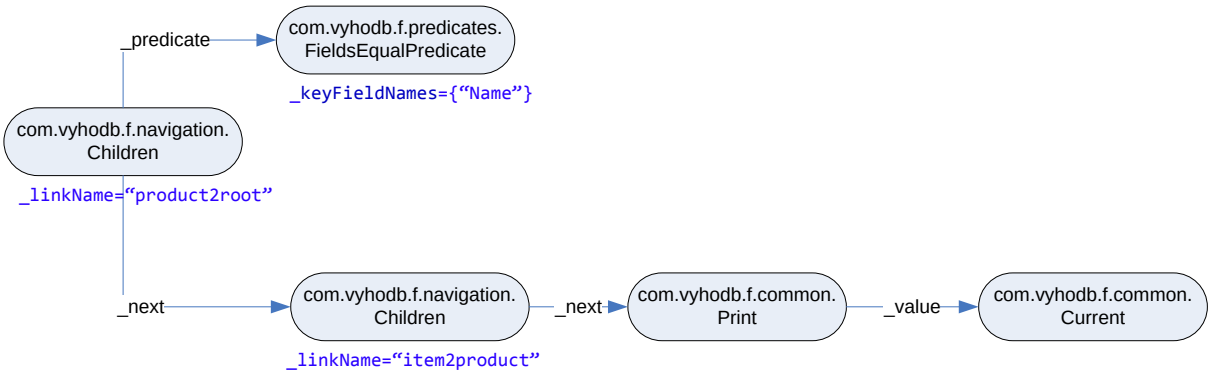
Special static methods (so called factory methods) are used to create function objects. These static methods are located in classes (so called factory classes) inside **com.vyhodb.f** package. To make code more readable and short, static import directives are used:

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

Using static import directives makes function's building code to look like functional programming (just like LISP).

As a result of function's building we get function's tree, which is shown on diagram below[1]:

[1] Some functions are omitted for simplicity (for instance internal predicate on children("item2product")).



Function evaluation

When we have tree of functions we can evaluate it. In our example we start evaluation of root function childrenIf("product2root"), which in turns evaluates others.

Method **evalTree()** evaluates function:

```
/**
 * Evaluates function as part of function tree evaluation.
 *
 * @param current
 *            current object
 * @param context
 *            evaluation context
 * @return evaluation result
 */
public abstract Object evalTree(Object current, Map<String, Object> context);
```

It accepts the following parameters:

1) Current object
2) Evaluation context

Usually, vyhodb Record is used as a current object; however it might be any other object. Functions read, modify, supersede current object – so they do any actions based on current object. Their result and behavior can also depend on child (next) functions evaluation result.

Evaluation context serves as a shared memory for functions within tree. Functions, during their evaluation can read/write key-value pairs.

Functions from **com.vyhodb.f.*** packages are thread-safe. It means, that you can build function tree only once and use it afterward many time by many concurrent threads. To be thread-safe function must satisfy the following criteria:

1) All objects which are referenced by function object should be immutable or thread-safe.
2) If function needs access to mutable (thread specific) object, then it should be passed as a current object or be placed into context.

com.vyhodb.f.F class implements Serializable interface, so you can serialize/deserialize tree of functions. Moreover your custom application can build function and pass it into RSI Service to evaluation on server side, using RSI invocation.

This approach, however, has security issues and can be used only when custom application and vyhodb server are running in trusted domain.

Navigation functions

Navigation functions are intended for traversing over graph of vyhodb records. Their concept is simple: they read parent or child records from current object (which is casted to Record) and after that evaluate next function down to tree, passing parent/child record as a current object. In case of child records, functions usually evaluate next functions many times for each child record.

Fabric methods for building navigation functions are defined in **com.vyhodb.f.NavigationFactory** class. Table below gives brief info about them:

Function	Description
children()	Retrieves child records from current vyhodb record. For each child record evaluates next functions down to tree and passes child record as current object.
parent()	Retrieves parent record from current vyhodb record. Evaluates next functions and passes parent record as current object.
search()	Searches child records using index. For each found child record evaluates next functions in tree and passes child record as current object to them.
hierarchy()	Traverses hierarchy structure in downwards direction. See "Functions Reference" for more information about hierarchies and hierarchy() function.

To illustrate navigation functions let's have a look at example below. In this example we print Product records, which are sold for particular Customer (we use index "order2root.Customer" to search orders by customer name):

```
package com.vyhodb.guide.functions;

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

. . .

public class NavigationFunctions {

. . .

    private static void example(Space space) {
        String customerName = "Customer 1";

        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        F f =
        search("order2root.Customer", CriterionFactory.equal(customerName),
            children("item2order",
                parent("item2product",
                    printCurrent()
                )
            )
        );

        f.eval(rootRecord);
    }
}
```

Output:

```
{Name="Product 1", Price=12.45} id=2063
{Name="Product 2", Price=1456.99} id=2074
{Name="Product 3", Price=58.99} id=2085
```

Record filtering

The following functions (with 'If' suffix) are indented for filtering records during record graph traversing: childrenIf(), parentIf(), searchIf(), hierarchyIf(). Some of their signatures:

```
public static F childrenIf(String linkName, Predicate predicate, F... next)

public static F parentIf(String linkName, Predicate predicate, F... next)
```

Their signatures have additional parameter – predicate (or predicate function). Predicates are discussed in next section; here we just mention that predicate function is a function which returns Boolean result.

In relation to navigation functions, predicates are used to validate records and understand whether next functions can be evaluated for particular child/parent record.

Predicates

Predicate is a function, which returns Boolean result as its evaluation. Base class for all predicate functions is **com.vyhodb.f.Predicate**:

```
public abstract class Predicate extends F {

    @Override
    public abstract Boolean evalTree(Object current, Map<String, Object> context);
}
```

Predicates are used for condition's validation. For instance, in record navigation functions they are used for checking whether current parent/child record could be passed further to next function.

Let's illustrate it by example below. Here we print Order records which client is "Customer 2" or which Date is "19.05.2015" or later:

```
package com.vyhodb.guide.functions;

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 PredicateExample1 {
. . .
    private static void example(Space space) {
        String customerName = "Customer 2";
        Date date = new Date("05/19/2015");

        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        F f =
        childrenIf("order2root",
            or(
                more(getField("Date"), c(date)),
                equal(getField("Customer"), c(customerName))
            ),

            printCurrent()
        );

        f.eval(rootRecord);
```

```
    }
}
```

Output:

```
{Customer="Customer 2", Date="Tue May 19 00:00:00 BRT 2015"} id=2162
{Customer="Customer 3", Date="Wed May 20 00:00:00 BRT 2015"} id=2195
```

One more example, where we print Order records which customer is "Customer 1". For other orders we print their OrderItem's products:

```
package com.vyhodb.guide.functions;

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

public class PredicateExample2 {
. . .

    private static void example(Space space) {
        String customerName = "Customer 1";

        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        F f =
        children("order2root",
            _if_else(
                // Predicate
                fieldsEqual("Customer", customerName),

                // True
                children("item2order",
                    printCurrent()
                ),

                // False
                children("item2order",
                    parent("item2product",
                        printCurrent()
                    )
                )
            )
        );

        f.eval(rootRecord);
    }
}
```

Output:

```
{Cost=62.25, Count=5} id=2129
{Cost=14569.90, Count=10} id=2140
{Cost=884.85, Count=15} id=2151
{Name="Product 3", Price=58.99} id=2085
{Name="Product 1", Price=12.45} id=2063
{Name="Product 2", Price=1456.99} id=2074
{Name="Product 3", Price=58.99} id=2085
```

Evaluation context

Evaluation context is a shared memory used by tree functions during evaluation. Context is an **Map<String, Object>** object which is created at the beginning of evaluation in **F#eval()** method.

Functions can read and write key-value pairs of context. Moreover, there are special functions in **com.vyhodb.f.CommonFactory** factory for reading, putting and clearing value from/into context: get(), put(), clear().

In example below we show how to use functions: get(), put(), clear(). Our example searches Product record with "Product 2" name and puts it into evaluation context. After that, we traverse over all OrderItem records and change their parent Product record to ones, stored in context.

We also use **com.vyhodb.f.RecordFactory** factory for creating functions which operate on records; and **com.vyhodb.f.CommonFactory** factory for **current()** function (it returns current object).

```
package com.vyhodb.guide.functions;

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

        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        F updateF =
        composite(
            // Searches product
            childrenIf("product2root", fieldsEqual("Name", productName),
                put("Product", current()),  // Stores record in context with "Product" key
                _break()
            ),

            // Updates all links "item2product" to found product
            children("order2root",
                children("item2order",
                    setParent("item2product", get("Product"))
                )
            )
        );

        // Updates Links
        updateF.eval(rootRecord);

        F printF =
        children("order2root",
            children("item2order",
                parent("item2product",
                    printCurrent()
                )
            )
        );

        // Prints modification result
        printF.eval(rootRecord);
    }
}
```

Output:

```
{Name="Product 2", Price=1456.99} id=2074
{Name="Product 2", Price=1456.99} id=2074
{Name="Product 2", Price=1456.99} id=2074
```

```
{Name="Product 2", Price=1456.99} id=2074
{Name="Product 2", Price=1456.99} id=2074
{Name="Product 2", Price=1456.99} id=2074
{Name="Product 2", Price=1456.99} id=2074
```

Aggregates

Aggregates are functions used for calculating min, max, sum and count values. Aggregates are created by **com.vyhodb.f.AggregatesFactory** factory. Aggregate functions operate on particular value, stored in context (so called aggregate – holds result value).

There are three groups of aggregate functions:

1) Functions which calculate aggregated value and store it in evaluation context: **sum()**, **min()**, **max()**, **count()**.
2) Functions which return aggregated value, stored in context: **getSum()**, **getMin()**, **getMax()**, **getCount()**.
3) Functions which clear aggregated values in context: **clearSum()**, **clearMin()**, **clearMax()**, **clearCount()**.

Example below calculates count of all OrderItem record:

```
package com.vyhodb.guide.functions;

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

. . .

public class AggregatesExample1 {

. . .

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

        F countF =
        composite(
            children("order2root",
                children("item2order",
                    count()
                )
            ),
            getCount()
        );

        long count = (long) countF.eval(rootRecord);
        System.out.println(count);
    }
}
```

Output:

```
7
```

Next example calculates total sales amount:

```
package com.vyhodb.guide.functions;

import static com.vyhodb.f.AggregatesFactory.*;
```

```
import static com.vyhodb.f.CommonFactory.*;
import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.f.RecordFactory.*;
. . .

public class AggregatesExample2 {
. . .

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

        F salesF =
        composite(
            children("order2root",
                children("item2order",
                    sum(getField("Cost"))
                )
            ),
            getSum()
        );

        BigDecimal sales = (BigDecimal) salesF.eval(rootRecord);
        System.out.println(sales);
    }
}
```

Output:

```
67268.90
```

Below there is the most interesting example. It calculates total cost of each order (based on OrderItem records) and stores it in new Order's field "Cost":

```
package com.vyhodb.guide.functions;

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

public class AggregatesExample3 {
. . .

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

        F updateF =
        children("order2root",
            children("item2order",
                sum(getField("Cost"))
            ),
            setField("Cost", clearSum())
        );

        updateF.eval(rootRecord);

        // Function for printing "Order" records
        F printF =
        children("order2root",
            printCurrent()
        );

        // Prints modified "Order" records
        printF.eval(rootRecord);
    }
}
```

Function **clearSum()** returns the last aggregate's value, that is why it is used for setting field "Cost".

Output:

```
{Cost=15517.00, Customer="Customer 1", Date="Mon May 18 00:00:00 BRT 2015"} id=2096
{Cost=5899.00, Customer="Customer 2", Date="Tue May 19 00:00:00 BRT 2015"} id=2162
{Cost=45852.90, Customer="Customer 3", Date="Wed May 20 00:00:00 BRT 2015"} id=2195
```

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.

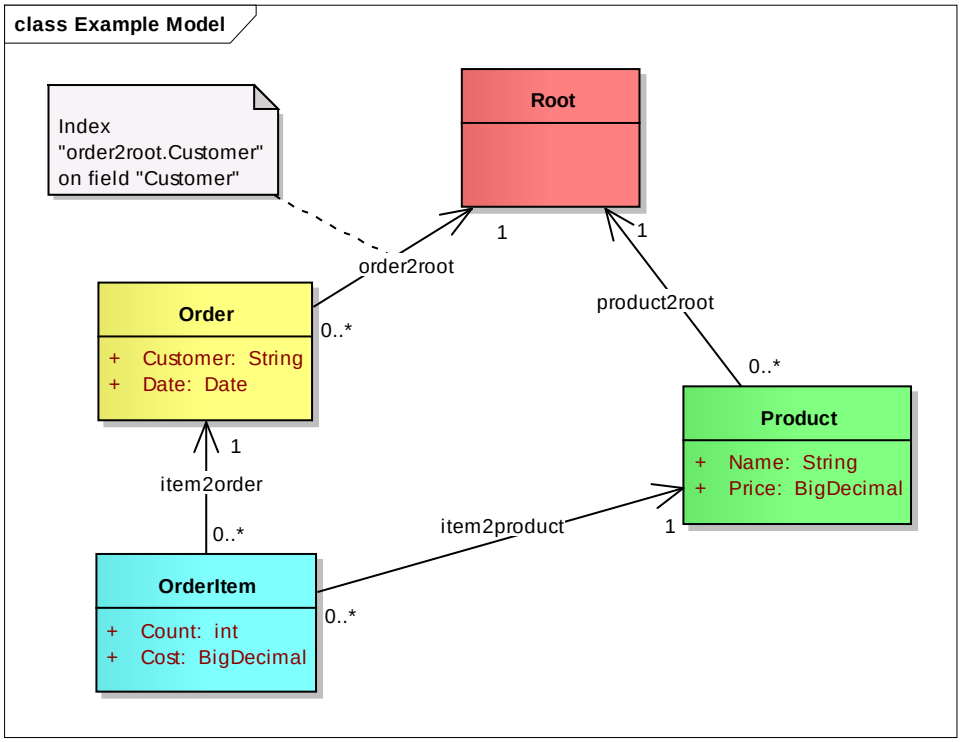
"ONM API" doesn't generate any classes nor uses proxy objects. Java objects are created, inspected and changed using "Java Reflection API". This allows using resulted Java objects in serialization/deserialization and pass ones by RSI or any other RMI framework.

All three operations (ONM Reading, ONM Writing, ONM Cloning) are explicitly called by application; ONM API doesn't perform any actions behind the scene.

ONM API classes, interfaces and annotations are located in package **com.vyhodb.onm** (**vdb-core-0.9.0.jar** jar file).

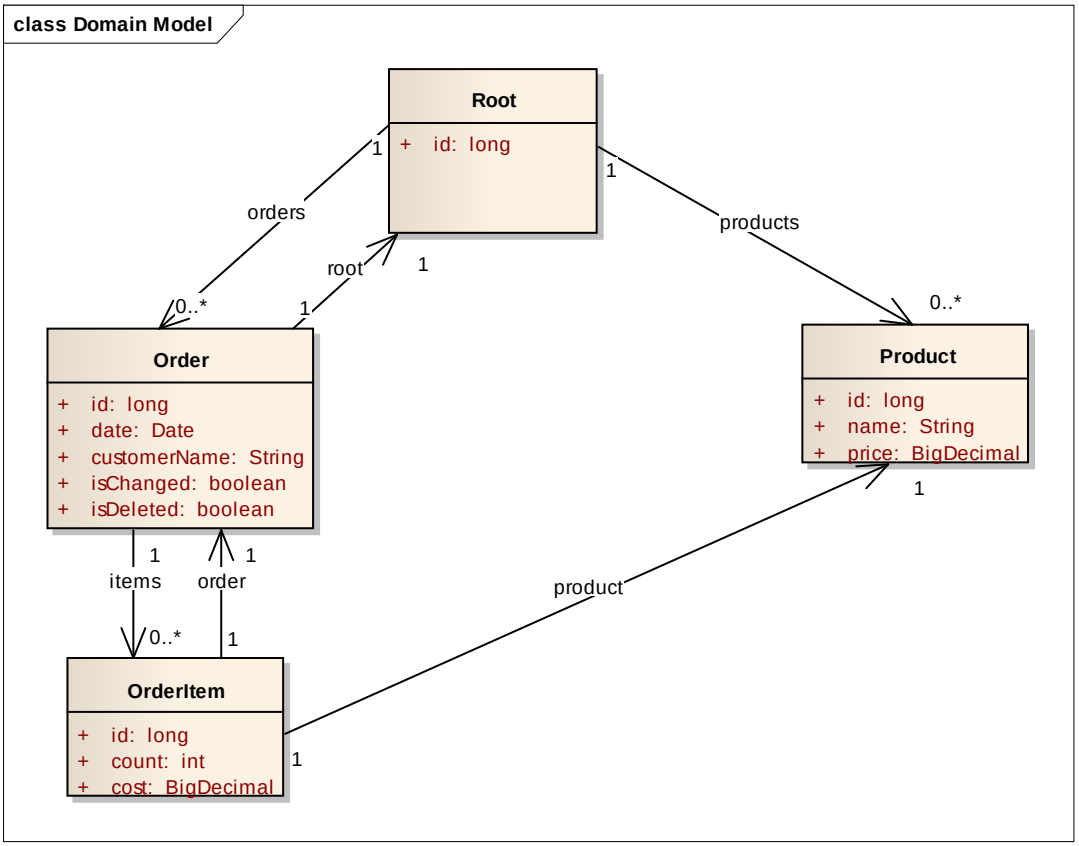
Domain model

To illustrate examples we use the same data model as in previous chapter:



To create sample data we will call **com.vyhodb.utils.DataGenerator#generate().**

ONM API is a mapping framework, so diagram below shows Java class model, which is used throughout the chapter:



Java source code for domain model is located in package: **com.vyhodb.guide.onm.domain**.

ONM Mapping

ONM API uses java reflection and operates on class field level (JavaBean properties are not used). Therefore mapping is defined on class field level.

ONM API can't access private fields of parent class.

There are two ways of defining mapping:

1) Annotations
2) Xml (usually external xml file)

Mapping class

Objects of **com.vyhodb.onm.Mapping** class are caches of mapping info. They are used in every ONM operation (ONM Reading, ONM Writing, ONM Cloning).

Mapping class if thread-safe and its objects can be shared by multiple threads.

Mapping class contains static methods for creating its objects based on source of mapping information. For instance, in case when mapping is specified by annotations, use:

```
public static Mapping newAnnotationMapping()
```

In case of XML source, use one of the following methods:

```
public static Mapping newXmlMapping(File file)
public static Mapping newXmlMapping(InputStream inputStream)
public static Mapping newXmlMapping(Reader reader)
public static Mapping newXmlMapping(URL url)
```

Annotations

Mapping annotations are located in **com.vyhodb.onm** package.

Below there is an Order class's source code with mapping annotations from our domain model:

```
package com.vyhodb.guide.onm.domain;

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

import com.vyhodb.onm.Children;
import com.vyhodb.onm.Field;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.IsChanged;
import com.vyhodb.onm.IsDeleted;
import com.vyhodb.onm.Parent;
import com.vyhodb.onm.Record;

@Record
public class Order {

    @Id
    private long id = -1;

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

    @Field(fieldName="Date")
    private Date date;

    @Parent(linkName="order2root")
    private Root root;

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

    @IsChanged
    private boolean isChanged = false;

    @IsDeleted
    private boolean isDeleted = false;

    public long getId() {
        return id;
    }

    public String getCustomerName() {
        return customerName;
    }

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

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
```

```
        this.date = date;
    }

    public Root getRoot() {
        return root;
    }

    public void setRoot(Root root) {
        this.root = root;
    }

    public boolean isChanged() {
        return isChanged;
    }

    public void setChanged(boolean isChanged) {
        this.isChanged = isChanged;
    }

    public boolean isDeleted() {
        return isDeleted;
    }

    public void setDeleted(boolean isDeleted) {
        this.isDeleted = isDeleted;
    }

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

    @Override
    public String toString() {
        . . .
    }
}
```

Annotations:

Annotation	Mandatory	Applicable for	Used in ONM operations	Description
@Record	Yes	class	Reading, Writing, Cloning	Allows particular class to participate in ONM operations.
@Id	Yes	field	Reading, Writing, Cloning	Marks record identifier field. Field must be of primitive long or Long class type. If annotated field's value <0 during ONM Writing operation, then object is considered as new and new record with corresponding fields and parent links will be created. Annotated field is updated to new record's identifier.
@Field	No	field	Reading, Writing, Cloning	Maps class's field and record's field. Record's field name is specified by "fieldName" element.
@Parent	No	field	Reading, Writing, Cloning	Maps class's field and parent record. Class of annotated field must participate in ONM and be annotated by @Record. Link name is specified by "linkName" element.

@Children	No	field	Reading, Writing, Cloning	Maps class's field and child records. Class of annotated field must implement interface Collection<T>, where T – is a class, participated in ONM and annotated by @Record. Link name is specified by "linkName" element.
@IsChanged	No	field	Writing	Annotates boolean field, which indicates whether object is changed or not. Field's value: 1) true – object has changed and corresponding record will be changed (including parent links) during ONM Writing operation. 2) false – object is clean, corresponding record won't be modified. When @IsChanged isn't defined within class, object always considered as changed and corresponding record is always updated (during ONM Writing).
@IsDeleted	No	field	Writing	Annotates boolean field, which is object's deletion indicator. Field's value: 1) true – object is considered as deleted, corresponding record will be deleted during ONM Writing operation. 2) false – object is not deleted. When @IsDeleted isn't defined within class, object always considered as existed (or new).

XML file

ONM mapping can also be specified by XML file. XSD Schema of mapping xml file (OnmSchema.xsd) is located in **com.vyhodb.onm** package.

Below, there is an example of xml mapping file for java class domain model, used in this chapter. This file can also be found in **com.vyhodb.guide.onm.domain** package.

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
    <class name="com.vyhodb.guide.onm.domain.Root" id="id">
        <childrenSet>
            <children name="products" linkName="product2root" />
            <children name="orders" linkName="order2root" />
        </childrenSet>
    </class>

    <class name="com.vyhodb.guide.onm.domain.Product" id="id">
        <fieldSet>
            <field name="name" fieldName="Name" />
            <field name="price" fieldName="Price" />
        </fieldSet>
    </class>

    <class name="com.vyhodb.guide.onm.domain.Order" id="id" isChanged="isChanged"
isDeleted="isDeleted">
```

```
        <fieldSet>
            <field name="customerName" fieldName="Customer" />
            <field name="date" fieldName="Date" />
        </fieldSet>
        <parentSet>
            <parent name="root" linkName="order2root" />
        </parentSet>
        <childrenSet>
            <children name="items" linkName="item2order" />
        </childrenSet>
    </class>

    <class name="com.vyhodb.guide.onm.domain.OrderItem" id="id">
        <fieldSet>
            <field name="count" fieldName="Count" />
            <field name="cost" fieldName="Cost" />
        </fieldSet>
        <parentSet>
            <parent name="order" linkName="item2order" />
            <parent name="product" linkName="item2product" />
        </parentSet>
    </class>
</metadata>
```

Java class requirements

There is a list of requirements for Java classes which intended to be used in ONM operations:

1) Class must have public no-argument constructor.
2) Class must be annotated by @Record or have appropriate <class/> element in xml mapping.
3) Class must have long/Long field which is annotated by @Id or specified in xml file by "id" attribute

Note, that ONM uses Java reflection for field access; therefore private fields of parent class are hidden for it and can't participate in ONM.

ONM Reading

Example

Let's have a look at example of using ONM Reading operation:

```
package com.vyhodb.guide.onm;

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

import com.vyhodb.f.F;
import com.vyhodb.guide.onm.domain.Root;
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;

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

public class OnmRead {

    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)) {
            TrxSpace space = server.startModifyTrx();
            example(space);
            space.rollback();
        }
    }

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

        Mapping mapping = Mapping.newAnnotationMapping();

        F onmReadF =
        startRead(Root.class, mapping,
            children("order2root",
                children("item2order",
                    parent("item2product")
                )
            ),
            children("product2root")
        );

        Root object = (Root) onmReadF.eval(rootRecord);
        System.out.println(object);
    }
}
```

We are going to examine **example()** method, where ONM Reading takes place. This method performs the following actions:

1) Generates sample data using **com.vyhodb.utils.DataGenerator.**
2) Creates Mapping object, which is a cache of mapping info between java classes and vyhodb records.
3) Creates function tree. Root function startRead() accepts the following parameters:
    a. Class of the object, which is a root object of read java object graph. In our case this is **com.vyhodb.guide.onm.domain.Root.**
    b. **Mapping** object.
4) Evaluates function tree. Evaluation result – object of **com.vyhodb.guide.onm.domain.Root** class, which is a root of java object graph.

Output (Root, OrderItem, Product classes have overridden toString() methods):

```
Root:
Products: {
  Product [id=2063, name=Product 1, price=12.45]
  Product [id=2074, name=Product 2, price=1456.99]
  Product [id=2085, name=Product 3, price=58.99]
}
Orders: {
  Order [id=2096, customerName='Customer 1', date=Mon May 18 00:00:00 BRT 2015], Items: {
    OrderItem [id=2129, count=5, cost=62.25, product='Product 1']
    OrderItem [id=2140, count=10, cost=14569.90, product='Product 2']
    OrderItem [id=2151, count=15, cost=884.85, product='Product 3']
  }
  Order [id=2162, customerName='Customer 2', date=Tue May 19 00:00:00 BRT 2015], Items: {
    OrderItem [id=2184, count=100, cost=5899.00, product='Product 3']
  }
  Order [id=2195, customerName='Customer 3', date=Wed May 20 00:00:00 BRT 2015], Items: {
```

```
    OrderItem [id=2217, count=30, cost=373.50, product='Product 1']
    OrderItem [id=2228, count=30, cost=43709.70, product='Product 2']
    OrderItem [id=2239, count=30, cost=1769.70, product='Product 3']
  }
}
```

Consider building of **onmReadF** function in more details:

```
. . .

F onmReadF =
        startRead(Root.class, mapping,
            children("order2root",
                children("item2order",
                    parent("item2product")
                )
            ),
            children("product2root")
        );
. . .
```

To start ONM Reading operation, **startRead()** function is used. Its fabric methods are defined in **com.vyhodb.onm.OnmFactory** class.

```
public static F startRead(Class<?> rootClass, F... next)

public static F startRead(Class<?> rootClass, Mapping mapping, F... next)
```

startRead() function wraps record navigation functions (see **Navigation functions**), which form record traversal route. Java objects are created for each visited record in traversal route. References between java objects are also set according to traversal route (in fields: @Parent, @Children).

Logic of Java object creation and setting references between them is moved to **com.vyhodb.onm.f.read.ReadStack** class, which implements **com.vyhodb.f.Stack** interface (for more information about Stack and how it is used by navigation functions see "Functions Reference" document). **startRead()** function just creates **ReadStack** object and puts it into context, so that navigation functions notify it about visited records.

ONM Reading creates graph of java objects, not a tree! In other words, when it visits particular record second, third and so forth times, new object isn't created: object, which has been created at first visit, is searched and references are set on it (parent and child).

As a result of this we get java object graph with references between objects according to traversal route.

For the same java classes it is possible to create different object graphs by changing traversal route over records. Navigation functions with filtering can also be used to filter out records.

Error-prone traversing

In case of ONM Reading, navigation using links which mappings are absent for classes, throws an **com.vyhodb.onm.OnmException.**

For instance, if we change function from our previous example by the following:

```
F onmReadF =
        startRead(Root.class, mapping,
            children("order2root",
                children("item2order",
                    parent("item2product",
                        parent("product2root")
```

```
                    )
                )
            ),
            children("product2root")
        );
```

or by the next one:

```
F onmReadF =
        startRead(Root.class, mapping,
            children("order2root",
                children("item2order",
                    parent("item2product")
                )
            ),
            children("product2root",
                children("item2product")
            )
        );
```

then we get **OnmException**.

Despite of the fact, that corresponding links ("product2root", "item2product") are exists between records, Product class has no mappings for them.

ONM Writing

There is an example of ONM Writing below (method main(), constants LOG, DATA and import directives are omitted, because they are identical to previous example):

```
package com.vyhodb.guide.onm;

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

. . .

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")
                )
            ),
            children("product2root")
        );

        // 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);

        // Prints read and updated graphs
        System.out.println("Before modification: \n" + readRoot);
        System.out.println("\nAfter modification: \n" + updatedRoot);
    }

    private static void modify(Root root) {
        // Deletes two orders
        List<Order> orders = root.getOrders();
        orders.get(0).setDeleted(true);
        orders.get(1).setDeleted(true);

        // Changes Customer
        Order order = orders.get(2);
        order.setCustomerName("Changes Customer");
        order.setChanged(true);

        // Changes product of order items
        Product product = root.getProducts().get(0);
        for (OrderItem item : order.getItems()) {
            item.setProduct(product);
        }
    }
}
```

The line below actually performs ONM Writing:

```
. . .
Writer.write(mapping, readRoot, space);
. . .
```

ONM Writing doesn't use Functions API. Instead of it, **com.vyhodb.onm.Writer#write()** method traverses over java graphs' objects and updates vyhodb records for each visited java object. Java object graph is specified by one of its object (**readRoot** variable in our example).

ONM Writing algorithm described in next section in more details.

Java graph traversing algorithm

We start from describing java object's state – when it is considered as new, modified or deleted. After that we describe how ONM Writing handles particular java object. ONM Writing is recursive algorithm: so the same steps are performed for each visited java object.

Object states

1) Object is considered new and new record is created for it, when field value, annotated by @Id, is < 0.
2) Object is considered deleted if it has **boolean** field, annotated by @IsDeleted, and this field's value is **true**.
3) Object is considered changed, when one of the following conditions is true:
    a. It doesn't have field, annotated by @IsChanged
    b. It has boolean field, annotated by @IsChanged and this field's value is **true**.

Object handling

ONM Writing starts with object, passed as **rootObject** parameter into **com.vyhodb.onm.Writer#write()** method. The following steps are performed for it and every other object in graph:

1) If object is new, then new record is created, fields are updated and parent links are set.
2) If object is changed, then fields and parent links are updated on record, corresponding to object.
3) If object is deleted, then corresponding record is deleted as well.

4) Parent objects traversal. For each field, annotated by @Parent, referenced object is retrieved and this algorithm is performed for it (from first step).
5) Child objects traversal. For each field, annotated by @Children, Collection object is retrieved. This algorithm is performed for each object from Collection (from first step).

Notes about object traversal

1) Collection elements (child objects) are used only for object traversal purposes. No record links are updated based on them.
2) Objects considered as deleted, are traversed as well: their parent and child objects are reached and visited.

ONM Cloning

ONM Cloning example is shown below (main() method, LOG, DATA constants and import directives are omitted):

```
package com.vyhodb.guide.onm;

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

. . .

public class OnmClone {

. . .
    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")
                )
            ),
            children("product2root")
        );

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

        // Gets Order object which is cloned
        Order sourceOrder = root.searchOrder("Customer 2");

        // Builds ONM Clone function
        F cloneF =
        startClone(mapping,
            objectChildren("item2order",
                objectParent("item2product")
            )
        );

        // Clones Order, Order Items and Products
        Order clonedOrder = (Order) cloneF.eval(sourceOrder);
        System.out.println(clonedOrder);
    }
}
```

Output:

```
Order [id=2162, customerName='Customer 2', date=Tue May 19 00:00:00 BRT 2015], Items: {
  OrderItem [id=2184, count=100, cost=5899.00, product='Product 3']
}
```

Java object cloning is implemented in the following code fragment:

```
// Builds ONM Clone function
F cloneF =
startClone(mapping,
    objectChildren("item2order",
        objectParent("item2product")
    )
);

// Clones Order, Order Items and Products
Order clonedOrder = (Order) cloneF.eval(sourceOrder);
```

Cloning is done by evaluation of function tree, which consists of object navigation functions, which in turns, wrapped by start cloning function – **startClone()**.

Object navigation functions are described in next section.

Object navigation functions

Object navigation functions are intended for traversing over java objects, using ONM mapping info (**com.vyhodb.onm.Mapping** object actually). Those functions get @Parent and @Children annotated fields to move from one to another object.

Functions get Mapping object from evaluation context (context key is "Sys$Mapping").

Object navigation functions don't do cloning, they just traverse over objects and notify **com.vyhodb.f.Stack** object about visited objects. **com.vyhodb.f.Stack** object is retrieved from evaluation context as well.

Object navigation functions are defined in **com.vyhodb.onm.OnmFactory** class.

startClone()

startClone() starts cloning process and does the following actions:

1) Puts Mapping object into evaluation context (context key = "Sys$Mapping").
2) Creates object **com.vyhodb.onm.f.object.clone.CloneStack** (which is actually implements cloning) and put it into context (context key ="Sys$Stack"). This object is a Stack object, which is notified by object navigation functions about visited objects.
3) Clones root object.
4) Evaluates next functions, which it wraps. Navigation functions during their evaluation notify Stack object, which in turns clones visited objects and set references between them.
5) Returns cloned object, which is actually a root of cloned java object graph.

For more information about **com.vyhodb.f.Stack** interface see document "Functions Reference".

Admin API

com.vyhodb.admin.Admin class is used for running administration operations like creating new storage, backup storage, log file truncating, etc. This class is located in **vdb-core-0.9.0.jar.**

In order to use **com.vyhodb.admin.Admin** objects, all jar files from lib directory should be accessible by class loader. In simple case, it means that custom application's **classpath** should include those jar files.

Example below shows how to use Admin API in order to create new vyhodb storage:

```
package com.vyhodb.guide.admin;

import java.io.IOException;
import com.vyhodb.admin.Admin;

public class AdminExample {

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

    public static void main(String[] args) throws IOException {
        Admin admin = Admin.getInstance();

        // Removes existed storage files
        admin.removeStorageFiles(LOG, DATA);

        // Creates new storage files
        admin.newStorage(LOG, DATA);
    }
}
```

For more information about Admin class see Javadoc and source code.

Appendix A. Supported field value classes

Tables below show classes which can be used as a field's values.

Classes:

java.lang.String	java.lang.Boolean	java.math.BigInteger
java.lang.Long	java.util.Date	java.lang.Double
java.lang.Integer	java.math.BigDecimal	java.lang.Float
java.lang.Character	java.lang.Byte	java.util.UUID
java.lang.Short		

Primitive arrays:

long[]	double[]	short[]
int[]	float[]	byte[]
boolean[]	char[]	

Object arrays:

String[]	BigInteger[]
Date[]	UUID[]
BigDecimal[]	

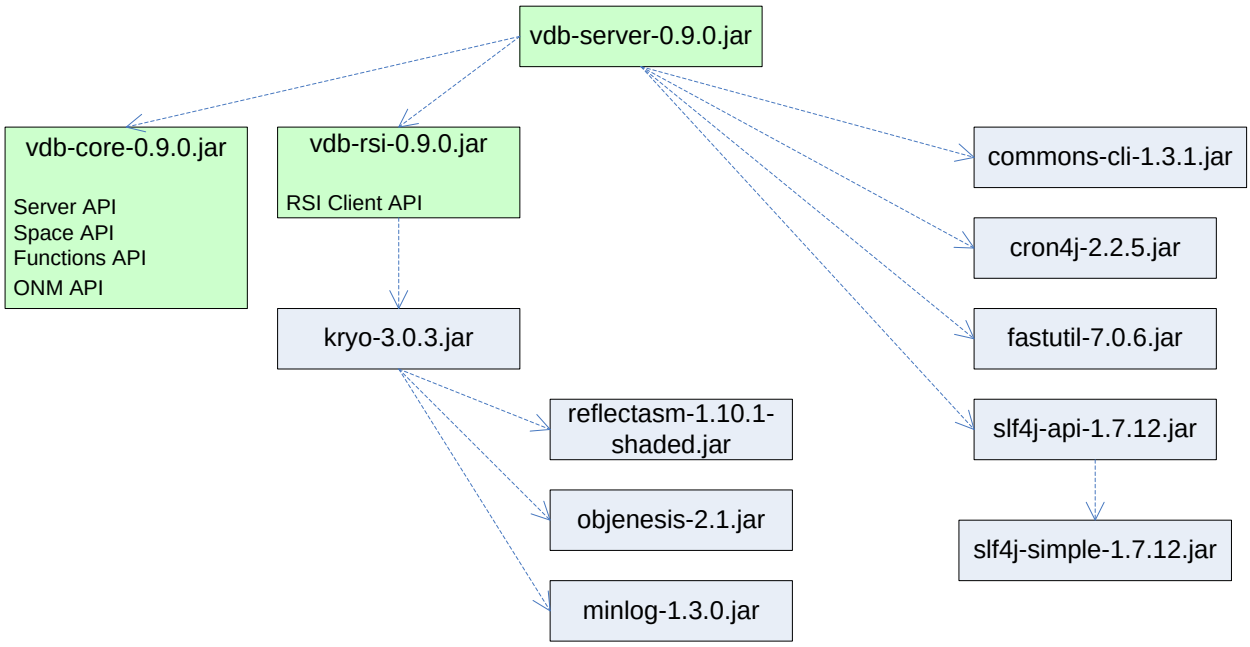
Appendix B. Supported indexed field value classes

Table below shows classes which can be used in indexed fields:

java.lang.String	java.lang.Short	java.math.BigInteger
java.lang.Long	java.util.Date	java.lang.Double
java.lang.Integer	java.math.BigDecimal	java.lang.Float
java.lang.Character	java.lang.Byte	java.util.UUID

Appendix C. Jar's dependencies

Diagram below shows vyhodb jars archives (located in **lib** directory), dependencies between them and API distribution:



Legend

boo.jar vyhodb jar

foo.jar third party jar