UAKGQuery: Programming Guide

April 17, 2001

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

Contents

1 Introduction

UAKGQuery is a CORBA ?? service intendent for database access. It's provide CORBA API for effective querying of relational databases and define hight-level object interfaces for data access.

With UAKGQuery is possible to organize effective and uniform access to relational databases, based on hight-level abstract model and independend 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 was grown from implementation of CORBA Query Service **??**. After analizing of scalability and performance of origonal CosQuery API, structure of UAKGQuery Service was fully refined. Some of methods applied for building of effective distributed applications can be found in [**?**] [**?**]

1.2 Knowledge needed for reading this manual

We assume, that reader have basic knowledge of CORBA architecture **??** and have some experience with CORBA C++ application. (we recommend [**?**] as learning matherial). Also assumed, that reader know what is SQL and what is relational databases.

Last chapter require knowing of such entities, as transactions, CORBA Transaction Service [**?**] and XA transactions [**?**]. But you can ommit this chapter during first reading.

2 General description of UAKGQuery using

Application programmer must perfrom next steps for using UAKGQuery:

- Application must receive object reference to DBConnectionManager (manager of database connection) via CORBA mechanizm of resolving initial references.
- create QueryManager, by call of `DBConnnectionManager::createQueryManager` , which expose API for creation and evaluation of SQL queries and so-called collections.
- With QueryManager is possible or directly evaluater SQL queries, or create Query or Collection objects for interactive reading/writing of data.
- Before end of work it is nesessorry to disconnect from UAKGQuery service, by call to method `destroy` of QueryManager.

3 Connection to database

For connecting to database is nesessorry 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:

```
// receive initial object
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(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 (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();
```

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 are differs only by type of input and output parameters.

Now we will show simple example of such request, than will begin 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 mean:

- OctSeq – This is type of result, which mean 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.

  [1]o you don't write before any query without parameters next piece of code:

  ```
  RecordDescription emptyDescription;
  emptyDescription.length(0);
  ```

[1]S

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

The next is function which print requested data:

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

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 aprropriative valis 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:

	$CORBA$	SQL
$TypeBoolean$	$boolean$	$-$
$TypeChar$	$char$	$CHAR(1)$
$TypeOctet$	$octet$	$-$
$TypeShort$	$short$	$NUMBER(5,0)$
$TypeUShort$	$unsigned\ \ short$	$NUMBER(5,0)$
$TypeLong$	$long$	$NUMBER(10,0)$
$TypeULong$	$unsignedlong$	$NUMBER(10,0)$
$TypeFloat$	$float$	$NUMBER(x,y)\ \ x<5 \wedge y>0$
$TypeDouble$	$double$	$NUMBER(x,y)\ \ x>5 \wedge y>0$
$TypeString$	$string$	$VARCHAR2(X)$
$TypeObject$	$Object$	$VARCHAR2(X) \vee UDF$
$TypeSmallInt$	$short$	$NUMBER(5,0)$
$TypeInteger$	$long$	$NUMBER(10,0)$
$TypeReal$, $TypeDoublePrecision$ $TypeCharacter$		
$TypeDecimal$	$CosQueryCollection :: Decimal$	$NUMBER(x,y) \wedge x>10$
$TypeNumeric$	$CosQueryCollection :: Decimal$	
$TypeDateTime$	$CosQueryCollection :: DateTime$	$DATE$
$TypeRaw$	$sequence < octet >$	LOB
$TypeLongRaw$	$sequnece < octet >$	$BLOB$
$TypeLongString$	$sequence < octet >$	
$TypeWString$	$wstring$	
$TypeLongWString$	$sequence < octet >$	

5.1.1 C++ Facade class FieldValueAccess

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

- `FieldValueAccess::isNil(const FieldValue& fv)` returns 1, if fv is NUL, 0 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 examle:

  ```
  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::FieldVualueIsNull throwed, if we try to read value of wrong type, then InvalidFieldTypeException is throwed.

- FieldValueAccess::setAsString and getAsString allows 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 denote 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 show 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

Appropreative 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 create record with 3 fields: `(3 Short, "aa" String, "bb" String)` .

Using `FieldValueAccess` code fragment, for creationg of this record will look 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 allow 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 append field of such type to `(*this)` and return one.

5.3 OctSeq, RCHeader

OctSeq defined in IDL as sequence of bytes:

```
typedef sequence<octet> OctSeq;
```

Why use OctSeq – becouse UAKGQuery define own coding of databse records to binary stream, which much more effective, than GIOP. Whith UAKGQuery is possible to acchieve performance of data transfer near theoretical maximum.

Yo can write data to octet stream with help of RCWriter and read with help of RCReader. [2]

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

UAKGqueryService provide helper classes, which incapsulate low-level operations, it is nesessorry to know only that RC-sequence constists from header and data; header is defined in next way:

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

[2] RC means Record Coding.

5.3.1 RCReader

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

Now, let's return to function printRC, which read RC stream:

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

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

```
for(Long nRecord=0; nRecord<header.nRecords; ++nRecord){
  for(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 in next way:

```
for(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 fragmet create RC stream and write one record to it:

```
OctSeq_var octSeq = new OctSeq;
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;
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;
 }

\end{verbatrim}

 Now we know, that:
\begin{itemize}
  \item Return value is RC-coded octet sequence.
  \item first parameter - text of SQL query.
  \item second parameter - flags.
  \item third parameter - type of query input parameters. In this case we
 pass empty RecordDescription, becouse this query have no parameters.
  \item  four - query parameters. In our case: empty
\end{itemize}

 Note, that in typical application exists many queries without input parameters, so we define

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

\begin{verbatim}

 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{verbatrim}

 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;
 }


\subsection{ queries with parameters  }

 Now, let's look at third and fourth parameter of QueryEvaluator::evaluate.
 Sense of thired parameter: RecordDescription - is description of record,
 which passed to query as parameter.
 \footnote{
 We guess, reader is familiar with concepts of binding variables (or,
  in some terminology: host variables). If no, please look at
 \cite{ORACLE-1}
}

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

 Let's illustrate this by next example:
 query is
\begin{verbatim}
select name from empls 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 empls where id=:ID","",
                                paramsDescription, params);
  if (retval->length()==0) {
      throw IncorrectEmployeeId(id);
  }
  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 help of SQL construction `select into`

The family of methods `evaluate_*_inout` is intendent 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 into 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 for using all functionality of realtional 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);

/**
 * synonim 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 have 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);

/**
 * synonim 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);

/**
 * synonim 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();


};
```

For using `Query` interface, application programmer must perform next steps:

1. create Query using method `QueryManager::createQuery`
2. prepare query, using method `Query::prepare` This method have one parameter: description of query binding variables.
3. execute query, using one of family of `Query::execute` methods.
4. receive data if nesessorry, using one of methods `Query::fetch_XX`
5. after end of work with query, delete it using mehtod `Query::destroy`

For example, receiving name of employeer 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 becouse parsing and passing of parameters description are performed only once.

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);
 .....
}

char* EmployeeManager::getNameById(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

Yet one future of query interface – API for sequential receiveing of data sets by parsts. (According to *Iterator* design pattern).

Let's look on next example:

```
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 return us description of table, setted by user:

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

8.1 Introduction

Query Collection expose hight-level object model, intendent for work with data sets in DB, organized as 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.

8.2 General description

8.2.1 "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: when you want to select some data from dataset, you use some SQL sentence, which have form:

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

So this SQL sentence define 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 satisficate some condition. In typical colection interfaces exists methods for this, like `retrieve_by_filter`; in SQL we must evaluate SQL sentence which must look as

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

Method `update_by_filter` (intendent to update data, which satisfy some condition) cause 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 appropriative SQL sentences are builded and evaluated.

So, what is the query collection itself: object, which simple keep and evaluate 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)

8.3 Steps for using collections interfaces

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

- create Collection using appropriative 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 nesessorry to free server resources, associated with UAKGCollection by call of `UAKGCollection::destroy`

8.4 Creation of Query Collections

UAKGQuery support two types of collections: UAKGCollection and UAKGKeyCollection.

UAKGCollection represent data set, UAKGKeyCollectiin - 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 on 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 constsits from 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.

8.5 Data Access

8.5.1 Receiving of data with help of iterators interface

You can use `Iterator` concepts for retrieving of 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 methods.
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.

8.5.2 Retryiving 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 returns RC-coded byte stream, which contains records from collection data set filtered by parameter. First method filter data set before retrieving by logical expression in `where_filter`, second – by principle of pattern matching (see 8.5.3).

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, in which F2 is equal to string `'test'` . (I. e. all records in result will have form `(1,'test')`).

8.5.3 Pattern matching

Concept of pattern matching is often used in UAKGQuery Service for filtering of data sets: for example, method `retrieve_by_pattern` retrieve records which are match pattern, passed as argument of this method. What this mean: 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 in data sets. Pattern matching for collection is defined as follow:

- Structure of pattern record must be equal to structure of records in queuing datatset (i.e. number of fields and types must be equal)
- Record `r` match pattern `p` iff
  1. $foralli \in 0 \ldots length(r) : p[i]! = NIL \rightarrow p[i]\ match\ r[i]$ (i. e. for all field indexes, appropriative fields must match if pattern field is not NIL).
  2. For fields p_i, r_i, $p_i\ match\ r_i$ iff
     (a) $type(r_i) == string \rightarrow (r_i\ \ LIKE\ \ p_i)$
     (b) $type(r_i)! = string \rightarrow (r_i\ \ ==\ \ p_i)$

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

Practical example:

Let we have table UAKGTEST with two fields: F1 of type NUMBER and F2 of type VARCHAR2. Then, let we created collection which work with records of this table. Then, for retryiving all records from this collection with F1=5 programmer must perfrm next steps:

- Form pattern as record from 2 fields, where first field must be 5, second - NULL . (i. e. second fill 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);
```

8.6 Work with UAKGCollection - adding, updating and deletiong of data

Of course, exceppt retryiving of data, you can add, modify or delete collection items.

List of aprropriative methods:

1. Adding of 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 record, coded in RC stream.

   Note, that during adding of record no checking of belonging added data to collection is performed. For example, let's look on 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;
```

On successfull 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 satisficate 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 of 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();
```

8.7 Collection queries

For comfortable access to collection data we need not only methods for work with collection methods, but 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 `UAKGColletion` is intendent. (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 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.

8.7.1 Limitations

- Colection can evaluate only SELECT queries without `HAVING_BY` and `GROUP_BY` clauses.

8.7.2 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)
  ```

8.8 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 for using 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 multiply 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();
```

8.9 UAKGKeyCollection

UAKGKeyCollection is a specialization of UAKGCollection with property "have primary key". [3].

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

8.9.1 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 appropriative parts of SQL sentence and yet ine additional part: list of field names, from which key is consists. Example:

  ```
  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"
                                  );
  ```

[3] i. e. key, which is unique for each record in collection

`WITH KEY` clause can be used only for creation of key collection and SQL sentence with it must be in next 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");
```

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.

8.9.2 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 seq uence 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 appropriative 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 F1
Record_var inpRecord_ = new Record;
inpRecord_->length(1);
FieldValueAccess::setString(inpRecord_[0], "test");
collection_->remove_record_with_key( inpRecord );
collection_->destroy();
```

8.10 Using of UAKGCollectionListener

In application software systems organizing of program coupling via notifying about important events in life of sonme service become a common and useful technique. For this purpose UAKGCollection provide such mechanism of "Listeners" - user can add to collection own implementation of `UAKGCollectionListener` callback interface, which would been 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 ellements 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 of events via method:

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

Method return number, which identifiy 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 notofication technique remember, that cost of collection data for notificating can be hight, and that synchronized calls of callback functions on each event is not scale. If you have situation, when N service clients must receive notifications, than do not register N listeners, but create additional element of infrastructure, which receive notification and send it's to clients, using asynchronics techniques (for example, via CORBA Event Service).

8.11 Limitations of using UAKGQuery collections interfaces

For time of now, 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).

9 Transactions

Two models of transactions are implementig un UAKQueryService:

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

XA transaction meneger is intendent to use in case, when you applications work in XA environment. UAKGQuery transaction manager is more effective.

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

9.0.1 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, whith parameters of DB 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`
- Interbase InterBase Programming Guide.

During work with XA transactions you must met next next limitations:

1. As connections to database performed by XA monitor, then only one global XA connection (i. e. login,password,db) can be used by one UAKGQuery Server. Parameters `username` nad `password` of `DBConnectionManager::createQueryManager` are ignored. So, if you want to acchieve 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 DLL statements in XA application.
3. You can't call UAKGQuery methotds outside of transaction context.
4. You must use `tread_per_request` threading mode.

9.0.2 UAKG OTS transactions

.

UAKG OTS transaction mode is usded by default.

In difference from 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 map to short local transactions.

9.1 Tupical usage of transactions

Next code fragment illustrate typical usage of OTS:

```
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 successfull or all will be complete rollbacked.

In details, trnasaction mechanizms are described in: [?] , [?].

10 IDL definitions

10.1 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>
* 1998, 1999, 2000
* $Id: ProgrammingGuide_eng.tex,v 1.13 2001/04/16 22:39:24 rssh Exp $
*/

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

#ifdef HAVE_ORB_IDL
#include <orb.idl>
#endif

#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,
    ///
    TypeLongRaw,
    ///
    TypeLongString,
    ///
    TypeWString,
    ///
    TypeLongWString
};

/**
 * 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;
};

/**
 * 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;
 };

///
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
```

10.2 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
* $Id: ProgrammingGuide_eng.tex,v 1.13 2001/04/16 22:39:24 rssh 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
```

10.3 UAKGQuery

```
#ifndef __UAKGQUERY_IDL
#define __UAKGQUERY_IDL

/*
* GradSoft specific part of CosQuery implementation.
* (C) Ruslan Shevchenko <Ruslan@Shevchenko.Kiev.UA>, 1998,1999, 2000, 2001
* (C) GradSOft 2001
* $Id: ProgrammingGuide_eng.tex,v 1.13 2001/04/16 22:39:24 rssh 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<octet> OctSeq;
    ///
    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 underlaying database
    long dbErrorCode;
   };

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


   /**
    * Hight level interface for evaluationg SQL queries
    **/
   interface QueryEvaluator
#ifdef CORBA_HAVE_OTS
                          //   :CosTransactions::TransactionalObject
                          // in all ORB-s context is passing uncoditionally
#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 queury, 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 queury, 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 descriptionas of appropriative execute parameters
 *  execute parameters are SQL host valiables.
 * 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);

/**
 * synonim 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 recod
```

```
 **/
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);

/**
 * synonim 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);

/**
 * synonim 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();

   };

//
// 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 rdonly)
                    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_expressinon - new expresiion 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, KeyNotFound);

    ///
    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_
     */
    UAKGCollection create_collection( in string queryText )
                                      raises(CosQuery::QueryInvalid,
                                             CosQuery::QueryProcessingError);

    /**
     * queryText - select <field_list> from <table_list> where <conditions> [order by <field_
```

```
     */
    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);

      ///
      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
```

10.4 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 1.13 2001/04/16 22:39:24 rssh Exp $
```

```
 */

#ifndef __COSQUERYCOLLECTION_IDL
#include <CosQueryCollection.idl>
#endif

#pragma prefix "gradsoft.kiev.ua"

/**
 * pseudo-interfaces for custom Record Masrshalling
 * The main entity is: RC-coded octet sequence, which
 * described in detail in Reference Guide.
 * 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;

/**
 * throwed, 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 fileds 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 float 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_);

  ///
  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);

};


};

#endif
```

11 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
        |      TypeWStrint    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
```

References

[1] Object Management Group, editor. *Common Object Services Specification*, chapter Transaction Service. OMG, 2000. formal/2000-06-28.

[2] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999. ISBN 0201379279.

[3] Ruslan Shevchenko. Analysis of methods of creating effecient distributed applications, based on corba standarts. *Processing of UKRPROG-2000*, 2000.

[4] X/Open. *X/Open CAE specification - Distributed Transactions Processing* . ISBN 1-872630-24-3.