

# UAKGQuery: Programming Guide

January 30, 2002

DocumentID: GradSoft-PR-e-16/06/2000-1.5

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	History . . . . .	3
1.2	Knowledge Needed For Reading This Manual . . . . .	3
<b>2</b>	<b>General description of UAKGQuery using</b>	<b>3</b>
<b>3</b>	<b>Connection To Database</b>	<b>3</b>
<b>4</b>	<b>Example of query evaluation</b>	<b>5</b>
<b>5</b>	<b>Data Types</b>	<b>7</b>
5.1	FieldValue . . . . .	7
5.1.1	C++ Facade class FieldValueAccess . . . . .	9
5.2	Record, RecordSeq . . . . .	10
5.2.1	C++ Facade classes RecordAccess, SRecord . . . . .	10
5.3	OctSeq, RCHeader . . . . .	11
5.3.1	RCReader . . . . .	12
5.3.2	RCWriter . . . . .	12
5.4	RecordDescription . . . . .	13
<b>6</b>	<b>Evaluation of queries</b>	<b>13</b>
6.1	Queries without parameters . . . . .	14
6.2	queries with parameters . . . . .	15
<b>7</b>	<b>Query Interface</b>	<b>16</b>
7.1	Prepared Query . . . . .	20
7.2	Fetching data via Iterator pattern . . . . .	21
7.3	Receiving of result description . . . . .	21
<b>8</b>	<b>QueryManager interface</b>	<b>22</b>
8.1	Sequences . . . . .	22
8.2	Properties . . . . .	22

<b>9</b>	<b>Large Objects (LOBs)</b>	<b>23</b>
<b>10</b>	<b>Collections</b>	<b>26</b>
10.1	Introduction . . . . .	26
10.2	General description . . . . .	26
10.2.1	“Physical sense” of collection . . . . .	26
10.3	Steps for using collections interfaces . . . . .	27
10.4	Creation of Query Collections . . . . .	27
10.5	Data Access . . . . .	28
10.5.1	Receiving data with the help of Iterator’s interface . . . . .	28
10.5.2	Retrieving data via collection methods . . . . .	29
10.5.3	Pattern matching . . . . .	29
10.6	adding, updating and deleting of data . . . . .	30
10.7	Collection queries . . . . .	32
10.7.1	Limitations . . . . .	33
10.7.2	List of methods . . . . .	33
10.8	SubCollection  . . . . .	33
10.9	UAKGKeyCollection . . . . .	34
10.9.1	Creation of UAKGKeyCollection . . . . .	34
10.9.2	Methods of UAKGKeyCollection . . . . .	35
10.10	Using of UAKGCollectionListener . . . . .	36
10.11	Limitations of using UAKGQuery collections interfaces . . . . .	37
<b>11</b>	<b>Transactions</b>	<b>37</b>
11.0.1	XA transactions . . . . .	38
11.0.2	UAKG OTS Transactions . . . . .	38
11.1	Typical Usage Of Transactions . . . . .	38
<b>12</b>	<b>IDL definitions</b>	<b>39</b>
12.1	CosQueryCollection . . . . .	39
12.2	CosQuery . . . . .	45
12.3	UAKGQuery . . . . .	47
12.4	RC.idl . . . . .	66
<b>13</b>	<b>RC-coding specifications</b>	<b>74</b>
<b>14</b>	<b>Grammar of collection SQL queries</b>	<b>76</b>
14.1	List Of Changes . . . . .	83

## 1 Introduction

UAKGQuery is a CORBA [1] service intended for database access. It provides CORBA API for effective querying of relational databases and define high-level object interfaces for data access.

With UAKGQuery it is possible to organize effective and uniform access to relational databases, based on high-level abstract model and independent from database service locations and database architecture limits.

This document is unformal description of using UAKGQuery API. For full formal reference, please, look at API guide.

## 1.1 History

UAKGQuery Service is in active development from 1998. It grew from the implementation of CORBA Query Service [2]. After analyzing of scalability and performance of original CosQuery API, structure of UAKGQuery Service was fully refined. Some of methods applied for building of effective distributed applications can be found in [6] [5]

## 1.2 Knowledge Needed For Reading This Manual

We assume, that reader has basic knowledge of CORBA architecture [1] and has some experience with CORBA C++ application. (we recommend [4] as learning material). Also assumed, that reader knows what SQL is and what relational databases are.

The last chapter requires knowing of such entities, as transactions, CORBA Transaction Service [3] and XA transactions [7]. But you can omit this chapter during the first reading.

## 2 General description of UAKGQuery using

Application programmer must perform next steps for using UAKGQuery:

- Initialize ORB, by calling function `CORBA::ORB_init`
- Initialize UAKGQueryService, by calling function `initUAKGQueryService`. This function is defined in header file `UAKGQueryInit.h`
- Receive object reference to `DBConnectionManager` (manager of database connection) via CORBA mechanism of resolving initial references.
- Create `QueryManager`, by calling `DBConnectionManager::createQueryManager`, which expose API for creation and evaluation of SQL queries and so-called collections.
- With `QueryManager` it is possible whether directly evaluator SQL queries, or create `Query` or `Collection` objects for interactive reading/writing data.
- Before end of work it is necessary to disconnect from UAKGQuery service, by calling method `destroy` of `QueryManager`.

### 3 Connection To Database

For connecting to database it is necessary to:

1. Receive initial object with name "UAKGQueryService",
2. Narrow it to type UAKGQuery::DBConnectionManager.
3. receive QueryManager, calling DBConnectionManager::create

This can be illustrated by next code fragment:

```
#include CORBA_H // for CORBA
#include <UAKGQuery.h> // for UAKGQuery
#include <UAKGQueryInit.h> // for UAKGQuery initialization

.....

// initialize ORB
orb = CORBA::ORB_init(argc, argv);

// initialize UAKGQueryService
initUAKGQueryService(orb.in());

// receive initial object
CORBA::Object_var obj;
try {
    obj = orb->resolve_initial_references("UAKGQueryService");
} catch(const ORB::InvalidName&){
    cerr << argv[0] << ": can't resolve UAKGQueryService" << endl;
    return 1;
}

if(CORBA::is_nil(obj)) {
    cerr << argv[0] << ": UAKGQueryService is a nil object reference" << endl;
    return 1;
}

// narrow it to DBConnectionManager.
DBConnectionManager_var dbManager = DBConnectionManager::_narrow(obj);
if (CORBA::is_nil(dbManager)) {
    cerr << argv[0] << ": can't narrow dbManager to correct type" << endl;
    return 1;
}

// receive QueryManager
QueryManager_var queryManager;
```

```

try {
    queryManager =
        dbManager->createQueryManager("skott","tiger","Oracle","Oracle8","");
}catch(QueryManagerNotFound){
    cerr << argv[0] <<" : can't find query manager." << endl;
    return 1;
}

// now you can do something with QueryManager
.....
.....

//time to disconnect.
queryManager->destroy();

// destroy ORB
orb->destroy();

```

## 4 Example of query evaluation

You can evaluate queries use family of methods `QueryEvaluator::evaluate`  
`evaluate_rc`, `evaluate_records`, `evaluate_record`, `evaluate_records_inout`,  
`evaluate_rc_inout`,

Those methods differ only by type of input and output parameters. Interface `QueryManager` is inherited from interface `QueryEvaluator`, so you can apply this methods to `QueryMeneger`, which received from `DBConnectionManager`.

Now we will show a simple example of such request, than it will start more detailed description of methods and data types.

```

try {

    OctSeq_var octSeq = queryManager->evaluate_rc("select * from tab","SQL92",
                                                RecordDescriptionAccess::empty(),
                                                RecordAccess::emptyOctSeq());

}catch(QueryInvalid ex){
    cerr << "QueryTypeInvalid" << endl;
}catch(QueryTypeInvalid ex){
    cerr << "QueryTypeInvalid" << endl;
}catch(QueryProcessingError ex){
    cerr << "QueryProcessingError" << endl;
    cerr << ex.why << endl;
}catch(const CORBA::SystemException& ex) {
#ifdef CORBA_SYSTEM_EXCEPTION_IS_STREAMBLE
    cerr << ex;

```

```

#else
    cerr << "System Exception" << endl;
#endif
}

```

What this means:

- OctSeq – This is a type of result, which means packed byte sequence. You can access to this sequence by using classes RCReader and RCWriter, described below.
- "select \* from tab" – text of SQL query.
- "SQL92" - query flags, where you can set query parameters such as language, number of prefetched rows and so on.
- RecordDescriptionAccess::empty() - constant of type RecordDescription: we pass descriptions of input query parameter there. As our query have no input parameters, than we pass empty RecordDescription. It is predefined in helper class RecordDescriptionAccess for programmers pleasure.

<sup>1</sup>o you don't write before any query without parameters next piece of code:

```

RecordDescription emptyDescription;
emptyDescription.length(0);

```

- RecordAccess::emptyOctSeq - input parameters of query. They must be in the same form, as results. As you see, RecordDescription::emptyOctSeq denotes empty octet sequence.

The next is function, which prints requested data:

```

void printRC(ostream& out, const OctSeq& octSeq)
{
    RCHeader header;
    CORBA::ULong pos=0;
    RCReader::readHeader(header,pos,octSeq);
    out << "Header: nRecords=" << header.nRecords << endl;
    out << "Header: nFields=" << header.nFields << endl;
    for(CORBA::Long nRecord=0; nRecord<header.nRecords; ++nRecord){
        for(CORBA::ULong nField=0; nField<header.nFields; ++nField){
            FieldValue_var fv = RCReader::readField(pos,octSeq);
            printField(out,fv);
            out << "|";
        }
        out << '\n';
    }
}

```

---

<sup>1</sup>s

## 5 Data Types

### 5.1 FieldValue

This datatype represents one field in database. It defined in IDL via next definitions:

```
///  
typedef boolean Null;  
  
/**  
 * this union represent one field in DB  
 **/  
union FieldValue switch(Null){  
    case FALSE : Value v;  
};
```

I.e. FieldValue can be NIL (when appropriate value is NIL value in RDB), or can have value of type `CosQueryCollection::Value`

Now, let's look on Value definitions:

```
/**  
 * what can be not null value in DB:  
 **/  
union Value switch(FieldType) {  
    ///  
    case TypeBoolean: boolean b;  
    ///  
    case TypeChar: char c;  
    ///  
    case TypeOctet: octet o;  
    ///  
    case TypeShort : short s;  
    ///  
    case TypeUShort : unsigned short us;  
    ///  
    case TypeLong : long l;  
    ///  
    case TypeULong : unsigned long ul;  
    ///  
    case TypeFloat : float f;  
    ///  
    case TypeDouble : double d;  
    ///  
    case TypeString : string str;  
    ///  
    case TypeObject : Object obj;
```

```

    ///
    case TypeSmallInt : short si;
    ///
    case TypeInteger : long i;
    ///
    case TypeReal : float r;
    ///
    case TypeDoublePrecision : double dp;
    ///
    case TypeCharacter : string ch;
    ///
    case TypeDecimal : Decimal dec;
    ///
    case TypeNumeric : Decimal n;
    ///
    case TypeDateTime : DateTime dt;
    ///
    case TypeRaw : sequence<octet> raw;
    ///
    case TypeLongRaw : sequence<octet> lrawid;
    ///
    case TypeLongString : sequence<octet> lstrid;
    ///
    case TypeWString : string wstr;
    ///
    case TypeLongWString : sequence<octet> lwstrid;
};

```

Relations between SQL types and CORBA types are shown in next table:

	<i>CORBA</i>	<i>SQL ORACLE</i>	<i>SQL InterBase</i>
<i>TypeBoolean</i>	<i>boolean</i>	<i>CHAR(1)[in]</i>	<i>CHAR(1)[in]</i>
<i>TypeChar</i>	<i>char</i>	<i>CHAR(1)</i>	<i>CHAR(1)</i>
<i>TypeOctet</i>	<i>octet</i>	<i>CHAR(1)[in]</i>	<i>CHAR(1)[in]</i>
<i>TypeShort</i>	<i>short</i>	<i>NUMBER(5,0)</i>	<i>SMALLINT</i>
<i>TypeUShort</i>	<i>unsignedshort</i>	<i>NUMBER(5,0)</i>	<i>SMALLINT</i>
<i>TypeLong</i>	<i>long</i>	<i>NUMBER(10,0)</i>	<i>INTEGER</i>
<i>TypeULong</i>	<i>unsignedlong</i>	<i>NUMBER(10,0)</i>	<i>INTEGER</i>
<i>TypeFloat</i>	<i>float</i>	<i>NUMBER(x,y) x &lt; 5 ^ y &gt; 0</i>	<i>FLOAT</i>
<i>TypeDouble</i>	<i>double</i>	<i>NUMBER(x,y) x &gt; 5 ^ y &gt; 0</i>	<i>DOUBLEPRECISION</i>
<i>TypeString</i>	<i>string</i>	<i>VARCHAR2(X)</i>	<i>VARCHAR(X)</i>
<i>TypeObject</i>	<i>Object</i>	<i>VARCHAR2(X) ∨ UDF</i>	<i>VARCHAR(X)</i>
<i>TypeSmallInt</i>	<i>short</i>	<i>NUMBER(5,0)</i>	<i>SMALLINT</i>
<i>TypeInteger</i>	<i>long</i>	<i>NUMBER(10,0)</i>	<i>INTEGER</i>
<i>TypeReal,</i>			<i>FLOAT</i>
<i>TypeDoublePrecision</i>			<i>DOUBLEPRECISION</i>
<i>TypeCharacter</i>			<i>CHAR(1)[in]</i>
<i>TypeDecimal</i>	<i>CosQueryCollection :: Decimal</i>	<i>NUMBER(x,y) ^ x &gt; 10</i>	<i>NUMERIC</i>
<i>TypeNumeric</i>	<i>CosQueryCollection :: Decimal</i>		<i>NUMERIC</i>
<i>TypeDateTime</i>	<i>CosQueryCollection :: DateTime</i>	<i>DATE</i>	<i>DATE</i>
<i>TypeRaw</i>	<i>sequence &lt; octet &gt;</i>	<i>LOB</i>	
<i>TypeLongRaw</i>	<i>sequence &lt; octet &gt;</i>	<i>BLOB</i>	
<i>TypeLongString</i>	<i>sequence &lt; char &gt;</i>	<i>CLOB</i>	
<i>TypeLongRaw</i>	<i>sequence &lt; octet &gt;</i>	<i>BLOB</i>	<i>BLOB</i>
<i>TypeLongString</i>	<i>sequence &lt; octet &gt;</i>		
<i>TypeWString</i>	<i>wstring</i>	<i>NCHAR, NVARCHAR</i>	
<i>TypeLongWString</i>	<i>sequence &lt; octet &gt;</i>		
<i>TypeBlob</i>	<i>CosQueryCollection :: Blob</i>	<i>BLOB</i>	<i>BLOB</i>
<i>TypeClob</i>	<i>CosQueryCollection :: Clob</i>	<i>CLOB</i>	<i>BLOB</i>
<i>TypeWclob</i>	<i>CosQueryCollection :: Wclob</i>	<i>NCLOB</i>	<i>BLOB</i>

### 5.1.1 C++ Facade class FieldValueAccess

You can directly work with FieldValue using IDL to C++ mapping, but it is easier to use access operators from class FieldValueAccess.

- `FieldValueAccess::isNil(const FieldValue& fv)` returns `ithrue`, if `fv` is `NUL`, `false` otherwise

```
if (FieldValueAccess::isNil(myRecord[0])) {
    cout << "NIL here" << endl;
}
```

- `FieldValueAccess::setNil(FieldValue& fv)` set `fv` to `NIL`.
- For each field datatype `XXX` exists function `FieldValueAccess::setXXX` and `FieldValueAccess::getXXX` for writing/reading of `FieldValue`. for example:

```
CORBA::Long x = FieldValueAccess::getLong(fv); // read Long from fv
FieldValueAccess::setLong(fv, 1); // set fv to 1.
```

If during reading `fv`, `fv` is `NIL`, then `RC::FieldVvalueIsNull` thrown, if we try to read value of wrong type, then `InvalidFieldTypeException` is thrown.

- `FieldValueAccess::setAsString` and `getAsString` allows to work with string representations of `FieldValue`.
- At least, class `FieldValueAccess` is defined in `$(prefix)/include/CosQueryFacade`, so you must include this file before using `FieldValueAccess`.

```
#include <CosQueryFacade/FieldValueAccess.h>
```

## 5.2 Record, RecordSeq

`Record` type is just a sequence of field values and denotes type for one record of DB. `RecordSeq` is a sequence of records.

IDL definitions:

```
typedef sequence<FieldValue> Record;  
typedef sequence<Record> RecordSeq;
```

The next code fragment shows how to create record from 2 fields: `(2, 'qqq')` of type `(Short, String)`:

```
Record_var record = new Record;  
record->length(2);  
FieldValueAccess::setShort(record[0], 2);  
FieldValueAccess::setString(record[1], "qqq");
```

### 5.2.1 C++ Facade classes `RecordAccess`, `SRecord`

Appropriative Facade class `RecordAccess` defines

- constant - empty record (`RecordAccess::empty()`),
- constant - empty sequence of records (`RecordAccess::emptySeq()`),
- constant - empty binary sequence (`RecordAccess::emptyOctSeq()`)
- static method, which transform record to records sequence with one element. (`RecordAccess::createSeq`)
- family of `addXxx` methods

Yet one Facade class is `SRecord`, which provide “syntax sugar” for records creation.

Let we want to create record with 3 fields: `(3 Short, "aa" String, "bb" String)`.

Using `FieldValueAccess` code fragment, for creating of this record will be looked like:

```

Record_var record = new Record();
record->length(3);
FieldValueAccess::setShort(record[0],3);
FieldValueAccess::setString(record[1],"aa");
FieldValueAccess::setString(record[2],"bb");

```

SRecord provide more comfortable API, which allows to create our record by one string of code:

```

SRecord sr;
sr._short(3)._string("aa")._string("bb");

```

i. e. for all database types Xxx exists method Srecord::\_xxx, which appends field of such type to (\*this) and return one.

### 5.3 OctSeq, RCHeader

In all interfaces UAKGQuery programmer has the choice: to pass data as set of notes or as binary set in RC-coding.

Why use OctSeq – because UAKGQuery defines own coding of database records to binary stream, which much more effective, than GIOP. With UAKG-Query it is possible to achieve performance of data transfer near theoretical maximum.

OctSeq defined in IDL as sequence of bytes:

```

typedef sequence<octet> OctSeq;

```

You can write data to octet stream with help of RCWriter and read with help of RCReader. <sup>2</sup>

About structure of RC byte stream: it consists of header and data section. Full BNF specifications of RC-coding is situated in App3 ???. You need this specifications only in case, when you want to read/write RC-coded sequence outside UAKGQuery access (for example, by python program).

UAKGQuery gives to the application programmer the RCWriter or RCReader class correspondingly for reading/recording of RC-sequences.

UAKGqueryService provides helper classes, which incapsulate low-level operations, it is necessary to know only that RC-sequence consists of header and data; header is defined next way:

```

struct RCHeader
{
    octet version;           // protocol version
    long nRecords;          // number of records in stream
    unsigned long nFields;  // number of fields in one record.
};

```

---

<sup>2</sup>RC means Record Coding.

### 5.3.1 RCReader

RCReader is the class, which number of static methods for reading-recording RC-streams.

So, for reading of RC-coded stream we must first read header, define number of records in stream and then read this records.

Now, let's return to function print RC, which reads RC stream:

```
RCHeader header;
CORBA::ULong pos=0;
RCReader::readHeader(header,pos,octSeq);
out << "Header: nRecords=" << header.nRecords << endl;
out << "Header: nFields=" << header.nFields << endl;
```

As we can see, RCReader::readHeader(header,pos,octSeq) fill header and set pos to offset of first record in octet stream.

```
for(CORBA::Long nRecord=0; nRecord<header.nRecords; ++nRecord){
    for(CORBA::ULong nField=0; nField<header.nFields; ++nField){
        FieldValue_var fv = RCReader::readField(pos,octSeq);
        printField(out,fv);
        out << "|";
    }
    out << endl;
}
```

RCReader::readField read one field and set pos to offset of next record.

You can read all record by one call, i. e. previous code fragment can be rewritten next way:

```
for(CORBA::Long nRecord=0; nRecord<header.nRecords; ++nRecord){
    Record_var record = RCReader::readRecord(pos,octSeq);
    printRecord(out,fv);
}
```

Or just sequence of records in one call:

```
RecordSeq_var recordSeq = RCReader::readRecordSeq(pos,octSeq);
```

Full description of RCReader you can find in API Reference .

### 5.3.2 RCWriter

RCWriter expose API for writing into RC stream. Next code fragment creates RC stream and writes one record to it:

```
OctSeq_var octSeq = new OctSeq;
CORBA::ULong pos;
RCWriter::writeHeader(1,record.length(),pos,octSeq);
RCWriter::writeRecord(record,pos,octSeq);
```

Of course, you can also write by field or by sequence of records.

Note, that write the number of records to header you can after writing of data. I.e. the next code will work:

```
OctSeq_var octSeq = new OctSeq;
CORBA::ULong pos;
RCWriter::writeHeader(1,record.length(),pos,octSeq);
RCWriter::writeRecord(record,pos,octSeq);
RCWriter::writeRecordSeq(recordSeq,pos,octSeq);
RCWriter::writeNRecords(recordSeq.length()+1);
```

## 5.4 RecordDescription

RecordDescription is a description of database record. It is defined via next IDL definitions:

```
/**
 * struct for description of field size.
 * name: name of field in DB.
 * ValueType: field type.
 * size: size of field in bytes. (for strings: include \0, i. e.
 *       for VARCHAR(x) size is x+1
 * precision (have sense only for NUMERIC types) - precision.
 * scale (have sense only for NUMERIC types) - scale, as signed byte.
 */
struct FieldDescription{
    string          name; // name of field
    CosQueryCollection::FieldType    type; // type
    unsigned long   size; // size of field in bytes
    unsigned short precision; // precision (have sense only for NUMBER)
    short          scale; //
};
typedef sequence<FieldDescription> RecordDescription;
```

I.e. if we have RecordDescription, than we know names and types of Data Fields.

Of course, exists Facade class, for easy access to RecordDescription. It's name is RecordDescriptionAccess.

## 6 Evaluation of queries

So, as you know, for evaluation of SQL queries you can use family of methods `QueryManager::evaluate_XXX`.

## 6.1 Queries without parameters

Let's return to our first example of query evaluation:

```
try {  
  
    OctSeq_var octSeq = queryManager->evaluate_rc("select * from tab","SQL92",  
                                                RecordDescriptionAccess::empty(),  
                                                RecordAccess::emptyOctSeq());  
  
}catch(QueryInvalid ex){  
    cerr << "QueryTypeInvalid:" << ex.why << endl;  
}catch(QueryTypeInvalid ex){  
    cerr << "QueryTypeInvalid" << endl;  
}catch(QueryProcessingError ex){  
    cerr << "QueryProcessingError:" << ex.why << endl;  
}
```

Now we know, that:

- Return value is RC-coded octet sequence.
- first parameter - text of SQL query.
- second parameter - flags.
- third parameter - type of query input parameters. In this case we pass empty RecordDescription, because this query has no parameters.
- four - query parameters. In our case: it is empty

Note, that many queries without input parameters exist in typical application, so we define special methods: `evaluate_rc_e` and `evaluate_records_e` for such case.

I.e. previous example we can rewrite in more compact form:

```
try {  
  
    OctSeq_var octSeq = queryManager->evaluate_rc_e("select * from tab","SQL92");  
  
}catch(QueryInvalid ex){  
    cerr << "QueryTypeInvalid:" << ex.why << endl;  
}catch(QueryTypeInvalid ex){  
    cerr << "QueryTypeInvalid" << endl;  
}catch(QueryProcessingError ex){  
    cerr << "QueryProcessingError:" << ex.why << endl;  
}
```

\end{verbatim}

Next question: how differs `\verb|evaluate_records|` family of methods from `\verb|evaluate_rc|` -- very simple, `\verb|evaluate_records|` family of methods accept query parameters and return query results as sequence of records instead RC-coded sequences.

```
\begin{verbatim}
RecordSeq_var records;
try {
    records = queryManager->evaluate_records_e("select * from tab","SQL92");
}catch(QueryInvalid ex){
    cerr << "QueryTypeInvalid:" << ex.why << endl;
}catch(QueryTypeInvalid ex){
    cerr << "QueryTypeInvalid" << endl;
}catch(QueryProcessingError ex){
    cerr << "QueryProcessingError:" << ex.why << endl;
}
\end{verbatim}
```

## 6.2 queries with parameters

Now, let's look at third and fourth parameter of `QueryEvaluator::evaluate`. Sense of third parameter: `RecordDescription` - is description of record, which passed to query as parameter. <sup>3</sup>

4-th parameter - is a record (or records) with values of query parameters.

Let's illustrate this by next example: query is

```
select name from empties where id=:ID
```

Where `:ID` – query parameter with type long.

Example of evaluating such query is below:

```
char* getEmployeeNameById(long id)
{
    RecordDescription paramsDescription;
    RecordDescriptionAccess::appendLong(paramsDescription, "ID");
    Record_var params = new Record;
    params->length(1);
    FieldValueAccess::setLong(params[0],id);
    RecordSeq_var retval = queryManager->evaluate_record(
        "select name from empl where id=:ID","",
        paramsDescription, params);
    if (retval->length()==0) {
        throw IncorrectEmployeeId(id);
    }
}
```

---

<sup>3</sup>We suppose, reader is familiar with concepts of binding variables (or, in some terminology: host variables). If not, please look at [?]

```

    return FieldValueAccess::createString(retval[0][0]);
}

```

Note, that host variables can be used not only for passing information to database but also for retrieving data; for example, with the help of SQL construction `select into`

The family of methods `evaluate_*_inout` is intended for this purpose.

I.e. after next call:

```

queryManager->evaluate_record_inout(
    "select name into :name from emps where id=:id",
    "SQL_92",
    recordDescription,
    record
)

```

value of `name` will be fetched into `record`.

## 7 Query Interface

Using of methods from family `QueryEvaluator::evaluate` does not overlap all needed functionality of database interface. We come to problems with next cases:

1. We can't query large data sets by parts: all data are passed to client during one request.
2. We can't query descriptions of received data set.
3. SQL server fully parse query during each call of `evaluate`.

So, we need more complex interface to use all functionality of relational database. This interface is named `Query`

```

interface Query
{
    ///
    readonly attribute QueryManager query_mgr;

    /**
     *@return text of query.
     */
    readonly attribute string queryText;

    /**

```

```

* return status of query: i.e:
* complete when query is executed, otherwise incomplete
*/
CosQuery::QueryStatus get_status ();

/**
* prepare query for executing.
* if query have no parameters, paramsDescription must be empty
* sequence.
*@param paramsDescription description of query input parameters.
*/
void prepare_query(in RecordDescription paramsDescription)
    raises(CosQuery::QueryProcessingError);

/**
* synonym for prepare_query
**/
void prepare(in RecordDescription paramsDescription)
    raises(CosQuery::QueryProcessingError);

/**
* execute query
*@param octSeq_ RC-coded sequence of input parameters.
* can be empty, if query have no parameters.
**/
void execute_rc(in OctSeq octSeq_)
    raises(CosQuery::QueryProcessingError);

/**
* execute query, and if query has out or inout parameter, then fill
* them
*@param octSeq_ RC-coded sequence of parameters.
* parameter can have in, out and inout modes.
**/
void execute_rc_inout(inout OctSeq octSeq_)
    raises(CosQuery::QueryProcessingError);
///
void execute_records(in RC::RecordSeq recordSeq_)
    raises(CosQuery::QueryProcessingError);
///
void execute_record(in CosQueryCollection::Record record_)
    raises(CosQuery::QueryProcessingError);
///
void execute_records_inout(inout RC::RecordSeq recordSeq_)

```

```

        raises(CosQuery::QueryProcessingError);

///
RecordDescription get_result_description()
    raises(CosQuery::QueryProcessingError,
           QueryNotPrepared);
///
RecordDescription get_parameters_description()
    raises(CosQuery::QueryProcessingError);

///
RC::RecordSeq get_all_parameters_records()
    raises(CosQuery::QueryProcessingError);
///
RC::RecordSeq get_parameters_records(in StringSeq neededFields)
    raises(CosQuery::QueryProcessingError,
           InvalidParameterName);
///
OctSeq get_all_parameters_rc()
    raises(CosQuery::QueryProcessingError);
///
OctSeq get_parameters_rc(in StringSeq fieldNames)
    raises(CosQuery::QueryProcessingError,
           InvalidParameterName);

/**
 * @returns number of fetched rows.
 */
unsigned long get_row_count()
    raises(CosQuery::QueryProcessingError);

/**
 * fetch query result in records.
 * @param numberOfRecords -- number of records to fetch.
 *      0 means, that we want to fetch all records.
 * @param more -- true, if status is incomplete (i.e. we can query
 * more results), otherwise false.
 * @returns fetched rows packed in RC coding to octet sequence.
 */
OctSeq fetch_rc(in unsigned long numberOfRecords, out boolean more)
    raises(CosQuery::QueryProcessingError);

/**
 * synonym for fetch_rc.
 */
OctSeq get_result_rc(in unsigned long numberOfRecords)

```

```

        raises(CosQuery::QueryProcessingError);

/**
 * fetch query result in records.
 * @param numberOfRecords -- number of records to fetch.
 *     0 means, that we want to fetch all records.
 * @param more -- true, if status is incomplete (i.e. we can query
 * more results), otherwise false.
 * @returns fetched records.
 */
RC::RecordSeq fetch_records(in unsigned long numberOfRecords,
                            out boolean more)
    raises(CosQuery::QueryProcessingError);

/**
 * synonym for fetch_records
 */
RC::RecordSeq get_result_records(in unsigned long numberOfRecords)
    raises(CosQuery::QueryProcessingError);

/**
 * skip N records without retrieving.
 * @returns actual number of skipped records.
 */
unsigned long skip(in unsigned long numberOfRecords,
                  out boolean more)
    raises(CosQuery::QueryProcessingError);

/**
 * @return last error.
 * if Query is ok, code in error is 0.
 */
QueryError get_last_error();

/**
 * destroy query, which not longer needed
 */
void destroy();
};

```

To use Query interface, application programmer must perform next steps:

1. create Query using method `QueryManager::createQuery`

2. prepare query, using method `Query::prepare` This method has one parameter: description of query binding variables.
3. execute query, using one of family of `Query::execute` methods.
4. receive data if necessary, using one of methods `Query::fetch_XX`
5. after end of work with query, delete it using method `Query::destroy`

For example, receiving name of employer from id of database record using query interface can be expressed in next code fragment:

```
Query_var query = queryManager->
RecordSeq_var query = queryManager->create_query(
    "select name from empls where id=:ID","");
query->prepare(paramsDescription);
query->execute_record(params);
RecordSeq_var retval = query->fetch_rc(0);
query->destroy();
```

## 7.1 Prepared Query

You can prepare query once and then execute it many times with different parameters. This can be more effective then creating new query for each request because parsing and passing of parameters description are performed only once.

<sup>4</sup>

Typical using of this technique can be illustrated by next code fragment:

```
class EmployeeManager
{
    Query id2nameQuery_;

    ....

public:

    char* getNameById();

};

void EmployeeManager::init()
{
    ....
    id2nameQuery_ = qm_->create_query("select name from empls where id=:id","");
    id2nameQuery_->prepare(paramsDescription);
    ....
}
```

---

<sup>4</sup>Note, that it is possible only inside one transaction

```

}

char* EmployeeManager::getNameById(CORBA::Long id)
{
    Record params(1);
    FieldValueAccess::setLong(params[0],id);
    id2nameQuery_ ->execute_record(params);
    .....
}

EmployeeManager::~EmployeeManager()
{
    .....
    id2nameQuery_ ->destroy();
    ....
}

```

I.e. we keep creating and initialization of query in section of class initialization, destroying of query in destructor and execution of query we put into function, which do real work.

## 7.2 Fetching data via Iterator pattern

Another feature of query interface – API for sequential receiving of data sets by parses. (According to *Iterator* design pattern).

Let's look on next example:

```

CORBA::Boolean more=true;
while(more)
{
    OctSeq_var rc = fetch_rc(chunkSize,more);
    .....
    do something;
}

```

i. e. `fetch` return `chunkSize` or less records and set `more` to false if we receive all records of query result set.

## 7.3 Receiving of result description

We can receive description of query output use method `Query::get_result_description()`;

The next code fragment returns our description of table, set by user:

```

char tablename[MAX_TNAME_LEN];
cout << "enter table name:";
cout.flush();

```

```

cin.getline(tablename,MAX_TNAME_LEN);
ostrstream ostr;
ostr << "select * from " << tablename;
Query_var query = queryManager->create_query(ostr.str(),"");
ostr.rdbuf()->freze(0);
RecordDescription_var tableDescription = query->get_result_description();

```

## 8 QueryManager interface

### 8.1 Sequences

NAME

`get_next_sequence_value` - return next value from sequence.

SYNOPSIS

```
long get_next_sequence_value(in string sequence_name);
```

DESCRIPTION

The function `get_next_sequence_value` get next value from sequence `sequence_name`.

RETURN VALUE

`get_next_sequence_value` return next value of sequence `sequence_name`.

### 8.2 Properties

NAME

`get_property` - return value of property.

SYNOPSIS

```
string get_property(in string property_name) raises(PropertyNotDefined);
```

DESCRIPTION

The function `get_property` return value set to property `property_name` if such property defined and assigned to some value. Enabled properties :

- collection

if property `collection` set to "yes" - then this QueryManager support collections.

RETURN VALUE

`get_property` return value of property if such exist and have a value, else function raise `PropertyNotDefined` exception.

## 9 Large Objects (LOBs)

Features of operating with large objects of database (LOB) consists in:

- three interfaces - Blob, Clob, Wclob
- three methods of Query interface (create\_blob, create\_clob, create\_wclob)

Blob, Clob, Wclob can be parameters of query. Objects of these types can be obtained from string returned.

To accomplish query to database, parameter of which is LOB, do not possible to use 'evaluate' mechanism. Query is single which has methods to create LOB to write. All LOB obtained as a result of query are read-only.

Blob, Clob and Wclob interfaces are analogous. They differ one from other by data types only.

- Blob is meant for operating with binary arrays, an unit of which is octet byte, and UAKGQuery::OctSeq as data type is used;
- Clob is meant for operating with character type data CORBA::String is used;
- Wclob - for unicode, CORBA::WString is used;

Let us consider LOB using Blob:

```
interface Blob {
    ///
    unsigned long length() raises(CosQuery::QueryProcessingError);
    ///
    UAKGQuery::OctSeq fetch_chunk(in unsigned long chunkSize
        , out boolean more)
        raises(CosQuery::QueryProcessingError, ForWritingOnly);
    ///
    void put_chunk(in UAKGQuery::OctSeq data)
        raises(CosQuery::QueryProcessingError, ForReadingOnly);
};
```

Method **length** return the length of LOB in units correspondent, (octet byte number for Blob, number of characters for Clob, number of unicode characters for Wclob). All dimensions are represented in these units. Method **fetch\_chunk** can be called for LOB, which has been obtained as a result of select-type query. It provides mechanism to obtaining of real LOB data. **chunkSize** argument determine maximal required size of readable part of LOB. At that argument **more=true** signify there are data for reading still. Null **chunkSize** value means LOB must be readed wholly. Method **put\_chunk** can be called for LOB having been created via one of Query's methods create\_<B|C|Wc>lob only. Parameter **data** represents actual data to write. On calling **fetch\_chunk** and **put\_chunk** in erroneous context an exception corresponding (ForWritingOnly, ForReadingOnly) will be thrown, pointing that the current LOB is write-only or read-only. Query's methods

```

/**
 * request Blob for filling query parameters
 *@returns empty Blob for writing only.
 */
CosQueryCollection::Blob create_blob()
    raises(CosQuery::QueryProcessingError);

/**
 * request Clob for filling query parameters
 *@returns empty Clob for writing only.
 */
CosQueryCollection::Clob create_clob()
    raises(CosQuery::QueryProcessingError);

/**
 * request Wclob for filling query parameters
 *@returns empty Wclob for writing only.
 */
CosQueryCollection::Wclob create_wclob()
    raises(CosQuery::QueryProcessingError);

```

returns LOBs of type corresponding to write and following binding together with Query's parameters. This is example of reading:

```

CORBA::String_var sql = CORBA::string_dup("select b from blob_test");
RecordSeq_var rs = new RecordSeq();
CORBA::Boolean more = true;
try {
    Query_var q = queryManager->create_query(sql, "");
    q->prepare(RecordDescriptionAccess::empty());
    q->execute_records(RecordAccess::emptyRecordSeq());
    rs = q->fetch_records(1, more);
    if (rs->length() > 0) {
        Blob_var bl;
        try {
            bl = FieldValueAccess::getBlob(rs[0][0]);
        } catch (const InvalidFieldTypeException& ) {
            cerr << "InvalidFieldTypeException" << endl;
            return;
        } catch (const FieldValueIsNull& ) {
            cerr << "FieldValueIsNull" << endl;
            return;
        }
        FILE* f = fopen("file_from_blob", "wb");
        OctSeq_var octSeq;
        long n = 0;
        for (more=true; more; n++){

```

```

        octSeq = bl->fetch_chunk(chunk_size, more);
        fwrite(octSeq->get_buffer(), 1, octSeq->length(), f);
    }
    fclose(f);
} else {
    cerr << "In database no records" << endl;
}
q->destroy();
} catch(const QueryTypeInvalid& ex) {
    cerr << "QueryTypeInvalid" << endl;
    return;
} catch(const QueryInvalid& ex) {
    cerr << "QueryInvalid" << endl;
    return;
} catch(const QueryProcessingError& ex) {
    cerr << "QueryProcessingError" << endl;
    cerr << ex.why << endl;
    return;
}
}

```

And this is example of writing:

```

UAKGQuery::OctSeq_var octSeq = new UAKGQuery::OctSeq();
CORBA::ULong chunk_size;

...// fill octSeq and choose chunk_size

RecordDescription_var recordDescription = new RecordDescription;
recordDescription->length(1);
recordDescription[0].name=CORBA::string_dup(":param");
recordDescription[0].type=TypeBlob;

RecordSeq_var recordSeq = new RecordSeq;
recordSeq->length(1);
recordSeq[0].length(1);
try {
    Query_var q = queryManager->create_query(
        "insert into blob_test(b) values(:param)", "");
    q->prepare(recordDescription);
    Blob_var bl = q->create_blob();
    if (chunk_size == 0) {
        bl->put_chunk(octSeq.in());
    } else {
        CORBA::ULong sentLength = 0;
        CORBA::ULong leftLength = 0;
        const CORBA::Octet* curr;
        while (sentLength < octSeq.length()) {

```

```

        curr = octSeq.get_buffer() + sentLength;
        leftLength = octSeq.length() - sentLength;
        if (leftLength >= chunk_size) {
            bl->put_chunk(OctSeq(chunk_size, chunk_size, curr, 0));
            sentLength += chunk_size;
        } else {
            bl->put_chunk(OctSeq(leftLength, leftLength, curr, 0));
            sentLength = octSeq.length();
        }
    }
}

FieldValueAccess::setBlob(recordSeq[0][0], bl);
q->execute_records(recordSeq);
q->destroy();
} catch(const QueryTypeInvalid& ex) {
    cerr << "QueryTypeInvalid" << endl;
    return;
} catch(const QueryInvalid& ex) {
    cerr << "QueryInvalid" << endl;
    return;
} catch(const QueryProcessingError& ex) {
    cerr << "QueryProcessingError" << endl;
    cerr << ex.why << endl;
    return;
}
}

```

## 10 Collections

### 10.1 Introduction

Query Collection expose high-level object model, intendent for work with data sets, organized as so-called collections.

Using of Query Collections allow to reduce low-level SQL programming for accessing and modifying of data: all typical operations are implemented in collections objects.

### 10.2 General description

#### 10.2.1 “Phisycal sense” of collection

What is collection inside - just a set of SQL sentences for reading, writing and modification of data, which are evaluated during call of collection object methods.

Example: if you want to select some data from one table, you use some SQL sentence which have form:

```
SELECT <select-part> FROM <from-part> WHERE <where-part>
```

So this SQL sentence defines some data set, which can be described by parts of our SQL sentence. Imagine now, that we want to retrieve from this data set all records, which satisfy some condition. To do it we can use such methods from typical collection interfaces as `retrieve_by_filter`; in SQL we must evaluate SQL sentence, which looks as

```
SELECT <select-part> FROM <from-part> WHERE <where-part> AND <condition>
```

Method `update_by_filter` (intendent to update data, which satisfy some condition ) causes evaluating of next SQL sentence:

```
UPDATE <select-part> SET <set-part> WHERE <where-part> AND <filter>
```

Note, that `<set-part>` can be automatically deduced from `<select-part>`.

I. e. during call of collection methods appropriate SQL sentences are built and evaluated.

So, what is the query collection itself: object, which simply keeps and evaluates set of SQL sentences for data access and modifying.

In more formal terms we can say, that `UAKGCollection` is defined by:

- Set of data fields (i. e. `SELECT` part of SQL sentence)
- Data Source (i. e. `FROM` part of SQL sentence)
- Query condition (i. e. `WHERE` part of SQL sentence)
- Ordering (i. e. `ORDER-BY` part of SQL sentence)

### 10.3 Steps for using collections interfaces

For work with query collections, application programmer must perform the next steps:

- create `Collection` using appropriate methods of `QueryManager`.
- Using `UAKGCollection` you can query or modify data in it, or receive `Iterator` for sequential access to data set.
- Before end of work it is necessary to clear server resources, associated with `UAKGCollection` by call of `UAKGCollection::destroy`

### 10.4 Creation of Query Collections

`UAKGQuery` supports two types of collections: `UAKGCollection` and `UAKGKeyCollection`.

`UAKGCollection` represents data set, `UAKGKeyCollection` - dataset with unique keys.

Exists 2 ways of creating `UAKGCollection`:

- by SQL sentence
- by set of field names and conditions. (i. e. by parts)

for example, let's look at next two code fragments:

```
UAKGCollection_var collection = queryManager->create_collection_by_parts(
    "F1,F2", "UAKGTEST", "F1=1", "F2");
```

or

```
UAKGCollection_var collection = queryManager->create_collection_by_query(
    "select F1,F2 from UAKGTEST where F1=1 order by F2");
```

This 2 fragments give us identical result: collection which consists of fields F1 and F2 of records in UAKTEST, ordered by F2, where F1 is 1.

Parameters of `create_collection_by_fields` in first code fragment are:

1. set of fields: SELECT part of SQL sentence
2. data source: FROM part.
3. select condition: WHERE part.
4. ordering (optional) : ORDER part.

## 10.5 Data Access

### 10.5.1 Receiving data with the help of Iterator's interface

You can use `Iterator` concepts to retrieve data. The steps for work with iterator are described below:

1. receive `Iterator`, by call of `create_iterator` method of collection.
2. Use `Iterator` API for navigation across data set.
3. destroy `Iterator`

Example:

```
UAKGIterator_var iterator_ = collection->create_iterator();
Boolean more = true;
while( more )
{
    OctSeq_var octSeq_ = iterator->fetch_rc(50, more);
    printRC( octSeq_ );
}
iterator->destroy();
```

Interface `Iterator` expose next methods:

- `fetch_rc( ULong n, Boolean& more )` Read `n` records as byte stream in RC-coding
- `fetch_records( ULong n, Boolean& more )` Read `n` records and return it as `RecordSequence`.
- `skip( ULong n, Boolean& more )` Skip `n` records without retrieving of data.

### 10.5.2 Retrieving data via collection methods

You can read data from `UAKGCollection` directly, using next collection methods:

- `retrieve_by_filter( const char* where_filter )`
- `retrieve_by_pattern( const Record& pattern )`

Both of those methods return RC-coded byte stream, which contains records from data set collection filtered by parameter. First method filters data set before retrieving by logical expression in `where_filter`, second – by principle of pattern matching (see 10.5.3).

Example:

```
UAKGCollection_var collection_ = queryManager->create_collection_by_query(
    "select F1,F2 from UAKGTEST where F1=1 order by F2");
OctSeq_var octSeq_ = collection_->retrieve_by_filter("F2='test'");
printRC( octSeq_ );
collection_->destroy();
```

This code fragment will print all records of created collection, where `F2` is equal to string `'test'`. (I. e. all records in result will have form `(1,'test')`).

### 10.5.3 Pattern matching

Concept of pattern matching is often used in `UAKGQuery Service` for filtering data sets: for example, method `retrieve_by_pattern` retrieves records which match pattern, passed as argument of this method. Which means: pattern matching concept is derived from `QBE (Query By Example)` concept – in result set we receive records, which are "the same" as pattern, exclude fields, which set in pattern as empty.

More formal:

Pattern is a record (i. e. have type `Record`). Fields of pattern are used for filtering data sets. Pattern matching for collection is defined as following:

- Structure of pattern record must be equal to structure of records in queuing dataset (i.e. number of fields and types must be equal)

- Record  $r$  match pattern  $p$  iff
  1.  $\forall i \in 0 \dots \text{length}(r) : p[i]! = \text{NIL} \rightarrow p[i] \text{ match } r[i]$  (i. e. for all field indexes, appropriate fields must match if pattern field is not NULL).
  2. For fields  $p_i, r_i, p_i \text{ match } r_i$  iff
    - (a)  $\text{type}(r_i) == \text{string} \rightarrow (r_i \text{ LIKE } p_i)$
    - (b)  $\text{type}(r_i)! = \text{string} \rightarrow (r_i == p_i)$

I. e. 2 fields match, if they are identical for non-string types, or if LIKE matching is met for string types.

Practical example:

Suppose we have table UAKGTEST with two fields: F1 of type NUMBER and F2 of type VARCHAR2. Then, we created collection which works with records of this table. Then, for retrieving all records from this collection with F1=5 programmer must perform next steps:

- Form pattern as a 2 field record , where first field must be 5, second - NULL . (i. e. second field will not participate in matching).
- get needed records as result for `retrieve_by_pattern` with pattern as parameter.

```
Record_var pattern = new Record;
pattern->length(2);
FieldValueAccess::setLong(pattern[0],5);
FieldValueAccess::setNull(pattern[1]);
OctSeq_var octSeq = collection_->retrieve_by_pattern(pattern);
printRC(octSeq);
```

## 10.6 adding, updating and deleting of data

Of course, except retrieving of data, you can add, modify or delete collection items.

List of appropriate methods:

1. Adding data:
  - `add_record( const Record& element )` – add one record to collection
  - `add_records( const RecordSeq& elements )` – add sequence of records to collection.
  - `add_rc( const OctSeq& rc )` – add sequence of records, coded in RC stream.

Note, that during adding of records no checking of belonging added data to target collection is performed. For example, let's look at next code fragment:

```

UAKGCollection_var collection=qm->create_collection(
    "select * from emp where deptno=22"
);

Ulong n=collection->number_of_records();
cout << "now number of records is:" << n << endl;
RecordDescription_var rd = collection->get_record_description();
Record_var record=CosQueryFacade::RecordAccess::createRecordByDescription(rd);
CosQueryFacade::RecordAccess::setShortByName(record.inout(),"EMPNO",11,rd);
CosQueryFacade::RecordAccess::setShortByName(record.inout(),"DEPTNO",11,rd);
collection->add_record(record);
n=collection->number_of_records();
cout << "now number of records is:" << n << endl;

```

After successful execution of this fragment one item will be added to emp table, but number of elements in collection will not change.

## 2. Updating of data:

- `update_by_pattern( const Record& newRecord, const Record& pattern )`  
 – set to `newRecord` records, which matching pattern `pattern`. For example, next code fragment:

```

SRecord sr1, sr2;
collection->update_by_pattern(
    sr1._short(1)._short(2).in(), sr2._nil()._short(2)
)

```

- `update_by_filter( const Record& newRecord, const char* filter )`  
 - set to `newRecord` all records, which satisfy logical condition in `filter`.

For example, next code fragment:

```

collection->update_by_filter(sr._short(1)._short(2),"x2=2");

```

will set to (1,2) all records in collection for which `x2=2`

## 3. Deleting data:

- `remove_record( const Record& record )` - remove records, which are equal to given record.
- `remove_records_by_filter( const char* filter )` - remove records by filter.
- `remove_records_by_pattern( const Record& pattern )` - remove records by pattern.
- `remove_all_records()` - remove all records from collection.

Example:

```

UAKGCollection_var collection_ = queryManager->create_collection_by_query(
    "select F1,F2 from UAKGTEST where F1=1 order by F2");
Record_var inpRecord = new Record;
inpRecord->length(2);
FieldValueAccess::setLong( inpRecord[0], 5);
FieldValueAccess::setString( inpRecord[1], "test" );
collection_->add_record(inpRecord);
FieldValueAccess::setNull( inpRecord[1] );
OctSeq_var octSeq_ = collection_->retrieve_by_pattern(inpRecord);
printRC( octSeq_ );
collection_->remove_records_by_pattern( inpRecord );
collection_->destroy();

```

## 10.7 Collection queries

The shown up set of methods is not always sufficient for comfortable access to data collection. Sometimes we need to perform more complicated actions: for example change set of requested fields or receive some data from linked table.

For this purpose family of `evaluate_xxx` methods of `UAKGCollection` is intended. (i. e. `collections` implements interface `QueryEvaluator`).

For example, next query:

```
result = collection->evaluate_rc_e("select x1,x3 from @ where x1=x2");
```

will fetch in result fields `x1` and `x2` for records, in which `x1=x3`.

Next code fragment:

```
collection=queryManager->create_collection("select * from orders");
result=collection->evaluate_rc(
    "select @,CUSTOMERS.NAME from @,CUSTOMERS where CUSTOMER.ID=customer_id
    );
```

will retrieve all fields of order table and names of customers from linked table.

Next query:

```
result=collection->evaluate_rc("select @ from @ where not @")
```

will invert collection (i. e. select all records which are situated at the same data source, but not belong to original collection).

Now, lets define semantics of our extending of SQL by `@` :

If collection is based on SQL expression in form:

```
select <select-part> from <from-part> where <where-part>
```

and we pass in `evaluate_xxx` expression:

```
select @,<new-select-part>
from @,<new-where-part> [where <new-where-part>]
```

than result expression will have form:

```
select <select-part>,<new-select-part>
from <from-part>,<new-from-part>
where (<where-part>) AND (<new-where-part>)
```

Suppressed SQL sentences is handled in less or more equal way.

### 10.7.1 Limitations

- Collection can evaluate only SELECT queries without HAVING\_BY and GROUP\_BY clauses. Full grammar of Collection SQL is listed in 14

### 10.7.2 List of methods

- `evaluate_rc( const char* queryText, const char* queryFlags, const RecordDescription& recordDescription, const OctSeq& params)`
- `evaluate_rc_inout(const char* queryText, const char* queryFlags, const RecordDescription& recordDescription, OctSeq& params)`
- `evaluate_record( const char* queryText, const char* queryFlags, const RecordDescription& recordDescription, const Record& params )`
- `evaluate_records_inout(const char* queryText, const char* queryFlags, const RecordDescription& recordDescription, RecordSeq& params)`
- `evaluate_records(const char* queryText, const char* queryFlags, const RecordDescription& recordDescription, const RecordSeq& params)`

## 10.8 SubCollection|

As can be deduced from name of sections: you can query some subset of collection in new collection (so-called subcollection).

Application programmer must perform next steps to use subcollection technique:

1. Creation of SubCollection

- `create_subcollection( const char* subquery )` In this method `subquery` is expression on the same language, as for `evaluate_xxx` family of collection methods. Created collection is result of evaluating such query as collection. Note, that if data-source (i.e. where-part of subquery) contains multiple tables, then resulting collection is read-only.
- `create_subcollection_by_pattern( const Record& pattern )` The resulting collection is just subset of records of original collection, which match pattern `pattern`.

2. Work with received subcollection, using UAKGCollection API

3. delete `SubCollection` using method `destroy()`

Example:

```
UAKGCollection_var collection_ = queryManager->
    create_collection_by_fields("tname, tabtype", "tab", "1=1", "");
UAKGIterator_var iterator = collection_->create_uakgiterator();
Boolean more;
RecordSeq_var recordSeq = iterator->fetch_records(0, more);
FieldValueAccess::setNull(recordSeq[0][0]);
iterator->destroy();
UAKGCollection_var new_collection_ = collection_->
    create_subcollection_by_pattern(recordSeq[0]);
iterator = new_collection_->create_uakgiterator();
recordSeq = iterator->fetch_records(0, more);
printRecordSeq(cout,recordSeq.in());
iterator->destroy();
new_collection_->destroy();
collection_->destroy();
```

## 10.9 UAKGKeyCollection

UAKGKeyCollection is a specialization of UAKGCollection with property “have primary key”.<sup>5</sup>

Additional methods provide API for accessing, modifying and deleting elements by keys.

### 10.9.1 Creation of UAKGKeyCollection

for creation of `KeyCollection` you can use 2 methods of `UAKGQueryManager`:

- `create_key_collection_by_parts` – which is create key collection, by appropriate parts of SQL sentence and yet in additional part: list of field names, of which key is consists. Example:

---

<sup>5</sup>i. e. key, which is unique for each record in collection

```
UAKGKeyCollection_var collection = queryManager->
    create_key_collection_by_parts("F1,F2", "UAKGTEST", "F1=1", "F2", "F1");
```

- `create_key_collection_by_query` – which is create key collection by SQL query, with yet one our extension of SQL: `WITH KEY` clause.

```
UAKGKeyCollection_var collection = queryManager->
    create_key_collection_by_query(
        "select F1,F2 from UAKGTEST where F1=1 order by F2 with key F1"
    );
```

`WITH KEY` clause can be used only for creation of key collection and SQL sentence with it must have following syntax:

```
SELECT <selection> <table-expr> WITH KEY <selection>
```

As in SQL92 key can be compound:

```
UAKGKeyCollection_var collection=queryManager->
    create_key_collection_by_query(
        "select * from X with key x1, x2");
```

-these two parts of code are equivalent, they create the collection `UAKGTest`, where `F1` is the primary key. As we can see, we have one more SQL- clause `WITH KEY`. It may be used only to get collection with keys, enclosure syntax with this clause is following:

```
SELECT <selection> <table-expr> WITH KEY <selection>
```

As in SQL92, the sequence of margins can create the key too:

```
UAKGKeyCollection_var collection = queryManager->
    create_key_collection_by_query(
        "select * from X with key x1,x2");
```

Note, that specification of correct keys is business of application programmer : `UAKGQueryService` use this information, but does not do any checking for accordance with real structure of database.

### 10.9.2 Methods of `UAKGKeyCollection`

`UAKGKeyCollection` provide next methods, for retrieving, updating and deleting elements by keys:

- `retrieve_record_with_key(const Record& key)` - retrieve record with key `key`.
- `retrieve_records_with_keys(const OctSeq& keys)` - retrieve sequence of records with accordance of `keys`

- `update_record_with_key(const Record& newRecord, const Record& key)`  
– set to `newRecord` element with key `key`.
- `update_records_with_keys(const OctSeq& records)` – update records, with the same keys, as appropriate records in argument.
- `remove_records_with_keys(const OctSeq& keys)` – remove records with keys.

Also 2 helper methods are provided by `UAKGKeyCollection`:

- `get_key_description()` – return description of collection key
- `extract_keys(const OctSeq& records)` – extract keys from sequence of records.

Example:

```
UAKGKeyCollection_var collection_ = queryManager->create_key_collection_by_query("select F
Record_var inpRecord_ = new Record;
inpRecord_->length(1);
FieldValueAccess::setString(inpRecord_[0], "test");
collection_->remove_record_with_key( inpRecord );
collection_->destroy();
```

## 10.10 Using of `UAKGCollectionListener`

It becomes common and useful technique to organize program coupling via notification about important events in life of some service. On this purpose `UAKGCollection` provide such mechanism of "Listeners" - user can add to collection own implementation of `UAKGCollectionListener` callback interface, which would be called during performing of collection actions.

```
interface UAKGCollectionListener
{
    // called, when elements are added:
    void elements_added(in OctSeq elements);

    //called, when elements updated
    void elements_updated(in OctSeq prev_elements,
                        in OctSeq new_elements);

    //called, when elements removed:
    void elements_removed(in OctSeq elements)

    //called, when all elements in collection are removed
    void all_elements_removed();
```

```

    //called, when collection is destroyed
    void collection_destroyed();
}

```

Application programmer can implement this interface and bind it with collection for receiving events via method:

```

unsigned long UAKGCollection::add_listener(
    in UAKGCollectionListener listener,
    in unsigned short eventMask);

```

Method returns number, which identify passed listener from collection side. This number can be used for unbinding listener from collection with help of method:

```

unsigned long UAKGCollection::remove_listener(in unsigned long listenerIndex);

```

At last, `eventMask` is a bit mask of events, which listener want to receive.

Using notification technique remember, that cost of collection data for notifying can be high, and that synchronized calls of callback functions on each event are not scalable technique. If you have situation, when N service clients must receive notifications, than better do not register N listeners but create additional element of infrastructure, which receive notification and send it to clients, using asynchronics techniques (for example, via CORBA Event Service).

## 10.11 Limitations of using UAKGQuery collections interfaces

For present tim , one instance of collection can be used in only one instance of transaction at the same time.

(I. e. concurrent access to collection from different transactions is not safe).

## 11 Transactions

Two models of transactions are implemented in UAKQueryService:

1. XA transactions, which use XA resource of underlying database and XA monitor of ORBacus Transaction Service.
2. Own transaction manager, which shown as resource to ORB OTS and perform, all operations using specific transaction API of underlying database.

XA transaction manager is intended to be used in case, when your applications work in XA environment. UAKGQuery transaction manager is more light and in some cases can be more effective.

Now, let's describe using of this two transaction models.

### 11.0.1 XA transactions

Using of XA transactions are initiated by setting next flags in command options of service:

- `ORACLE_XA=<xa-open-string>` - for Oracle
- `INTERBASE_XA=<xa-open-string>` - for Interbase.

Where `<xa-open-string>` is XA string, with parameters of database connection. For detail description of XA string, look at documentation of you database:

- Oracle: [http://technet.oracle.com/doc/server.815/a68003/01\\_app1x.htm#619504](http://technet.oracle.com/doc/server.815/a68003/01_app1x.htm#619504)
- Interbase InterBase Programming Guide.

During work with XA transactions you must meet next limitations:

1. As far as connections to database is performed by XA monitor, then only one global XA connection (i. e. login,password,db) can be used by one UAKGQuery Server. Parameters `username` and `password` of `DBConnectionManager::createQueryManager` are ignored. So, if you want to achieve one-time work of few different connections to one databases in XA mode, than you must start few copies of `UAKGQueryService`.
2. You can't use DDL statements in XA application.
3. You can't call UAKGQuery methods outside of transaction context.
4. You must use `tread_per_request` threading mode.

### 11.0.2 UAKG OTS Transactions

UAKG OTS transaction mode is used by default.

Unlike XA transactions, UAKGQ transactions does not touch parameters of db connections. All, what you need is call methods of `UAKGQuery` in transactional context. You can mix transactional and non-transactional calls: in this case all non-transactional calls will be mapped to short local transactions.

## 11.1 Typical Usage Of Transactions

Next code fragment illustrate typical usage of OTS:

```
CORBA::Object_var obj = orb->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var current =
    CosTransactions::Current::_narrow(obj);
```

```

current->begin();
try {

    // do something with db:
    queryManger->evaluate_query(query1,"SQL92",query1ParamsDescriptions,
                                query1Params);
    queryManger->evaluate_query(query2,"SQL92",query2ParamsDescriptions,
                                query2Params);

    current->commit();

}catch(const QueryProcessingError& ex){

    current->rollback();

}catch(...){
    cerr << "Fatal: unknown exception";
    current->rollback();
}

```

I.e. typical usage of transaction: to achieve atomity of sequence of operations: all operations in sequence will be successful or all will be complete rollbacked.

In details, transaction mechanisms are described in: [3] , [7].

## 12 IDL definitions

### 12.1 CosQueryCollection

```

#ifndef __COSQUERYCOLLECTION_IDL
#define __COSQUERYCOLLECTION_IDL

/*
 * module CosQueryCpllection.
 * writeln from specifications of OMG froup for CORBA Query Service.
 * (C) Ruslan Shevchenko <Ruslan@Shevchenko.Kiev.UA>
 * (C) GradSoft <info@gradsoft.com.ua>
 * 1998, 1999, 2000, 2001
 * $Id: ProgrammingGuide_eng.tex,v 1.25 2002/01/29 13:24:48 srkr Exp $
 */

#ifndef __CosQueryIDLConfigV2_idl
#include <CosQueryIDLConfigV2.idl>
#endif

#ifdef HAVE_ORB_IDL

```

```

#include <orb.idl>
#endif

#pragma prefix "gradsoft.kiev.ua"

/**
 *
 **/
module UAKGQuery
{
    ///
    /**
     * UAKGQuery sequence of octets
     **/
    typedef sequence<octet> OctSeq;
};

#include <CosQuery.idl>

#pragma prefix "omg.org"

/**
 * module CosQueryCpllection.
 * data definitions for CORBA Query Service
 * writeln from specifications of OMG group .
 * (C) Ruslan Shevchenko <Ruslan@Shevchenko.Kiev.UA>
 * 1998, 1999, 2000
 **/
module CosQueryCollection {

    ///
    exception ElementInvalid {};
    ///
    exception IteratorInvalid {};
    ///
    exception PositionInvalid {};

    /**
     * possible DB field types
     **/
    enum FieldType {
        ///

```

```
TypeBoolean,  
///  
TypeChar,  
///  
TypeOctet,  
///  
TypeShort,  
///  
TypeUShort,  
///  
TypeLong,  
///  
TypeULong,  
///  
TypeFloat,  
///  
TypeDouble,  
///  
TypeString,  
///  
TypeObject,  
///  
TypeSmallInt,  
///  
TypeInteger,  
///  
TypeReal,  
///  
TypeDoublePrecision,  
///  
TypeCharacter,  
///  
TypeDecimal,  
///  
TypeNumeric,  
///  
TypeDateTime,  
///  
TypeRaw,  
///  
TypeWString,  
///  
TypeBlob,  
///  
TypeClob,  
///  

```

```

    TypeWclob
};

/**
 * decimal field.
 **/
struct Decimal {
    /**
     * precision of number.
     **/
    long precision;
    /**
     * scale of number
     **/
    long scale;
    /**
     * valus in BCD format.
     **/
    sequence<octet> value;
};

/**
 * type, corresponding to DATE field.
 * (all values are start from 1)
 **/
struct DateTime {
    ///
    short year;
    ///
    octet month;
    ///
    octet day;
    ///
    octet hour;
    ///
    octet minute;
    ///
    octet second;
};
    ///
    exception ForReadingOnly {};

    ///
    exception ForWritingOnly {};

/**

```

```

*
**/
interface Blob {
    ///
    unsigned long length() raises(CosQuery::QueryProcessingError);
    ///
    UAKGQuery::OctSeq fetch_chunk(in unsigned long chunkSize
        , out boolean more)
        raises(CosQuery::QueryProcessingError, ForWritingOnly);
    ///
    void put_chunk(in UAKGQuery::OctSeq data)
        raises(CosQuery::QueryProcessingError, ForReadingOnly);
};

/**
*
**/
interface Clob {
    ///
    unsigned long length() raises(CosQuery::QueryProcessingError);
    ///
    string fetch_chunk(in unsigned long chunkSize, out boolean more)
        raises(CosQuery::QueryProcessingError
            , ForWritingOnly);
    ///
    void put_chunk(in string data) raises(CosQuery::QueryProcessingError
        , ForReadingOnly);
};

/**
*
**/
interface Wclob {
    ///
    unsigned long length() raises(CosQuery::QueryProcessingError);
    ///
    wstring fetch_chunk(in unsigned long chunkSize, out boolean more)
        raises(CosQuery::QueryProcessingError
            , ForWritingOnly);
    ///
    void put_chunk(in wstring data) raises(CosQuery::QueryProcessingError
        , ForReadingOnly);
};

```

```

/**
 * what can be not null value in DB:
 **/
union Value switch(FieldType) {
    ///
    case TypeBoolean: boolean b;
    ///
    case TypeChar: char c;
    ///
    case TypeOctet: octet o;
    ///
    case TypeShort : short s;
    ///
    case TypeUShort : unsigned short us;
    ///
    case TypeLong : long l;
    ///
    case TypeULong : unsigned long ul;
    ///
    case TypeFloat : float f;
    ///
    case TypeDouble : double d;
    ///
    case TypeString : string str;
    ///
    case TypeObject : Object obj;
    ///
    case TypeSmallInt : short si;
    ///
    case TypeInteger : long i;
    ///
    case TypeReal : float r;
    ///
    case TypeDoublePrecision : double dp;
    ///
    case TypeCharacter : string ch;
    ///
    case TypeDecimal : Decimal dec;
    ///
    case TypeNumeric : Decimal n;
    ///
    case TypeDateTime : DateTime dt;
    ///
    case TypeRaw : sequence<octet> raw;

```

```

        ///
        case TypeWString : wstring wstr;
        ///
        case TypeBlob    : Blob bl;
        ///
        case TypeClob    : Clob cl;
        ///
        case TypeWclob   : Wclob wcl;

};

///
typedef boolean Null;

/**
 * this union represent one field in DB
 **/
union FieldValue switch(Null){
    case FALSE : Value v;
};

/**
 * one record in DB
 **/
typedef sequence<FieldValue> Record;

/**
 *
 **/
typedef string Istring;

};

#endif

```

## 12.2 CosQuery

```

#ifndef __COSQUERY_IDL
#define __COSQUERY_IDL

/*
 * module CosQuery.
 * from specifications of OMG group for CORBA Query Service.
 * (C) Ruslan Shevchenko <Ruslan@Shevchenko.Kiev.UA>, 1998, 1999, 2000

```

```

* (C) GradSoft, 2001
* $Id: ProgrammingGuide_eng.tex,v 1.25 2002/01/29 13:24:48 srkr Exp $
*/

#include <CosQueryCollection.idl>

#pragma prefix "omg.org"

/**
 * CosQuery: legacy definitions from OMG Query Service.
 **/
module CosQuery {

    ///
    exception QueryInvalid
    {
        ///
        string why;
    };

    ///
    exception QueryProcessingError
    {

        ///
        string why;
    };

    ///
    exception QueryTypeInvalid { };

    ///
    enum QueryStatus
    {
        ///
        complete,
        ///
        incomplete
    };

};

#endif

```

## 12.3 UAKGQuery

```
#ifndef __UAKGQUERY_IDL
#define __UAKGQUERY_IDL

/*
 * IDL Definitions for UAKGQueryService
 * (C) Ruslan Shevchenko <Ruslan@Shevchenko.Kiev.UA>, 1998,1999, 2000, 2001
 * (C) GradSoft 2001
 * $Id: ProgrammingGuide_eng.tex,v 1.25 2002/01/29 13:24:48 srkr Exp $
 */

#ifdef CORBA_HAVE_OTS
#ifndef __COSTRANSACTIONS_IDL
#include <CosTransactions.idl>
#endif
#endif

#ifndef __COSQUERY_IDL
#include <CosQuery.idl>
#endif

#ifndef __RC_IDL
#include <RC.idl>
#endif

#pragma prefix "gradsoft.kiev.ua"

/**
 * UAKGQuery module
 * (GradSoft-specific type of UAKGQuery implementation).
 */
module UAKGQuery
{

    ///
    typedef sequence<string> StringSeq;

    /**
     * struct for description of field size.
     * name: name of field in DB.
     * ValueType: field type.
     * size: size of field in bytes. (for strings: include \0, i. e.
     *       for VARCHAR(x) size is x+1
     * precision (have sense only for NUMERIC types) - precision.
     * scale (have sense only for NUMERIC types) - scale, as signed byte.
     */

```

```

    **/
struct FieldDescription{
    /// name of field in db
    string      name;
    /// field type
    CosQueryCollection::FieldType      type;
    /// size of field in bytes (for strings: include trailing \0, i. e.
    /// for VARCHAR2(x) size is x+1
    unsigned long  size;
    /// precision (have sense only for numeric types)
    unsigned short precision;
    /// scale (have sense only for numeric types)
    short          scale;
};

///
typedef sequence<FieldDescription> RecordDescription;

/*
struct ParameterDescription
{
    FieldDescription      field;
    CORBA::ParameterMode mode;
};
typedef sequence<ParameterDescription> ParametersDescription;
*/

///
struct QueryError
{
    /// error code: 0 is OK.
    long  errorCode;
    /// error message
    string errorMessage;
    /// sql string, during execution of which error causes.
    string sqlString;
    /// db name
    string dbName;
    /// error code from underlying database
    long dbErrorCode;
};

///
exception QueryNotPrepared {};
///
exception InvalidParameterName{};

```

```

///
exception InvalidParameterType{};
///
exception PropertyNotDefined{};

/**
 * Hight level interface for evaluationg SQL queries
 */
interface QueryEvaluator
#ifdef CORBA_HAVE_OTS
    // :CosTransactions::TransactionalObject
    // in all ORB-s context is passing unconditionally
#endif
{

    /**
     * evaluate query <code> queryText </code> and return result as
     * RC-coded octet sequence.
     * @param queryText -- text of query
     * @param queryFlags -- flags for query executing
     * @param recordDescription -- description of input parameters.
     * @param params -- input parameters as RC-coded octet sequence
     * @return result of query
     */
    OctSeq evaluate_rc(in string queryText, in string queryFlags,
                      in RecordDescription recordDescription_,
                      in OctSeq params)
        raises(CosQuery::QueryTypeInvalid,
              CosQuery::QueryInvalid,
              CosQuery::QueryProcessingError);

    /**
     * evaluate query <code> queryText </code> and return result as
     * sequence of records.
     * @param queryText -- text of query
     * @param queryFlags -- flags for query executing
     * @param recordDescription -- description of input parameters.
     * @param params -- input parameters as record sequence.
     * @return result of query
     */
    RC::RecordSeq evaluate_records(in string queryText, in string queryFlags,
                                  in RecordDescription recordDescription_,
                                  in RC::RecordSeq params)
        raises(CosQuery::QueryTypeInvalid,
              CosQuery::QueryInvalid,

```

```

CosQuery::QueryProcessingError);

/**
 * evaluate query <code> queryText </code> and return result as
 * RC-coded octet sequence.
 * @param queryText -- text of query
 * @param queryFlags -- flags for query executing
 * @param recordDescription_ -- description of input parameters.
 * @param params -- input parameters as record .
 * @return result of query
 */
RC::RecordSeq evaluate_record(in string queryText,
                              in string queryFlags,
                              in RecordDescription recordDescription_,
                              in CosQueryCollection::Record params)
    raises(CosQuery::QueryTypeInvalid,
           CosQuery::QueryInvalid,
           CosQuery::QueryProcessingError);

/**
 * evaluate query <code> queryText </code> without bind parameters
 * and return result as RC-coded octet sequence.
 * @param queryText -- text of query
 * @param queryFlags -- flags for query executing
 * @return result of query
 */
OctSeq evaluate_rc_e(in string queryText, in string queryFlags)
    raises(CosQuery::QueryTypeInvalid,
           CosQuery::QueryInvalid,
           CosQuery::QueryProcessingError);

/**
 * evaluate query <code> queryText </code> without bind parameters
 * and return result as sequence of records.
 * @param queryText -- text of query
 * @param queryFlags -- flags for query executing
 * @return result of query
 */
RC::RecordSeq evaluate_records_e(in string queryText, in string queryFlags)
    raises(CosQuery::QueryTypeInvalid,
           CosQuery::QueryInvalid,
           CosQuery::QueryProcessingError);

```

```

/**
 * evaluate query <code> queryText </code> and fill out and inout
 * parameters of query, return result as RC-coded octet sequence.
 * @param queryText -- text of query
 * @param queryFlags -- flags for query executing
 * @param recordDescription_ -- description of input parameters.
 * @param params -- input parameters as record .
 * @return result of query
 **/
OctSeq evaluate_rc_inout(in string queryText, in string queryFlags,
                        in RecordDescription recordDescription_,
                        inout OctSeq params)
    raises(CosQuery::QueryTypeInvalid,
           CosQuery::QueryInvalid,
           CosQuery::QueryProcessingError);

/**
 * evaluate query <code> queryText </code> and fill out and inout
 * parameters of query, return result as sequence of records.
 * @param queryText -- text of query
 * @param queryFlags -- flags for query executing
 * @param recordDescription_ -- description of input parameters.
 * @param params -- input parameters as record .
 * @return result of query
 **/
RC::RecordSeq evaluate_records_inout(in string queryFlags,
                                     in string queryType,
                                     in RecordDescription recordDescription_,
                                     inout RC::RecordSeq params)
    raises(CosQuery::QueryTypeInvalid,
           CosQuery::QueryInvalid,
           CosQuery::QueryProcessingError);

};

interface Query;
interface QueryManager;

/**
 * this is interface for UAKG Query
 * Query is SQL text with set of parameters: prepare parameters and
 * execute parameters.
 * prepare parameters are descriptions of appropriate execute parameters

```

```

* execute parameters are SQL host variables.
* i. e. let we have query (SELECT * from T where x=:x and y=:y);
* than prepare query have type RecordDescription and consist from
* FieldDescription of :x and :y.
* execute query are values of :x and :y (or sequence of pair of values
* for multiple evaluated query).
*/
interface Query
{

    /**
     *@return owner of query
     **/
    readonly attribute QueryManager query_mgr;

    /**
     *@return text of query.
     */
    readonly attribute string  queryText;

    /**
     * return status of query: i.e:
     * complete when query is executed, otherwise incomplete
     */
    CosQuery::QueryStatus get_status ();

    /**
     * prepare query for executing.
     * if query have no parameters, paramsDescription must be empty
     * sequence.
     */
    void prepare_query(in RecordDescription paramsDescription)
        raises(CosQuery::QueryProcessingError);

    /**
     * synonym for prepare_query
     **/
    void prepare(in RecordDescription paramsDescription)
        raises(CosQuery::QueryProcessingError);

    /**
     * execute query
     *@params octSeq_ records of execute parameters, coded as RCSeq

```

```

* (note, that prepare parameters is record descriptio of execute
* record).
**/
void execute_rc(in OctSeq octSeq_)
    raises(CosQuery::QueryProcessingError);

/**
* execute query with inout parameters
*@params octSeq_ records of execute parameters, coded as RCSeq
**/
void execute_rc_inout(inout OctSeq octSeq_)
    raises(CosQuery::QueryProcessingError);

/**
* execute query
*@params records -- query host parameters in RecordSeq
* (query will be evaluated records.length() times)
**/
void execute_records(in RC::RecordSeq records)
    raises(CosQuery::QueryProcessingError);

/**
* execute query
*@params record_ -- query host parameters in one record
**/
void execute_record(in CosQueryCollection::Record record_)
    raises(CosQuery::QueryProcessingError);

///
void execute_records_inout(inout RC::RecordSeq recordSeq_)
    raises(CosQuery::QueryProcessingError);

///
RecordDescription get_result_description()
    raises(CosQuery::QueryProcessingError,
           QueryNotPrepared);

/**
* get description of records parameters
*@precondition
* must be called after prepare
**/
RecordDescription get_parameters_description()
    raises(CosQuery::QueryProcessingError);

///

```

```

RC::RecordSeq get_all_parameters_records()
    raises(CosQuery::QueryProcessingError);

///  

RC::RecordSeq get_parameters_records(in StringSeq neededFields)
    raises(CosQuery::QueryProcessingError,  
          InvalidParameterName);

///  

OctSeq get_all_parameters_rc()
    raises(CosQuery::QueryProcessingError);

///  

OctSeq get_parameters_rc(in StringSeq fieldNames)
    raises(CosQuery::QueryProcessingError,  
          InvalidParameterName);

/**
 * @returns number of fetched rows.
 */
unsigned long get_row_count()
    raises(CosQuery::QueryProcessingError);

/**
 * fetch query result in records.
 * @param numberOfRecords -- number of records to fetch.
 *     0 means, that we want to fetch all records.
 * @param more -- true, if status is incomplete (i.e. we can query
 * more results), otherwise false.
 * @returns fetched rows packed in RC coding to octet sequence.
 */
OctSeq fetch_rc(in unsigned long numberOfRecords, out boolean more)
    raises(CosQuery::QueryProcessingError);

/**
 * synonym for fetch_rc.
 */
OctSeq get_result_rc(in unsigned long numberOfRecords)
    raises(CosQuery::QueryProcessingError);

/**
 * fetch query result in records.
 * @param numberOfRecords -- number of records to fetch.
 *     0 means, that we want to fetch all records.
 * @param more -- true, if status is incomplete (i.e. we can query

```

```

    * more results), otherwise false.
    * @returns fetched records.
    **/
RC::RecordSeq fetch_records(in unsigned long numberOfRecords,
                           out boolean more)
    raises(CosQuery::QueryProcessingError);

/**
 * synonym for fetch_records
 */
RC::RecordSeq get_result_records(in unsigned long numberOfRecords)
    raises(CosQuery::QueryProcessingError);

/**
 * skip N records without retrieving.
 * @returns actual number of skipped records.
 */
unsigned long skip(in unsigned long numberOfRecords,
                  out boolean more)
    raises(CosQuery::QueryProcessingError);

/**
 * request Blob for filling query parameters
 * @returns empty Blob for writing only.
 */
CosQueryCollection::Blob create_blob()
    raises(CosQuery::QueryProcessingError);

/**
 * request Clob for filling query parameters
 * @returns empty Clob for writing only.
 */
CosQueryCollection::Clob create_clob()
    raises(CosQuery::QueryProcessingError);

/**
 * request Wclob for filling query parameters
 * @returns empty Wclob for writing only.
 */
CosQueryCollection::Wclob create_wclob()
    raises(CosQuery::QueryProcessingError);

/**
 * @return last error.

```

```

        * if Query is ok, code in error is 0.
        */
        QueryError  get_last_error();

        /**
         * destroy query, which not longer needed
         **/
        void        destroy();

};

//
// UAKGQueryCollections
//

interface UAKGCollectionListener;
interface UAKGIterator;

///
exception ReadOnlyCollection {};
///
exception ReadOnlyIterator {};
///
exception KeyNotFound {};

///
interface UAKGCollection: QueryEvaluator
{

    ///
    readonly attribute string  selectQueryText;
    ///
    readonly attribute string  selectDistinctQueryText;
    ///
    readonly attribute string  selectRangeQueryText;
    ///
    readonly attribute string  countQueryText;
    ///
    readonly attribute string  insertQueryText;
    ///
    readonly attribute string  removeAllQueryText;
    ///
    readonly attribute string  orderByText;

    ///

```

```

RecordDescription getRecordDescription()
                    raises(CosQuery::QueryProcessingError);

///
void      set_readonly(in boolean ronly)
                    raises(ReadOnlyCollection);

///
boolean  is_readonly();

/**
 * true, is select collection is ordered.
 **/
readonly attribute boolean  sorted;

/**
 * add record
 **/
void      add_record(in CosQueryCollection::Record element)
                    raises(CosQueryCollection::ElementInvalid,
                          CosQuery::QueryProcessingError,
                          ReadOnlyCollection);

/**
 * add records
 **/
void      add_records(in RC::RecordSeq elements)
                    raises(CosQueryCollection::ElementInvalid,
                          CosQuery::QueryProcessingError,
                          ReadOnlyCollection);

/**
 * add records coded in RC sequence
 **/
void      add_rc(in OctSeq rc)
                    raises(CosQueryCollection::ElementInvalid,
                          CosQuery::QueryProcessingError,
                          ReadOnlyCollection);

//
// retrieve record number

/**

```

```

*return number of records in collection
*@returns number of records in collection
**/
unsigned long get_number_of_records()
    raises(CosQuery::QueryProcessingError);

//
// retrieve records

/**
 * retrieve records by filter.
 * @param where-filter : logical expression for selection of records
 * to delete (in SQL-like DBs is context of where clause)
 * TODO: what it return is it correct ?
 */
OctSeq retrieve_by_filter(in string where_filter)
    raises(CosQuery::QueryProcessingError);

/**
 * retrieve records by pattern.
 * @param : pattern
 * TODO: what it return is it correct ?
 */
OctSeq retrieve_by_pattern(in CosQueryCollection::Record pattern)
    raises(CosQuery::QueryProcessingError,
           CosQueryCollection::ElementInvalid);

//
// replacing
//

/**
 * update records by pattern
 * @param newRecord -- new record instead pattern matched
 * @param pattern -- pattern for matching
 */
void update_by_pattern(in CosQueryCollection::Record newRecord,
                      in CosQueryCollection::Record pattern )
    raises(CosQuery::QueryProcessingError,
           CosQueryCollection::ElementInvalid,
           ReadOnlyCollection);

/**
 * update records by filter

```

```

*@param newRecord -- new record instead filter matched
*@param filter -- condition
**/
void      update_by_filter( in CosQueryCollection::Record newRecord,
                           in string filter )
        raises(CosQuery::QueryProcessingError,
               CosQueryCollection::ElementInvalid,
               ReadOnlyCollection);

//
// removing
//

/**
 * remove all records from collection
 **/
void      remove_all_records()
        raises(CosQuery::QueryProcessingError,
               ReadOnlyCollection);

/**
 * remove records with same value as <code> record_ </code>
 *@param record_ - value of record to be removed.
 **/
void      remove_record(in CosQueryCollection::Record record_)
        raises(CosQuery::QueryProcessingError,
               CosQueryCollection::ElementInvalid,
               ReadOnlyCollection);

/**
 * remove records with are satisficated to <code> filter </code>
 *@param filter - logical expression for selectiong removed records.
 **/
void      remove_records_by_filter(in string filter)
        raises(CosQuery::QueryProcessingError,
               ReadOnlyCollection);

/**
 * remove records with are match pattern <code> pattern </code>
 *@param pattern - pattern to match.
 **/
void      remove_records_by_pattern(in CosQueryCollection::Record pattern)
        raises(CosQuery::QueryProcessingError,
               ReadOnlyCollection);

//

```

```

// elements ordering
//

/**
 * sort - set new order expression
 * @param order_expression - new expression for ORDER BY clause
 */
void      sort(in string order_expression)
           raises(CosQuery::QueryProcessingError);

//
// access interfaces factories
//

/**
 * create iterator
 */
UAKGIterator  create_iterator();

/**
 * create iterator which iterate records, matched for pattern
 */
UAKGIterator  create_iterator_by_pattern(
               in CosQueryCollection::Record pattern)
           raises(CosQueryCollection::ElementInvalid,
                 CosQuery::QueryProcessingError);

/**
 *
 * subquery must be specified in next form:
 * <code>
 *   select <field_list> from <table_list>
 *   where <conditions> [order by <field_list>]
 * </code>
 */
UAKGCollection  create_subcollection(in string subquery)
               raises(CosQuery::QueryInvalid,
                     CosQuery::QueryProcessingError);

///
UAKGCollection  create_subcollection_by_pattern(
               in CosQueryCollection::Record pattern)
           raises(CosQuery::QueryInvalid,
                 CosQuery::QueryProcessingError,
                 CosQueryCollection::ElementInvalid);

```

```

/**
 * add listener to collection events
 **/
unsigned long  add_listener(in UAKGCollectionListener listener,
                          in unsigned short eventMask);

/**
 * remove listener
 **/
boolean  remove_listener(in unsigned long listenerIndex);

/**
 * destroy collection and free server resources, associated with
 * this collection.
 **/
void      destroy();
};

///
interface UAKGCollectionListener
{
    ///
    void  elements_added(in OctSeq elements);
    ///
    void  elements_updated(in OctSeq prev_elements,
                          in OctSeq new_elements);
    ///
    void  elements_removed(in OctSeq elements);
    ///
    void  all_elements_removed();
    ///
    void  collection_destroyed();
};

struct ListenersSeqStruct
{
    UAKGCollectionListener listener;
    unsigned short          mask;
};

typedef sequence<ListenersSeqStruct> UAKGCollectionListeners;

/**
 * Iterator for retrieving data

```

```

**/
interface UAKGIterator
{
    /**
     * are we situated at the end of data set ?
     */
    readonly attribute boolean end;

    /**
     * fetch n records as RC-coded octet sequence
     * @param n - number of records to fetch
     * @param more - set to true, if we not at end of collection.
     * @returns fetched records.
     */
    OctSeq      fetch_rc(in unsigned long n, out boolean more);

    /**
     * fetch n records as records sequence
     * @param n - number of records to fetch
     * @param more - set to true, if we not at the end of collection.
     * @returns fetched records.
     */
    RC::RecordSeq  fetch_records(in unsigned long n, out boolean more);

    /**
     * skip n records
     * @param n - number of records to skip
     * @param more - set to true, if we not at the end of collection.
     * @returns actual number of skipped records.
     */
    unsigned long  skip(in unsigned long n, out boolean more);

    /**
     * destroy iterator and free associated server resources.
     */
    void destroy();
};

/**
 * Collection of records with unique keys.
 */
interface UAKGKeyCollection: UAKGCollection
{
    ///
    RecordDescription  get_key_description();
}

```

```

///
CosQueryCollection::Record
    retrieve_record_with_key(in CosQueryCollection::Record key)
        raises(CosQuery::QueryProcessingError);

///
void update_record_with_key(in CosQueryCollection::Record newRecord,
    in CosQueryCollection::Record key)
    raises(CosQuery::QueryProcessingError, KeyNotFoundError);

///
void remove_record_with_key(in CosQueryCollection::Record key)
    raises(CosQuery::QueryProcessingError);

///
OctSeq retrieve_records_with_keys(in OctSeq keys)
    raises(CosQuery::QueryProcessingError);

///
void update_records_with_keys(in OctSeq records)
    raises(CosQuery::QueryProcessingError);

///
void remove_records_with_keys(in OctSeq keys)
    raises(CosQuery::QueryProcessingError);

};

/**
 * factory for collection interfaces
 */
interface UAKGCollectionFactory
{

/**
 * queryText - select <field_list> from <table_list>
 * where <conditions> [order by <field_list>]
 */
UAKGCollection create_collection( in string queryText )
    raises(CosQuery::QueryInvalid,
        CosQuery::QueryProcessingError);

```

```

/**
 * queryText - select <field_list> from <table_list>
 * where <conditions> [order by <field_list>] with key [key-fields-list]
 */
UAKGKeyCollection create_key_collection(
    in string queryText
    )
    raises(CosQuery::QueryInvalid,
        CosQuery::QueryProcessingError);

///
UAKGCollection create_collection_by_parts(
    in string selectPartText,
    in string fromPartText,
    in string wherePartText,
    in string orderByPartText)
    raises(CosQuery::QueryInvalid,
        CosQuery::QueryProcessingError);

///
UAKGKeyCollection create_key_collection_by_parts(
    in string selectPartText,
    in string fromPartText,
    in string wherePartText,
    in string orderByPartText,
    in string keysPartText)
    raises(CosQuery::QueryInvalid,
        CosQuery::QueryProcessingError);
};

/**
 * interface for our QueryManager.
 */
interface QueryManager: QueryEvaluator,
    UAKGCollectionFactory
{
    ///
    string get_username() raises(CosQuery::QueryProcessingError);
    ///
    string get_dblink() raises(CosQuery::QueryProcessingError);

    ///
    readonly attribute unsigned long number_of_queries;

    ///

```

```

Query create_query(in string query, in string flags)
    raises(CosQuery::QueryTypeInvalid,
           CosQuery::QueryInvalid);

///
Query create(in string query, in string flags)
    raises(CosQuery::QueryTypeInvalid,
           CosQuery::QueryInvalid);

///
string get_property(in string property_name)
    raises(PropertyNotDefined);

///
long get_next_sequence_value(in string sequence_name)
    raises(CosQuery::QueryProcessingError,
           CosQuery::QueryInvalid,
           CosQuery::QueryTypeInvalid);

///
void destroy();
};

///
exception QueryManagerNotFound {};

typedef sequence<QueryManager> UAKGQueryManagerSeq;

///
interface DBConnectionManager
{
    ///
    QueryManager createQueryManager(in string login, in string password,
                                    in string db_name, in string drv_name,
                                    in string implementation_specific_data)
        raises(QueryManagerNotFound,
               CosQuery::QueryProcessingError);

    /**
     * shutdown query service.
     */
    void shutdown();
};

```

```
};
```

```
#endif
```

## 12.4 RC.idl

```
#ifndef __RC_IDL
```

```
#define __RC_IDL
```

```
/*
```

```
 * definitions and pseudo-interfaces for custom Record Marshalling.
 * (C) Ruslan Shevchenko <Ruslan@Shevchenko.Kiev.UA>, 1999
 * (C) GradSoft, 2001
 * $Id: ProgrammingGuide_eng.tex,v 1.25 2002/01/29 13:24:48 srkr Exp $
 */
```

```
#ifndef __COSQUERYCOLLECTION_IDL
```

```
#include <CosQueryCollection.idl>
```

```
#endif
```

```
#pragma prefix "gradsoft.kiev.ua"
```

```
/**
```

```
 * pseudo-interfaces for custom Record Marshalling
 * The main entity is: RC-coded octet sequence.
 * We provide 2 pseudo-interfaces: RCReader and RCWriter
 * for reading/writing from/to RCSeq.
 */
```

```
module RC
```

```
{
```

```
///
```

```
typedef sequence<octet> OctetSeq;
```

```
//typedef CosQueryCollection::Record Record;
```

```
///
```

```
typedef sequence<CosQueryCollection::Record> RecordSeq;
```

```
//typedef CosQueryCollection::Decimal Decimal;
```

```
/**
```

```
 * thrown, when Reader discovered error in OctSeq.
```

```
*/
```

```
exception BadOctSeq
```

```
{
```

```
/**
```

```
 * position of read failure (in bytes).
```

```

    **/
long    pos;
/**
 * what was happened ?
 **/
string reason;
};

///
exception TypeNotImplemented
{
    ///
    CosQueryCollection::FieldType fieldType;
};

///
exception FieldValueIsNull {};
///
exception InvalidPosition {};

/**
 * header of RC-coded octet sequence.
 **/
struct RCHeader
{
    ///
    octet version;
    /// number of records in sequence.
    /// (if -1, than number of records is unknown).
    long nRecords;
    /// number of fields in one record.
    unsigned long nFields;
};

///
exception InvalidHeadData {};

/**
 * this pseudointerface must be mapped to RCWriter static class
 * in host language.
 */
interface Writer // pseudo
{
    /**
     *write header of Octet Sequence to octSeq_.

```

```

    *@param nRecords - number of records to be coded.
    *@param nFields - number of fields in one record.
    *@param pos - position (input really ignored, on output it
    * is settet to first position after header).
    *@param octSeq_ - sequence, in which we code.
    **/
void writeHeader(in long nRecords, in unsigned long nFields,
                inout unsigned long pos, inout OctetSeq octSeq_)
                raises (InvalidHeadData);

///
void writeHead(inout unsigned long pos, inout OctetSeq octSeq_)
                raises (InvalidHeadData);

///
void writeRecord(in CosQueryCollection::Record record,
                inout unsigned long pos,
                inout OctetSeq octSeq_ )
                raises(TypeNotImplemented);

///
void writeRecordSeq(in RecordSeq recordSeq_)
                raises(TypeNotImplemented);

///
void writeBoolean(in boolean value, inout unsigned long pos,
                 inout OctetSeq octSeq_);

///
void writeChar(in char value, inout unsigned long pos,
               inout OctetSeq octSeq_);

///
void writeShort(in short value, inout unsigned long pos,
                inout OctetSeq octSeq_);

///
void writeLong(in long value, inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeULong(in unsigned long value, inout unsigned long pos,
                inout OctetSeq octSeq_);

///
void writeFloat(in float value, inout unsigned long pos,

```

```

                                inout OctetSeq octSeq_);

///
void writeDouble(in float value, inout unsigned long pos,
                inout OctetSeq octSeq_);

///
void writeString(in string value, inout unsigned long pos,
                inout OctetSeq octSeq_);

///
void writeWString(in wstring value, inout unsigned long pos,
                 inout OctetSeq octSeq_);

///
void writeObject(in Object value, inout unsigned long pos,
                inout OctetSeq octSeq_);

///
void writeDecimal(in CosQueryCollection::Decimal value,
                 inout unsigned long pos,
                 inout OctetSeq octSeq_);

///
void writeRaw(in OctetSeq value, inout unsigned long pos,
              inout OctetSeq octSeq_);

///
void writeDateTime(in CosQueryCollection::DateTime value,
                  inout unsigned long pos,
                  inout OctetSeq octSeq_);

///
void writeFieldValue(in CosQueryCollection::FieldValue value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeNullField(inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeBooleanField(in boolean value,
                      inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeCharField(in char value,
                   inout unsigned long pos, inout OctetSeq octSeq_);

```

```

///
void writeOctetField(in char value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeShortField(in short value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeUShortField(in unsigned short value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeLongField(in long value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeULongField(in unsigned long value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeFloatField(in float value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeDoubleField(in double value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeStringField(in string value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeObjectField(in Object value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeDecimalField(in CosQueryCollection::Decimal value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///
void writeRawField(in OctetSeq value,
                    inout unsigned long pos, inout OctetSeq octSeq_);

///

```

```

void    writeLongRawField(in OctetSeq value,
                          inout unsigned long pos, inout OctetSeq octSeq_);
///
void    writeBlob(in CosQueryCollection::Blob value,
                  inout unsigned long pos, inout OctetSeq octSeq_);
///
void    writeClob(in CosQueryCollection::Clob value,
                  inout unsigned long pos, inout OctetSeq octSeq_);
///
void    writeWclob(in CosQueryCollection::Wclob value,
                  inout unsigned long pos, inout OctetSeq octSeq_);

///
OctetSeq copyStream(in unsigned long from_pos, in unsigned long to_pos,
                    in OctetSeq octSeq_)
                raises(InvalidPosition);

};

/**
 * this pseudointerface must be mapped to RCReader static class
 * in host language.
 */
interface Reader
{
    ///
    void    readHeader(inout RCHeader header, inout unsigned long pos,
                      in OctetSeq octSeq_)
                raises(BadOctSeq);

    ///
    CosQueryCollection::Record readRecord(inout unsigned long pos, in OctetSeq octSeq_ )
                raises(BadOctSeq);

    ///
    RecordSeq readRecordSeq(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq);

    ///
    CosQueryCollection::FieldValue readField(inout unsigned long pos,
                                              in OctetSeq octSeq_)
                raises(BadOctSeq);

    /**
     * return true and skip null value, if return was null, otherwise

```

```

    * return false and not touch pos.
    */
boolean    nextFieldIsNull(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq);

///
CosQueryCollection::FieldType nextFieldType(inout unsigned long pos,
                in OctetSeq octSeq_)
                raises(BadOctSeq);

///
boolean    readBooleanField(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
void       readBooleanField_inout(inout boolean value,
                inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
char       readCharField(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
void       readCharField_inout(inout char value,
                inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
octet      readOctetField(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
short      readShortField(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
unsigned short readUShortField(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
long       readLongField(inout unsigned long pos, in OctetSeq octSeq_)
                raises(BadOctSeq,FieldValueIsNull);

///
unsigned long readULongField(inout unsigned long pos, in OctetSeq octSeq_)

```

```

                                                    raises(BadOctSeq,FieldValueIsNull);

///
float    readFloatField(inout unsigned long pos, in OctetSeq octSeq_)
                                                    raises(BadOctSeq,FieldValueIsNull);

///
double   readDoubleField(inout unsigned long pos, in OctetSeq octSeq_)
                                                    raises(BadOctSeq,FieldValueIsNull);

///
string   readStringField(inout unsigned long pos, in OctetSeq octSeq_)
                                                    raises(BadOctSeq,FieldValueIsNull);

///
Object   readObjectField(inout unsigned long pos, in OctetSeq octSeq_)
                                                    raises(BadOctSeq,FieldValueIsNull);

///
CosQueryCollection::Decimal readDecimalField(
    inout unsigned long pos,
    in OctetSeq octSeq_
)raises(BadOctSeq,FieldValueIsNull);

///
CosQueryCollection::Decimal readNumericField(
    inout unsigned long pos,
    in OctetSeq octSeq_
)raises(BadOctSeq,FieldValueIsNull);

///
CosQueryCollection::DateTime readDateTimeField(
    inout unsigned long pos,
    in OctetSeq octSeq_
)raises(BadOctSeq,FieldValueIsNull);

///
OctetSeq readRawField(inout unsigned long pos, in OctetSeq octSeq_)
                                                    raises(BadOctSeq,FieldValueIsNull);

///
void     readRawField_inout(inout OctetSeq value,
    inout unsigned long pos, in OctetSeq octSeq_)
                                                    raises(BadOctSeq,FieldValueIsNull);

///

```

```

OctetSeq readLongRawField(inout unsigned long pos, in OctetSeq octSeq_)
    raises(BadOctSeq,FieldValueIsNull);

///
void readLongRawField_inout(inout OctetSeq value,
    inout unsigned long pos, in OctetSeq octSeq_)
    raises(BadOctSeq,FieldValueIsNull);

///
string readLongStringField(inout unsigned long pos, in OctetSeq octSeq_)
    raises(BadOctSeq,FieldValueIsNull);

///
void readLongStringField_inout(inout string value,
    inout unsigned long pos, in OctetSeq octSeq_)
    raises(BadOctSeq,FieldValueIsNull);

///
void readBlob(inout CosQueryCollection::Blob value,
    inout unsigned long pos, in OctetSeq octSeq_)
    raises(BadOctSeq,FieldValueIsNull);

///
void readClob(inout CosQueryCollection::Clob value,
    inout unsigned long pos, in OctetSeq octSeq_)
    raises(BadOctSeq,FieldValueIsNull);

///
void readWclob(inout CosQueryCollection::Wclob value,
    inout unsigned long pos, in OctetSeq octSeq_)
    raises(BadOctSeq,FieldValueIsNull);

};

};

#endif

```

## 13 RC-coding specifications

```
RCStream:: Version, RecordArray
```

```
Version:: 0x01
```

```
RecordArray :: NumberOfRecords[4], RecordHeader , RecordData<1..infinity> ;
```

```

Record :: RecordHeader,RecordData;

RecordHeader :: NumberOfFields[1];

RecordData :: FieldBlock<NumberOfFields> ;

FieldBlock :: DataType , DataValue;

DataType ::
    TypeNull      0x00
|   TypeBoolean  0x01
|   TypeChar     0x02
|   TypeOctet    0x03
|   TypeShort    0x04
|   TypeUShort   0x05
|   TypeLong     0x06
|   TypeULong    0x07
|   TypeFloat    0x08
|   TypeDouble   0x09
|   TypeString   0x0A
|   TypeObject   0x0B
|   TypeAny      0x0C
|   TypeSmallInt 0x0D
|   TypeInteger  0x0E
|   TypeDecimal  0x0F
|   TypeNumeric  0x10
|   TypeRaw      0x11
|   TypeLongRaw  0x12
|       TypeLongString 0x13
|       TypeWString  0x14
|       TypeDateTime 0x15
;

DataValue ::
    ValueNull[0],
|   ValueBoolean[1]
|   ValueChar[1]
|   ValueWchar[2]
|   ValueShort[2] // network order
|   ValueUShort[2] // network order
|   ValueLong[4] // network order
|   ValueULong[4] // network order
|   ValueFloat[4] // network order
|   ValueDouble[8] // network order
|   ValueString
|   ValueOctets

```

```

|      ValueWString
|      ValueOctet[1]
|          ValueDecimal
|          ValueAny
|          ValueObject
|          ValueDateTime
;

ValueDecimal:: ValueLong, ValueLong, ValueRaw
              // precision, scale, value

ValueDateTime:: ValueShort, ValueOctet, ValueOctet, ValueOctet, ValueOctet, ValueOctet
               //      year      , month   , day     , hour   , minute  , second

ValueString  :: Length[4] //network order ,ValueChar<Length> ;
ValueWString :: Length[4] //network order ,ValueWChar<Length> ;
ValueOctets  :: Length[4] //network order ,ValueOctet<Length> ;
ValueAny     :: Length[4], TypeCode id as String, value as OctetSeq
ValueObject  :: Length[4], GIOP ObjectReference

```

## 14 Grammar of collection SQL queries

```

%%

startTerm: query
          ;

query:
  SELECT select_opt selection table_expr opt_order_by_clause opt_key_clause
  |
  was_error
  ;

select_opt:
  /* empty */
|
  ALL
|
  DISTINCT
  ;

```

```

selection:
select_scalar_expr_comma_list
|
'*,
        ;

table_expr:
        from_clause
        opt_where_clause
        opt_group_by_clause
        opt_having_clause
        ;

from_clause:
        FROM table_ref_comma_list
        |
        FROM '(' query ')',
        ;

opt_where_clause:
        /* empty */
        |
        WHERE search_condition
        ;

opt_group_by_clause:
        /* empty */
        |
        GROUP BY column_ref_comma_list
        ;

opt_having_clause:
        /* empty */
        |
        HAVING search_condition
        ;

opt_key_clause:
        /* empty */
        |
        WITH KEY selection
        ;

```

```

opt_order_by_clause:
    /* empty */
    |
    ORDER BY ordering_spec_commlist
    ;

search_condition:
    search_condition OR search_condition
    |
    search_condition AND search_condition
    |
    NOT search_condition
    |
    '(' search_condition ')'
    |
    predicate
    |
    '@'
    ;

predicate:
    comparison_predicate
    |
    between_predicate
    |
    like_predicate
    |
    test_for_null
    |
    in_predicate
    |
    all_or_any_predicate
    |
    existence_predicate
    ;

comparison_predicate:
    scalar_expr LESS subquery
    |
    scalar_expr LESS scalar_expr
    |
    scalar_expr LESS_EQ subquery
    |
    scalar_expr LESS_EQ scalar_expr
    ;

```

```

    scalar_expr GT scalar_expr
    |
    scalar_expr GT subquery
    |
    scalar_expr GT_EQ scalar_expr
    |
    scalar_expr GT_EQ subquery
    |
    scalar_expr EQ subquery
    |
    scalar_expr EQ scalar_expr
    |
    scalar_expr NEQ subquery
    |
    scalar_expr NEQ scalar_expr
    ;

```

```

test_for_null:
scalar_expr IS NOT DBNULL
|
    scalar_expr IS DBNULL
    ;

```

```

in_predicate:
    scalar_expr NOT IN subquery
    |
    scalar_expr IN subquery
    |
    scalar_expr NOT IN '(' atom_commlist ')'
    |
    scalar_expr IN '(' atom_commlist ')'
    ;

```

```

all_or_any_predicate:
    scalar_expr LESS any_all_some subquery
    |
    scalar_expr LESS_EQ any_all_some subquery
    |
    scalar_expr GT any_all_some subquery
    |
    scalar_expr GT_EQ any_all_some subquery
    |
    scalar_expr EQ any_all_some subquery
    |
    scalar_expr NEQ any_all_some subquery

```

```

        ;

any_all_some:
    ANY
    |
    ALL
    |
    SOME
    ;

existence_predicate:
    EXISTS subquery
    ;

between_predicate:
    scalar_expr NOT BETWEEN scalar_expr AND scalar_expr
    |
    scalar_expr BETWEEN scalar_expr AND scalar_expr
    ;

like_predicate:
    scalar_expr NOT LIKE atom opt_escape
    |
    scalar_expr LIKE atom opt_escape
    ;

opt_escape:
    /* empty */
    { $$=NULL; }
    |
    ESCAPE atom
    ;

ordering_spec_commlist:
    ordering_spec
    |
    ordering_spec_commlist ',' ordering_spec
    ;

ordering_spec:
    INTNUM opt_ordering
    |
    column_ref opt_ordering
    ;

opt_ordering:

```

```

        /* empty */
        |
        ASC
        |
        DESC
        ;

subquery:
    '(' SELECT select_opt selection table_expr ')'
    ;

select_scalar_expr_comma_list:
    select_scalar_expr
    |
    select_scalar_expr_comma_list ',' select_scalar_expr
    ;

select_scalar_expr:
    '@'
    |
    scalar_expr
    ;

scalar_expr:
scalar_expr '+' scalar_expr
    |
    scalar_expr '-' scalar_expr
    |
    scalar_expr '*' scalar_expr
    |
    scalar_expr '/' scalar_expr
    |
    scalar_expr CONCAT_OP scalar_expr
    |
    '-' scalar_expr %prec UMINUS
    |
    '+' scalar_expr %prec UMINUS
    |
    atom
    |
    column_ref
    |
    function_ref
    |

```

```

        '(' scalar_expr ')'
    ;

table_ref_commalist:
    table_ref
    |
        table_ref_commalist ',' table_ref
    ;

atom_commalist:
    atom
    |
        atom_commalist ',' atom
    ;

atom:
parameter_ref
    |
    literal
    |
    { $$=yy_createConstStringTerm("USER"); }
    ;

column_ref_commalist:
    column_ref
    |
        column_ref_commalist ',' column_ref
    ;

column_ref:
    IDENT
    |
    IDENT '.' IDENT
    |
    IDENT '.' IDENT '.' IDENT
    ;

function_ref:
    compound_name '(' '*' ')'
    |
    compound_name '(' DISTINCT column_ref ')'
    |
    compound_name '(' ALL scalar_expr ')'
    |
    compound_name '(' scalar_expr ')'

```

```

        ;

compound_name:
    IDENT
    |
    IDENT '.' IDENT
    ;

table_ref:
    compound_name
    |
    compound_name IDENT
    |
    '@'
    ;

parameter_ref:
parameter
    |
    parameter parameter
    |
    parameter INDICATOR parameter
    ;

parameter:
    ':' IDENT
    ;

literal:
    STRING
    |
    INTNUM
    |
    APPROXNUM
    ;

was_error:
    error
    ;

```

## 14.1 List Of Changes

- 24.01.2002 routine of operating with LOB described
- 21.01.2002 examples code review, InterBase to CORBA type relations

described

- 19.06.2001 - 1.0.2 initialization is described.
- 06.06.2001 - review.
- 08.05.2001 review, adding of applications.
- 16.04.2001 first preview.

## References

- [1] Object Management Group, editor. *The Common Object Request Broker: Architecture & Specification*. OMG, 1999. formal/99-10-07.
- [2] Object Management Group, editor. *Common Object Services Specification*, chapter 14. Query. OMG, 2000. formal/2000-06-23.
- [3] Object Management Group, editor. *Common Object Services Specification*, chapter Transaction Service. OMG, 2000. formal/2000-06-28.
- [4] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999. ISBN 0201379279.
- [5] Anatoliy Doroshenko Ruslan Shevchenko. Techniques to uncreasing performance of corba parallel distributed applications. *Processing of PACT-2001*, 2001. in process.
- [6] Ruslan Shevchenko. Analysis of methods of creating effecient distributed applications, based on corba standarts. *Processing of UKRPROG-2000*, 2000.
- [7] X/Open. *X/Open CAE specification - Distributed Transactions Processing*. ISBN 1-872630-24-3.