Horus Database Reference

Version 2.0 - Jan 2003

Marc Navarro

Dennis Koelma

Intelligent Sensory Information Systems
University of Amsterdam, Faculty of Science
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
koelma@science.uva.nl
http://www.science.uva.nl/~horus/

Contents

Chapter 1

Video segmentations in Oracle

1.1 Table overview

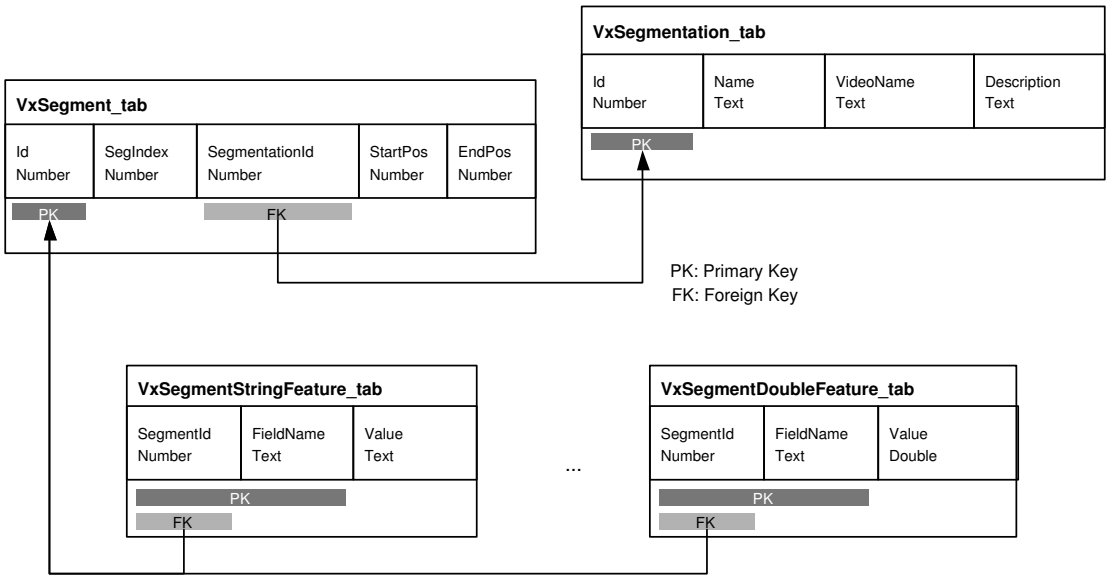

Figure 1.1: Database schema for Video Segmentations

```
CREATE TABLE VxSegmentation_tab (
    Id                NUMBER         NOT NULL,
    Name              VARCHAR2(40)   NOT NULL,
    VideoName         VARCHAR2(40)   NOT NULL,
    Description       VARCHAR2(80),
  CONSTRAINT VxSegmentation_pk   PRIMARY KEY(id),
  CONSTRAINT VxSegmentation_uniq UNIQUE(Name, VideoName)
);

CREATE TABLE VxSegment_tab (
    Id                NUMBER         NOT NULL,
    SegIndex          NUMBER         NOT NULL,
```

```
    SegmentationId    NUMBER         NOT NULL,
    StartPos          NUMBER         NOT NULL,
    EndPos            NUMBER         NOT NULL,
  CONSTRAINT VxSegment_pk   PRIMARY KEY(Id),
  CONSTRAINT VxSegment_uniq UNIQUE(SegmentationId, SegIndex),
  CONSTRAINT VxSegment_fk   FOREIGN KEY(SegmentationId)
      REFERENCES VxSegmentation_tab ON DELETE CASCADE
);

CREATE SEQUENCE VxSegmentation_id INCREMENT BY 1 START WITH 1;
CREATE SEQUENCE VxSegment_id INCREMENT BY 1 START WITH 1;

CREATE TABLE VxSegmentStringFeature_tab (
    SegmentId         NUMBER         NOT NULL,
    FieldName         VARCHAR2(20)   NOT NULL,
    Value             VARCHAR2(512),
  CONSTRAINT VxSegmentStringFeat_pk PRIMARY KEY(SegmentId, FieldName),
  CONSTRAINT VxSegmentStringFeat_fk FOREIGN KEY(SegmentId)
      REFERENCES VxSegment_tab ON DELETE CASCADE
);

CREATE TABLE VxSegmentDoubleFeature_tab (
    SegmentId         NUMBER         NOT NULL,
    FieldName         VARCHAR2(20)   NOT NULL,
    Value             NUMBER         NOT NULL,
  CONSTRAINT VxSegmentDoubleFeat_pk PRIMARY KEY(SegmentId, FieldName),
  CONSTRAINT VxSegmentDoubleFeat_fk FOREIGN KEY(SegmentId)
      REFERENCES VxSegment_tab ON DELETE CASCADE
);

CREATE TABLE VxSegmentIntFeature_tab (
    SegmentId         NUMBER         NOT NULL,
    FieldName         VARCHAR2(20)   NOT NULL,
    Value             NUMBER         NOT NULL,
  CONSTRAINT VxSegmentIntFeat_pk PRIMARY KEY(SegmentId, FieldName),
  CONSTRAINT VxSegmentIntFeat_fk FOREIGN KEY(SegmentId)
      REFERENCES VxSegment_tab ON DELETE CASCADE
);
```

All this information can be retrieved from SQL∗Plus, using the `DESCRIBE` command. Example:

```
SQL> DESCRIBE VxSegment_tab
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL NUMBER
 SEGINDEX                                  NOT NULL NUMBER
 SEGMENTATIONID                            NOT NULL NUMBER
 STARTPOS                                  NOT NULL NUMBER
 ENDPOS                                    NOT NULL NUMBER
```

Constraint information can also be obtained from SQL∗Plus, querying data dictionary views:

```
SQL> SELECT c1.constraint_name, c1.constraint_type, c1.delete_rule, c2.column_name,
  c1.r_constraint_name
FROM user_constraints c1, user_cons_columns c2
WHERE (c1.table_name='VXSEGMENT_TAB') AND (c1.constraint_name = c2.constraint_name)
  AND (c1.generated='USER NAME');

CONSTRAINT_NAME      TYPE DEL_RULE  COLUMN_NAME          R_CONSTRAINT_NAME
-------------------- ---- --------- -------------------- --------------------
VXSEGMENT_PK         P              ID
VXSEGMENT_UNIQ       U              SEGINDEX
VXSEGMENT_UNIQ       U              SEGMENTATIONID
VXSEGMENT_FK         R    CASCADE   SEGMENTATIONID       VXSEGMENTATION_PK
```

1.2 Table VxSegmentation_tab

```
CREATE TABLE VxSegmentation_tab (
    Id                NUMBER         NOT NULL,
    Name              VARCHAR2(40)   NOT NULL,
    VideoName         VARCHAR2(40)   NOT NULL,
    Description       VARCHAR2(80),
  CONSTRAINT VxSegmentation_pk   PRIMARY KEY(id),
  CONSTRAINT VxSegmentation_uniq UNIQUE(Name, VideoName)
);
```

VxSegmentation_tab contains information about the segmentations: name of segmentation, name of the video and an optional description. Each segmentation stored in the database receives an Id used to reference a segmentation from other tables.

Query examples:

```
SQL> SELECT * FROM VxSegmentation_tab;

 ID NAME                 VIDEONAME                      DESCRIPTION
--- -------------------- ------------------------------ --------------------
 61 blocks               aviewtoakill_cd1
 22 anchorman            20010125_20uur_journaal
 23 blocks               20010125_20uur_journaal
 24 blocks_flash         20010125_20uur_journaal
 25 blocks_kineticEnergy 20010125_20uur_journaal
 26 cam                  20010125_20uur_journaal
 27 dst                  20010125_20uur_journaal
 28 faceout              20010125_20uur_journaal
 29 faces                20010125_20uur_journaal
 30 genre                20010125_20uur_journaal
 62 cam                  aviewtoakill_cd1
 63 dst                  aviewtoakill_cd1
 64 raw.blocks           aviewtoakill_cd1
 65 scenes               aviewtoakill_cd1
 66 scenes.hanjalic      aviewtoakill_cd1
 67 scenes.rui           aviewtoakill_cd1
 68 scenes.yeung         aviewtoakill_cd1
 69 trackout             aviewtoakill_cd1
 70 blocks               friends418
 71 cam                  friends418
 72 dst                  friends418
 73 raw.blocks           friends418
 74 scenes               friends418
 75 scenes.hanjalic      friends418
 76 scenes.rui           friends418
 77 scenes.yeung         friends418
 55 ling_topic           20010125_20uur_journaal
 56 persons              20010125_20uur_journaal
 57 silence              20010125_20uur_journaal
 58 teletekst            20010125_20uur_journaal
 59 trackout             20010125_20uur_journaal

31 rows selected.
```

Use the following commands to format the query output as shown by the example:

```
COLUMN id FORMAT 99
COLUMN name FORMAT a20
COLUMN videoname FORMAT a30
COLUMN description FORMAT a20
```

Next query will sort the results:

```
SQL> SELECT * FROM VxSegmentation_tab ORDER BY VideoName, Name;
```

And this one will show only the name of segmentations of video 'friends418':

```
SQL> SELECT Name FROM VxSegmentation_tab WHERE VideoName='friends418';
```

1.2.1 Sequence VxSegmentation_id

```
CREATE SEQUENCE VxSegmentation_id INCREMENT BY 1 START WITH 1;
```

This sequence has been defined to assign Ids to new segmentations. Example:

```
INSERT INTO VxSegmentation_tab
 SELECT VxSegmentation_id.NEXTVAL, 'test', 'friends418', 'A test' FROM dual;
```

This statement will introduce a new segmentation to VxSegmentation_tab. To retrieve the last assigned id, use:

```
SELECT VxSegmentation_id.CURRVAL FROM dual;
```

dual is a table that only has one row. It's useful to evaluate expressions, like:

```
SQL> SELECT sysdate FROM dual;

SYSDATE
---------
04-JUL-02

SQL> SELECT sin(3) FROM dual;

    SIN(3)
----------
.141120008
```

sysdate and *sin(3)* are not columns of the *dual* table. We can also select *sysdate* and *sin(3)* from VxSegmentation_tab, but we will obtain the result 31 times. We use *dual* to get the result just one time. That's the purpose of the *dual* table.

We will now remove the segmentation we have added:

```
SQL> DELETE FROM VxSegmentation_tab WHERE (videoName='friends418') AND (Name='test');
```

1.3 Table VxSegment_tab

```
CREATE TABLE VxSegment_tab (
    Id                NUMBER         NOT NULL,
    SegIndex          NUMBER         NOT NULL,
    SegmentationId    NUMBER         NOT NULL,
    StartPos          NUMBER         NOT NULL,
    EndPos            NUMBER         NOT NULL,
  CONSTRAINT VxSegment_pk   PRIMARY KEY(Id),
```

```
  CONSTRAINT VxSegment_uniq UNIQUE(SegmentationId, SegIndex),
  CONSTRAINT VxSegment_fk   FOREIGN KEY(SegmentationId)
      REFERENCES VxSegmentation_tab ON DELETE CASCADE
);

CREATE SEQUENCE VxSegment_id INCREMENT BY 1 START WITH 1;
```

A segmentation consists on a list of segments. Segments are stored in the table VxSegment_tab. For each segment we store its index (the position where it appears in the segment list, beggining with 0), the Id of the **segmentation** (p. 4) it belongs to and its start and end time. An Id is also assigned to each segment. The sequence VxSegment_id is used to assign new Ids.

Query examples:

Select all segments from 'blocks' segmentation of 'friends418';

```
SQL> SELECT s.SegIndex, s.StartPos, s.EndPos,
FROM VxSegmentation_tab seg, VxSegment_tab s
WHERE (seg.Id = s.segmentationId) AND (seg.VideoName = 'friends418')
AND (seg.Name = 'blocks');
```

Select segments that end before frame 50:

```
SQL> SELECT seg.VideoName, seg.Name, s.SegIndex, s.StartPos, s.EndPos
FROM VxSegmentation_tab seg, VxSegment_tab s
WHERE (seg.Id = s.segmentationId) AND (s.endPos < 50);
```

1.4 Table VxSegmentStringFeature_tab

```
CREATE TABLE VxSegmentStringFeature_tab (
    SegmentId         NUMBER         NOT NULL,
    FieldName         VARCHAR2(20)   NOT NULL,
    Value             VARCHAR2(512),
  CONSTRAINT VxSegmentStringFeat_pk PRIMARY KEY(SegmentId, FieldName),
  CONSTRAINT VxSegmentStringFeat_fk FOREIGN KEY(SegmentId)
      REFERENCES VxSegment_tab ON DELETE CASCADE
);
```

The features of each **segment** (p. 5) are stored in **Table VxSegmentStringFeature_tab** (p. 6), and **Tables VxSegmentDoubleFeature_tab and VxSegmentIntFeature_tab** (p. 8) depending on their type. We will introduce first the table to store string features: VxSegmentStringFeature_tab. This table contains the Id of the segment, the name of the feature and its value.

Query examples:

List all string features from a segmentation:

```
SQL> SELECT DISTINCT f.FieldName
FROM VxSegmentation_tab seg, VxSegment_tab s, VxSegmentStringFeature_tab f
WHERE (seg.Id = s.segmentationId) AND (seg.VideoName = 'friends418')
 AND (seg.Name = 'blocks') AND (f.SegmentId = s.Id);
```

List blocks of 'friends418' with type 'effect':

```
SQL> SELECT s.SegIndex, s.StartPos, s.EndPos
FROM VxSegmentation_tab seg, VxSegment_tab s, VxSegmentStringFeature_tab f
WHERE (seg.Id = s.segmentationId) AND (seg.VideoName = 'friends418')
  AND (seg.Name = 'blocks') AND (f.SegmentId = s.Id) AND (f.FieldName = 'type')
  AND (f.Value = 'effect');
```

List blocks of 'friends418' with all their string features:

```
SQL> SELECT s.segIndex, s.startPos, s.endPos, f0.value description,
  f1.value effect, f2.value type
FROM VxSegmentation_tab seg, VxSegment_tab s,
  VxSegmentStringFeature_tab f0, VxSegmentStringFeature_tab f1,
  VxSegmentStringFeature_tab f2
WHERE (seg.videoName = 'friends418') AND (seg.Id = s.segmentationId)
  AND (seg.name = 'blocks')
  AND (f0.segmentId (+)= s.Id) AND (f0.fieldName (+)= 'description')
  AND (f1.segmentId (+)= s.Id) AND (f1.fieldName (+)= 'effect')
  AND (f2.segmentId (+)= s.Id) AND (f2.fieldName (+)= 'type');
```

Use the following result formatting to get a better display of results:

```
COLUMN segIndex HEADING '#'
COLUMN segIndex FORMAT 999
COLUMN startPos FORMAT 999999
COLUMN endPos FORMAT 999999
COLUMN description FORMAT a30
COLUMN effect FORMAT a10
COLUMN type FORMAT a10
```

Segmentation views:

The previous query can be introduced as a view, and then be able to perform queries using a simpler statement (but maybe slower):

```
SQL> CREATE VIEW VxBlocksSegment_tab AS
SELECT seg.Id, s.segmentationId, s.segIndex, s.startPos, s.endPos,
  f0.value description, f1.value effect, f2.value type
FROM VxSegmentation_tab seg, VxSegment_tab s,
  VxSegmentStringFeature_tab f0, VxSegmentStringFeature_tab f1,
  VxSegmentStringFeature_tab f2
WHERE (seg.Id = s.segmentationId) AND (seg.name = 'blocks')
  AND (f0.segmentId (+)= s.Id) AND (f0.fieldName (+)= 'description')
  AND (f1.segmentId (+)= s.Id) AND (f1.fieldName (+)= 'effect')
  AND (f2.segmentId (+)= s.Id) AND (f2.fieldName (+)= 'type');
```

Example of querying VxBlocksSegment_tab view:

```
SQL> SELECT b.segIndex, b.startPos, b.endPos, b.effect
FROM VxSegmentation_tab seg, VxBlocksSegment_tab b
WHERE (b.segmentationId = seg.Id) AND (seg.VideoName = 'friends418')
  AND (b.type = 'effect');
```

And here is the same query, without using the view:

```
SQL> SELECT s.segIndex, s.startPos, s.endPos, f1.value effect
FROM VxSegmentation_tab seg, VxSegment_tab s,
  VxSegmentStringFeature_tab f1, VxSegmentStringFeature_tab f2
WHERE (s.segmentationId = seg.Id) AND (seg.VideoName = 'friends418')
  AND (seg.Name = 'blocks') AND (f1.SegmentId = s.Id)
  AND (f2.SegmentId = s.Id) AND (f2.FieldName = 'type')
  AND (f2.Value = 'effect') AND (f1.FieldName = 'effect');
```

1.5 Tables VxSegmentDoubleFeature_tab and VxSegmentIntFeature_tab

```
CREATE TABLE VxSegmentDoubleFeature_tab (
    SegmentId         NUMBER         NOT NULL,
    FieldName         VARCHAR2(20)   NOT NULL,
    Value             NUMBER         NOT NULL,
  CONSTRAINT VxSegmentDoubleFeat_pk PRIMARY KEY(SegmentId, FieldName),
  CONSTRAINT VxSegmentDoubleFeat_fk FOREIGN KEY(SegmentId)
      REFERENCES VxSegment_tab ON DELETE CASCADE
);

CREATE TABLE VxSegmentIntFeature_tab (
    SegmentId         NUMBER         NOT NULL,
    FieldName         VARCHAR2(20)   NOT NULL,
    Value             NUMBER         NOT NULL,
  CONSTRAINT VxSegmentIntFeat_pk PRIMARY KEY(SegmentId, FieldName),
  CONSTRAINT VxSegmentIntFeat_fk FOREIGN KEY(SegmentId)
      REFERENCES VxSegment_tab ON DELETE CASCADE
);
```

VxSegmentIntFeature_tab and VxSegmentDoubleFeature_tab are similar to **VxSegmentStringFeature_tab** (p. 6). The only difference is the type of the *Value* column.

Query examples:

Get ”zoom” segments of ”cam” segmentation of ”friends418” with positive factor:

```
SQL> SELECT s.segIndex, s.startPos, s.endPos, factorFeat.value factor
FROM VxSegmentation_tab seg, VxSegment_tab s,
  VxSegmentStringFeature_tab typeFeat, VxSegmentDoubleFeature_tab factorFeat WHERE (seg.videoName = 'frien
  AND (seg.id = s.segmentationId) AND (typeFeat.segmentId = s.id)
  AND (typeFeat.fieldName = 'type') AND (typeFeat.value = 'zoom')
  AND (factorFeat.segmentId = s.id) AND (factorFeat.fieldName = 'factor')
  AND (factorFeat.value >= 0.0);
```

Get ”zoom” segments of ”cam” segmentation of ”friends418” with positive factor, only in scenes with ”description” beginning with ”Ross”:

```
SQL> SELECT sCam.segIndex, sCam.startPos, sCam.endPos
FROM VxSegment_tab sCam, VxSegmentation_tab segCam,
  VxSegmentStringFeature_tab fType, VxSegmentDoubleFeature_tab fFactor,
  VxSegment_tab sScenes, VxSegmentation_tab segScenes,
  VxSegmentStringFeature_tab fDescr
WHERE (segCam.videoName = 'friends418')
  AND (sCam.segmentationId = segCam.id) AND (segCam.name = 'cam')
  AND (fType.segmentId = sCam.id) AND (fType.fieldName = 'type')
  AND (fType.value = 'zoom') AND (fFactor.segmentId = sCam.id)
  AND (fFactor.fieldName = 'factor') AND (fFactor.value >= 0.0)
  AND (segScenes.videoName = 'friends418')
  AND (sScenes.segmentationId = segScenes.id) AND (segScenes.name = 'scenes')
  AND (fDescr.segmentId = sScenes.id) AND (fDescr.fieldName = 'description')
  AND (fDescr.value LIKE 'Ross%')
  AND (sCam.startPos >= sScenes.startPos) AND (sCam.endPos <= sScenes.endPos);
```

Chapter 2

CORBA Server for Databases

2.1 Introduction

The first reason to develop a CORBA server to access a Database was to be able to store video segmentations in an Oracle Database. The data structures of the video segmentations was ideal to be stored in a relational database, and Oracle was choosen. And a CORBA server was develop to be able to access this data from any other component of the framework. **OracleServer** (p. 16) was born.

One of the important points in the development of **OracleServer** (p. 16) was the query functionality. Relational databases are queried via SQL, but SQL is not easy to map to CORBA. Some functions has been introduced to perform easy queries without the complexity of SQL, and other mechanisms have been proved to allow users use the full potential of SQL to query the database. These functions and mechanisms will be described in detail in the **IDL Definitions** (p. 9) section.

Oracle is not the only database available, and it's not the most suitable for all platforms. We wanted to be able to develop a CORBA server for any other database, so the components of the **OracleServer** (p. 16) (DatabaseServer classes) were designed such a way that they can be reused for the development of a server for another database, like MySQL. Section **Java Structure** (p. 14) explains the components (Java classes) that have been created for developing a CORBA server for a Database and how to use these componets to build the **OracleServer** (p. 16) or the **MySQLServer** (p. 17).

Finally, in the development of the DatabaseServer was taken into account the possibility of handle other data different to video segmentations, as will be shown in the **IDL Definitions** (p. 9) section.

2.2 IDL Definitions

Here is the definition of the CORBA interfaces for the **OracleServer** (p. 16) (file **HxCorbaDatabase.idl**):

```
module HxCorba
{
    exception DatabaseException
    {
        long            dbCode;
        string          dbMessage;
        string          message; // maybe more friendly
    };

    struct SegmentQueryResult
    {
        string      videoName;
```

```
        string      segmentationName;
        VxSegment   segment;
        VxTimeSpan  time;
    };

    typedef sequence<SegmentQueryResult> SegmentQueryResultSeq;

    interface DatabaseSession
    {
        StringSeq           listVideos();
        StringSeq           listSegmentations(in string videoName);
        VxSegmentation      getSegmentation(in string videoName, in string segName);

        StringSeq           queryStrings(in string sqlQuery)
                                raises(DatabaseException);

        VxSegmentSeq        querySegments(in string sqlQuery)
                                raises(DatabaseException);

        SegmentQueryResultSeq queryMultipleSegments(in string sqlQuery)
                                raises(DatabaseException);

        void                close();
    };

    interface Database
    {
        DatabaseSession   openSession(in string username, in string password)
                            raises(DatabaseException);
    };
};
```

The `Database` interface is the entry point. It allows to initiate a database session, providing identification.

The `DatabaseSession` interface offers funtions to perform easy queries on video segmentations : `listVideos`, `listSegmentations` and `getSegmentation`.

`queryStrings`, `querySegments` and `queryMultipleSegments` can be used to query the database with SQL, although each function determines the result type of the query. Only SQL queries that return that type are allowed.

Subclasses of `DatabaseSession` have been defined to offer other functionality: adding and modifying video segmentations, more powerful query mechanisms and accessing other type of data.

2.2.1 Other Database Sessions

Here is the definition of other database sessions (file **HxCorbaDatabaseSessions.idl**):

```
module HxCorba
{
    ////////////////////////////////////////////////////////////////
    //
    // StoreSession
    //
    ////////////////////////////////////////////////////////////////

    interface VxSegmentBuilder
    {
        void                addInt(in string id, in long value)
                                raises(DatabaseException);
        void                addDouble(in string id, in double value)
                                raises(DatabaseException);
```

```
    void                addString(in string id, in string value)
                            raises(DatabaseException);
};

interface VxSegmentationBuilder
{
    void                setDescription(in string description);
    VxSegmentBuilder    buildSegment(in long start, in long end)
                            raises(DatabaseException);
};

interface StoreSession : DatabaseSession
{
    void                addSegmentation(in VxSegmentation seg,
                            in string videoName, in string segName,
                            in string description)
                                raises(DatabaseException);

    VxSegmentationBuilder buildSegmentation(in string videoName,
        in string segName) raises(DatabaseException);
};


//////////////////////////////////////////////////////////////
//
// UpdateSession
//
//////////////////////////////////////////////////////////////

interface VxMutableSegment : VxSegmentBuilder, VxSegment
{
    void    setStart(in long start);
    void    setEnd(in long end);

    void    removeInt(in string id) raises(DatabaseException);
    void    removeDouble(in string id) raises(DatabaseException);
    void    removeString(in string id) raises(DatabaseException);

    void    changeInt(in string id, in long newValue)
                raises(DatabaseException);
    void    changeDouble(in string id, in double newValue)
                raises(DatabaseException);
    void    changeString(in string id, in string newValue)
                raises(DatabaseException);
};

interface VxMutableSegmentation : VxSegmentationBuilder, VxSegmentation
{
    void    removeSegment(in long index) raises(DatabaseException);
};

interface UpdateSession : StoreSession
{
    void    removeVideo(in string videoName) raises(DatabaseException);
    void    removeSegmentation(in VxSegmentation seg) raises(DatabaseException);
    void    removeSegment(in VxSegment segment) raises(DatabaseException);
};


//////////////////////////////////////////////////////////////
//
// XMLSession
//
//////////////////////////////////////////////////////////////
```

```
enum DBDataTag { DBINT, DBDOUBLE, DBSTRING, DBSEGMENTATION, DBSEGMENT };

union DBData switch (DBDataTag) {
case DBINT:
    long intData;
case DBDOUBLE:
    double doubleData;
case DBSTRING:
    string stringData;
case DBSEGMENTATION:
    VxSegmentation segmentation;
case DBSEGMENT:
    VxSegment segment;
};

typedef sequence<DBDataTag> DBDataTagSeq;
typedef sequence<DBData> DBDataRow;
typedef sequence<DBDataRow> DBDataRowSeq;

interface XMLSession : DatabaseSession
{
    string queryXML(in string sqlQuery) raises(DatabaseException);

    DBDataRowSeq queryDBData(in string sqlQuery, in DBDataTagSeq resultType)
        raises(DatabaseException);
};


//////////////////////////////////////////////////////////////////
//
// HistogramSession
//
//////////////////////////////////////////////////////////////////

interface HistogramSession : DatabaseSession
{
    void        addHistogram(in string imageName, in string setName,
                    in FloatSeq histoData) raises(DatabaseException);
    FloatSeq    getHistogram(in string imageName, in string setName)
                    raises(DatabaseException);
    StringSeq   nearest(in string imageName, in string setName, in long count)
                    raises(DatabaseException);
    StringSeq   random(in string setName, in long count)
                    raises(DatabaseException);

    StringSeq   search(in long count, in FloatSeq sample)
                    raises(DatabaseException);
};


//////////////////////////////////////////////////////////////////
//
// VxSimilaritySession
//
//////////////////////////////////////////////////////////////////

interface VxSimilarityBuilder
{
    void        addSimilarity(in long index1, in long index2, in double value,
                    in long keyFrame1, in long keyFrame2)
                        raises(DatabaseException);
};

interface VxSimilaritySession : DatabaseSession
{
```

```
        VxSimilarityBuilder addSimilarities(in string videoName, in string segName, in string featureName)
                        raises(DatabaseException);
    };


    ////////////////////////////////////////////////////////////
    //
    // FullSession
    //
    ////////////////////////////////////////////////////////////

    interface FullSession : UpdateSession, XMLSession, HistogramSession,
                VxSimilaritySession
    {
    };
};
```

Functionality of `DatabaseSession` has been splitted in different subclasses to structure it a little and to allow different database servers implement different functionality. That way, existing servers doesn't have to be modified when new functionality is added.

Users have to check which interfaces are supported by the `DatabaseSession` object returned by `Database.openSession` operation to see which functionality is supported by the server.

The server can even decide to return different subclasses of `DatabaseSessions` depending on the identification provided by the user: normal users can receive only a basic `DatabaseSession`, but advanced users can get instead an `UpdateSession` because they are allowed to modify the contents of the database.

Here is a small description of each session type:

StoreSession:

It allows to add new video segmentations to the database, but not modify existing ones.

UpdateSession:

It offers all the functionality of `StoreSession` and also allows to modify existing segmentations.

To modify a segmentation, first obtain one with `DatabaseSession.getSegmentation` and downcast it to `VxMutableSegmentation`.

XMLSession:

These session has a function to test the retrieval of query results through XML, and also offers a mechanism to perform SQL queries that return any type. "any type" is not really true: only rows of int, double, strings, VxSegmentations and VxSegments can be retrieved (in any order and any amount) but it can be easily be extended to any other type if required.

`XMLSession.queryDBData` takes an SQL query and a description of the rows returned by the query (as a sequence of `DBDataTag`), and returns a sequence of rows. A row is a sequence of unions that encapsulate the allowed types. Section **Tools and Examples** (p. 17) contains an example of using `queryDBData`.

HistogramSession:

This is an example of how to access other data than video segmentations. The functionality to access this other data is moved out from the base `DatabaseSession` and that way not all CORBA servers that implement the `Database` interface are forced to support histogram sessions.

The server can even decide to return a `HistogramSession` object only if the user scheema contains the tables required to handle histograms. *(This is not implemented yet by any server, but it's not a bad idea).*

VxSimilaritySession:

Another example of storing other kind of data. To retrieve similarities, use the querying functions of `XMLSession`.

FullSession:

This interface has been defined to access servers that support all the interfaces, so all the functionality can be accessed through just one interface

2.3 Java Structure

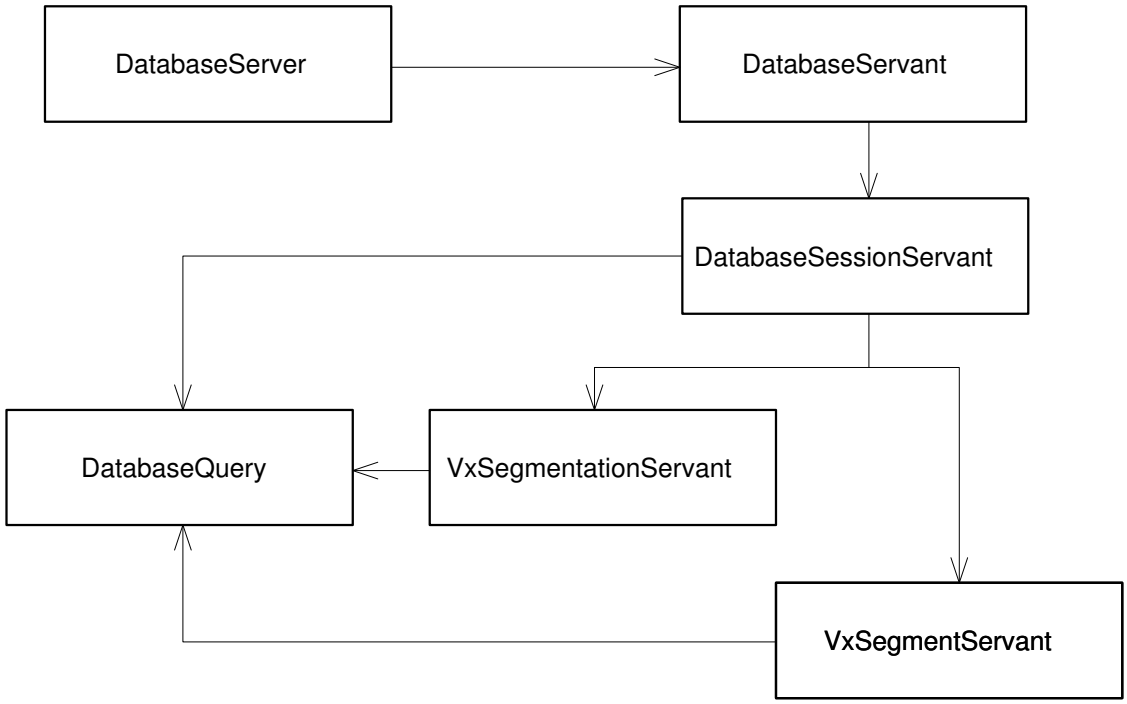

Figure 2.1: Java Structure of DatabaseServer

- `DatabaseServer` is the main class. It initializes the ORB and registers the `DatabaseServant`. It also offers useful methods to the rest of classes.

  `DatabaseServer` is an abstract class. Subclasses have to implement the method that creates the `DatabaseSessionServant` (`DatabaseServer::makeDatabaseSessionServant`), because different servers support different kind of sessions (and different `DatabaseSession` servants then).

  It contains an inner private class (`DatabaseServant`) that implements `HxCorba.DatabaseOperations`. It has just the `openSession` method, that returns a DatabaseSession object, created through the abstract method `DatabaseServer::makeDatabaseSessionServant` .
- `DatabaseSessionServant` is the base class for session servants. It implements `HxCorba.UpdateSessionOperations`.

  It's an abstract class. Subclasses have to redefine the method that connects to the Database through JDBC (DatabaseSessionServant::makeConnection).

  DatabaseSessionServant can also be subclassed to implement a different kind of database session (**OracleServer** (p. 16) subclasses DatabaseSessionServant to implement a `FullSession`) or to

implement the `UpdateSession` operations in a different way (like **MySQLServer** (p. 17) does).
- VxSegmentationServant implements HxCorba.VxMutableSegmentationOperations, and VxSegmentServant implements HxCorba.VxMutableSegmentOperations.
- DatabaseQuery is a helper class that supports querying though JDBC.

2.3.1 Operations interfaces and Tie servants

Usually servants for a *Aaa* CORBA Object are implemented as classes that extend the corresponding *Aaa*POA class (generated by the compiler), but all the servants described above implement *Aaa*Operations instead. Why?

This is that way to be able to implement a servant for a *Bbb* CORBA Object (where the interface *Bbb* inherits from the interface *Aaa*) extending the servant for *Aaa* (because we don't want to implement again the operations of *Aaa* in the servant for *Bbb*).

*Aaa*Servant class extends *Aaa*POA class, but *Bbb*Servant class can not be implemented extending both *Aaa*Servant and *Bbb*POA, because Java classes can only extend one class. The solution is implementing *Bbb*Servant as a class that extends *Aaa*Servant and implements *Bbb*Operations. This *Bbb*Servant class is not yet a servant for *Bbb*, because *Bbb*POA is not in its inheritance tree. But it can be used to instantiate a *Bbb*POATie class, and then we have finally a servant for *Bbb*.

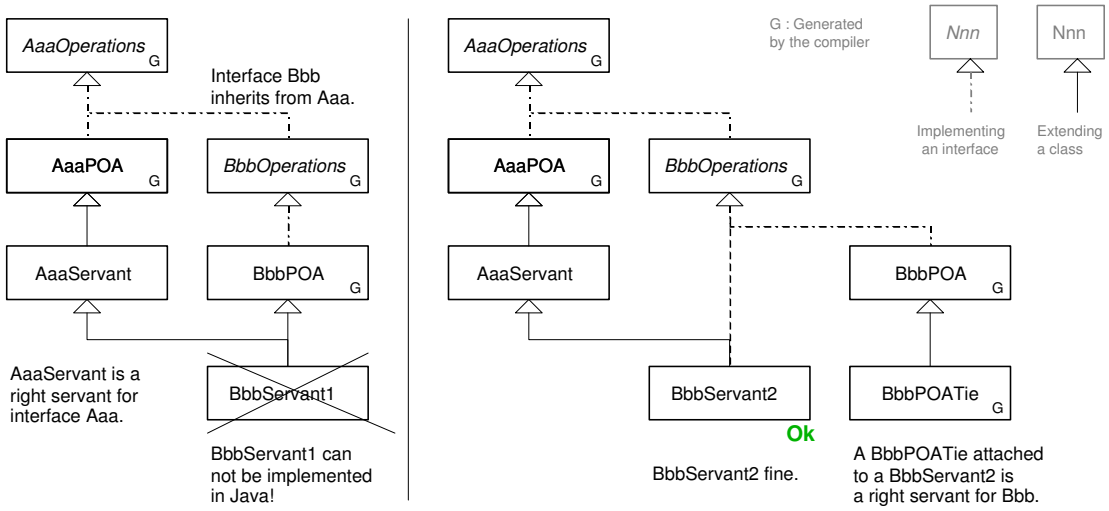

Figure 2.2: Using Tie classes to implement extensible servants

To be consistent, *Aaa*Servant could be implement *Aaa*Operations instead of extending *Aaa*POA, and be used with a *Aaa*POATie. Then all servants are implemented the same way.

2.3.2 Default servants

From a client point of view, the server contains a different CORBA object for each "entity" (or "entry") of the database: an object for each segmentation, an object for each segment, etc...

But the server can not keep a servant for each "entry" of the database: it would be quite expensive.

The solution is to use a single servant for all "entities" of the same type. These servants are called Default servants. We will have a default servant for segmentations (`VxSegmentationServant`), a default servant for segments (`VxSegmentServant`) and so on.

The default servant implements the functionality of multiple objects (all of the same type). These objects have different ”state” (the ”state” of the object is what we store about it in the database), and the result of the operations of the objects depend on their states. Then, the default servant needs to know which object is calling the client. This is done through the `ObjectId`.

Each CORBA Object of a server has a different `ObjectId`. Although objects exist in the server only in a ”logical” point of view, not ”physically” (because there is no a servant for each object) they still do have an `ObjectId`. And our servers will choose an `ObjectId` that will help in accessing the ”state” of the objects: `ObjectIds` will usually be the primary keys of the tables storing the objects.

When a default servant receives a request, it will obtain the `ObjectId` of the object that is being called, access the ”state” of the object through this `ObjectId` and process the request based on this ”state”.

Next we will see how **OracleServer** (p. 16) and **MySQLServer** (p. 17) have been built using all these classes.

2.4 OracleServer

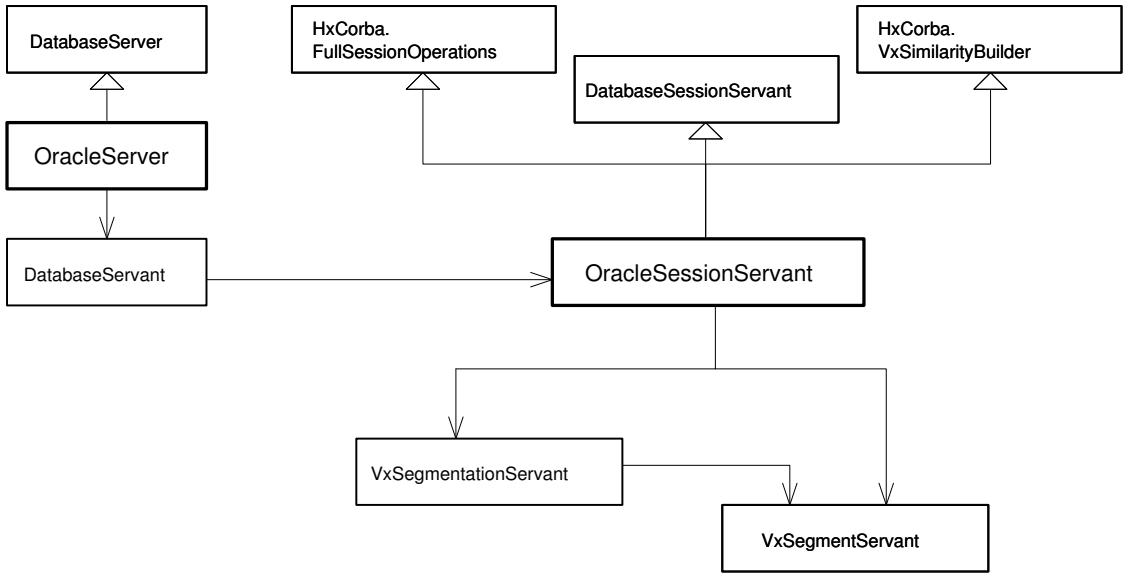

Figure 2.3: Java Structure of OracleServer

`OracleServer` is the main class. It inherits from `DatabaseServer` and redefines `makeDatabaseSessionServant` to return an `OracleSessionServant` as a servant for `DatabaseSession`. It also instantiates the JDBC driver for Oracle.

`OracleSessionServant` extends the base `DatabaseSessionServant` and implements also all the functionality of `XMLSession`, `HistogramSession` and `VxSimilaritySession`. This class extends also `HxCorba.VxSimilarityBuilderOperations`, so it can be used as default servant for `VxSimilarityBuilder`.

`OracleServer` uses the base `VxSegmentationServant` and `VxSegmentServant`. They work fine for Oracle.

2.5 MySQLServer

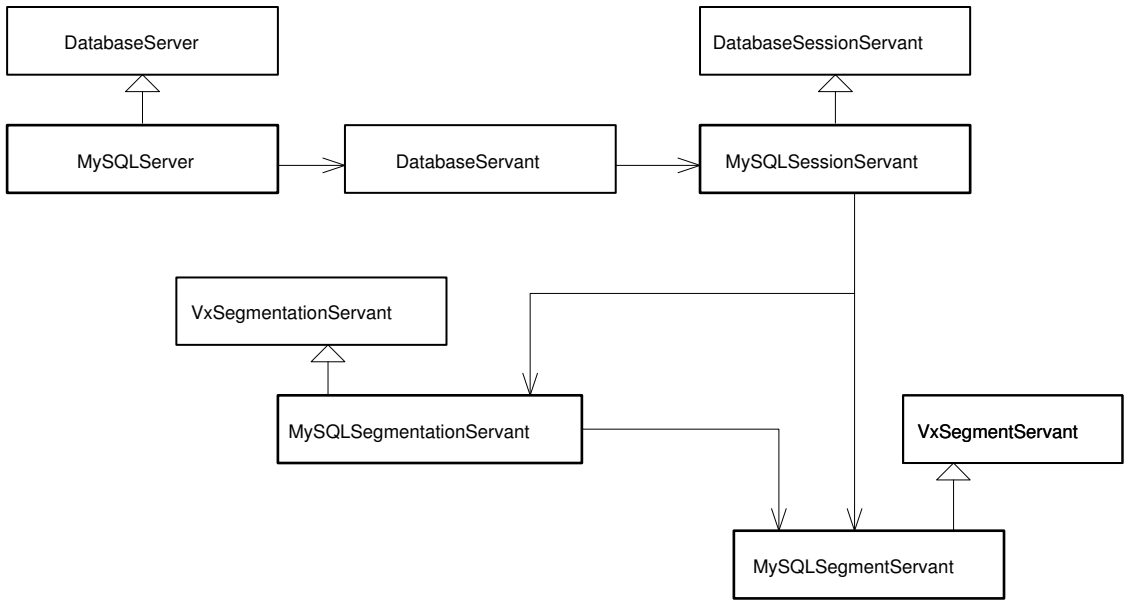

Figure 2.4: Java Structure of MySQLServer

`MySQLServer` is the main class. It inherits from `DatabaseServer` and redefines `makeDatabaseSessionServant` to return a `MySQLSessionServant` as a servant for `DatabaseSession`. It also instantiates the JDBC driver for MySQL.

`MySQLSessionServant` extends `DatabaseSessionServant`. It doesn't implement any other kind of session (XML, Histogram and Similarity stuff still haven't been ported to MySQL) but it redefines some methods of `DatabaseSessionServant` to adapt it to MySQL (there are some things missed in the SQL implementation in MySQL, so the way `DatabaseSessionServant` implement some functions doesn't work for MySQL).

For the same reason, `MySQLSegmentationServant` and `MySQLSegmentServant` have been implemented. They just redefine some methods of `VxSegmentationServant` and `VxSegmentServant` to implement them in the "MySQL way".

`MySQLSessionServant` redefines `makeSegmentationDefSvt` and `makeSegmentDefSvt` to set `MySQLSegmentationServant` and `MySQLSegmentServant` as default servants.

2.6 Tools and Examples

- **DatabaseApp** (p. 18)
- **QuerySegmentations** (p. 19)
- **ClientQueryDemo** (p. 21)

2.6.1 DatabaseApp

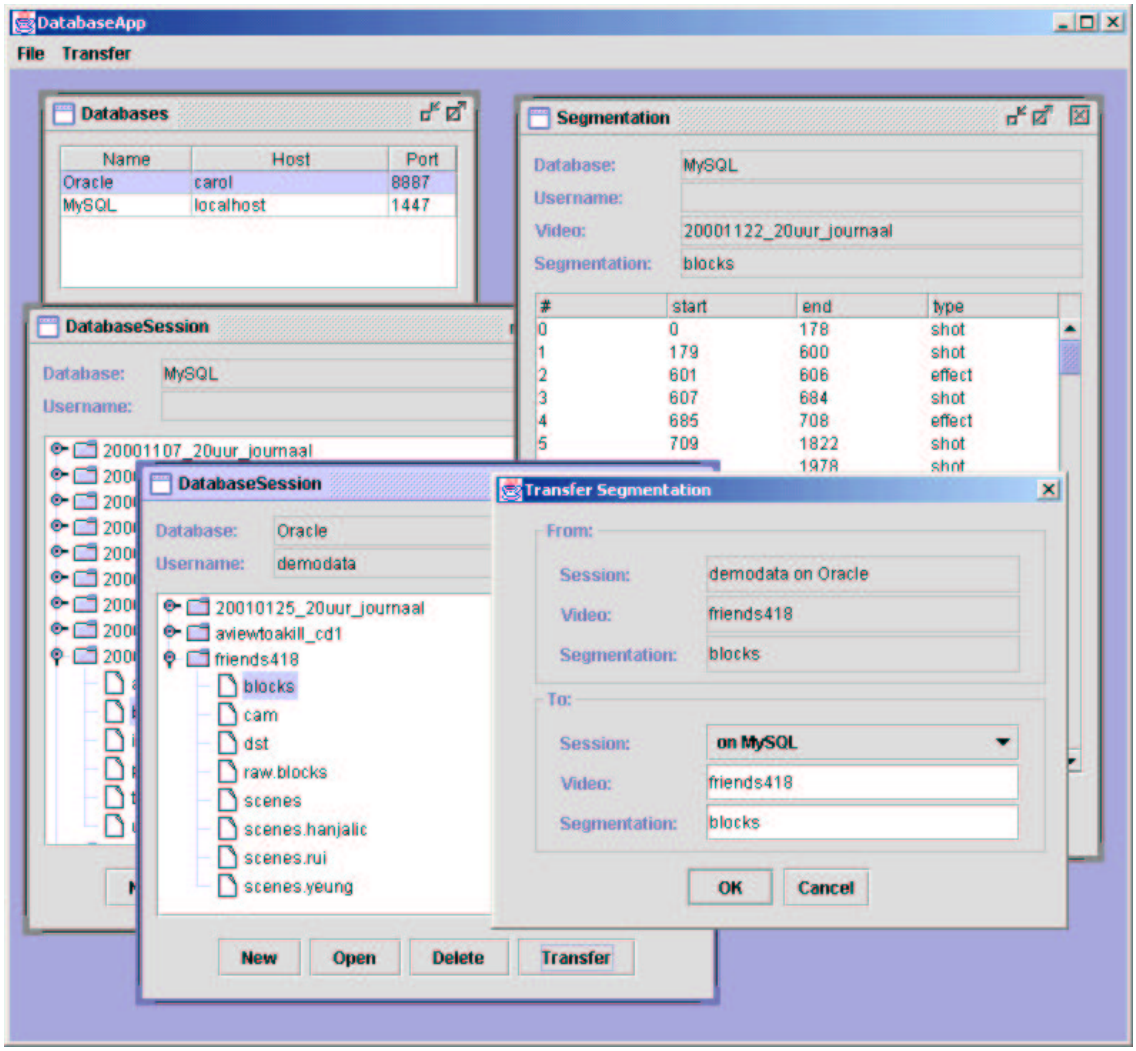

Figure 2.5: DatabaseApp

DatabaseApp is a tool to manage video segmentations stored in databases. It can contact with different servers, list segmentations stored in those servers, view and modify segmentations and transfer segmentations from one database to another.

DatabaseApp can also store in the database a segmentation file (with the help of a HxServer).

2.6.2 QuerySegmentations

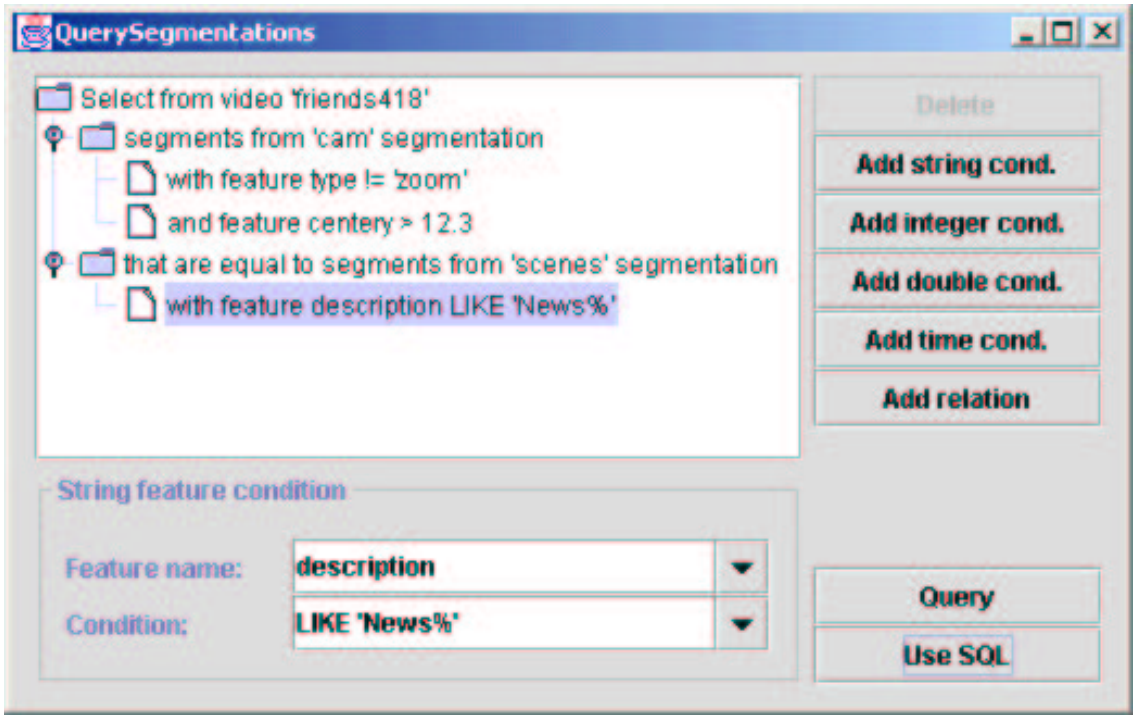

Figure 2.6: QuerySegmentations

DatabaseApp is a tool to query video segmentations stored in databases. It offers a friendly graphical interface to build complex queries without the need to use SQL.

It also offers a SQL mode for experienced users and can translate a graphical query into a SQL query for users who want to learn SQL.

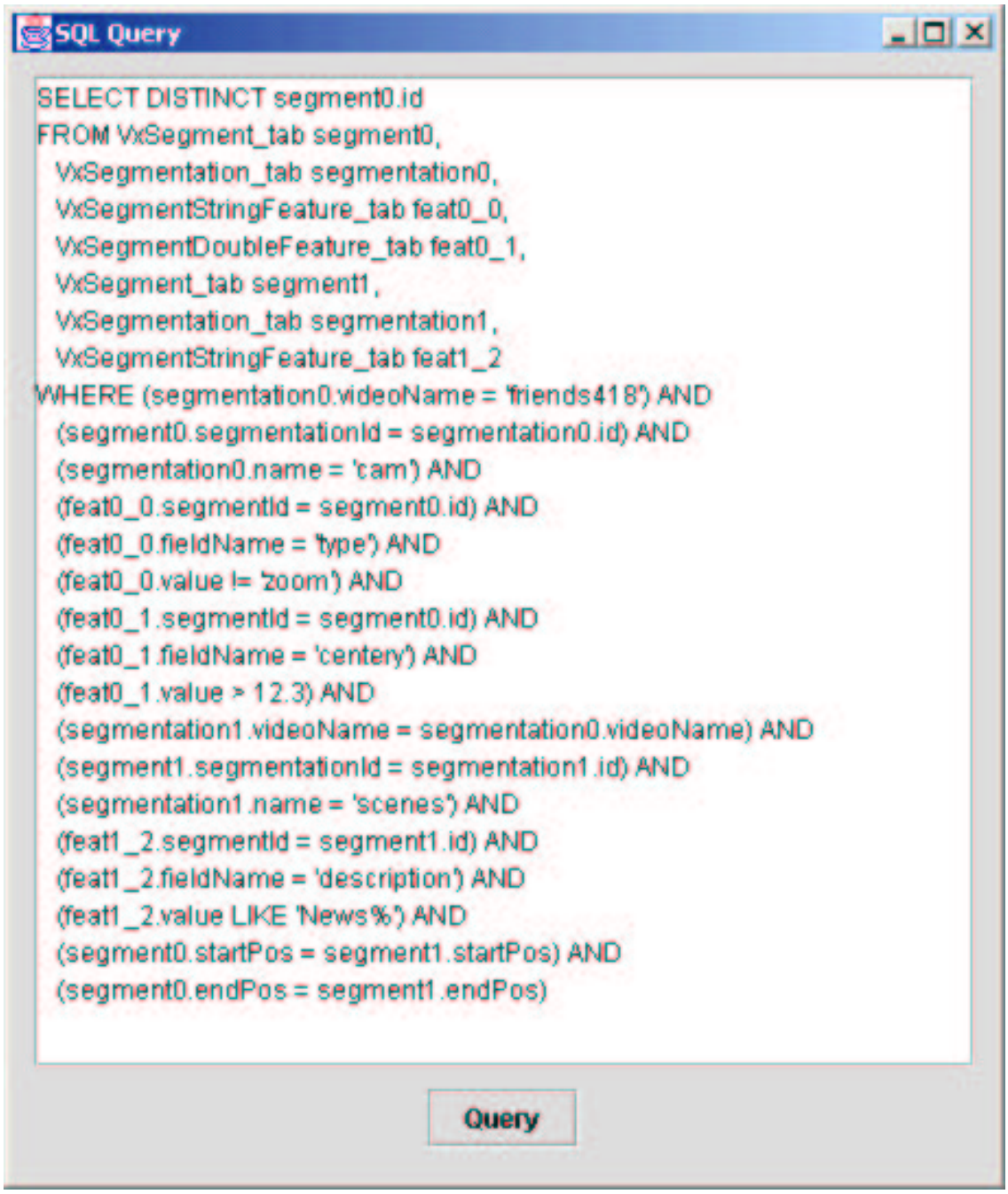

Figure 2.7: Using SQL with QuerySegmentations

2.6.3 ClientQueryDemo

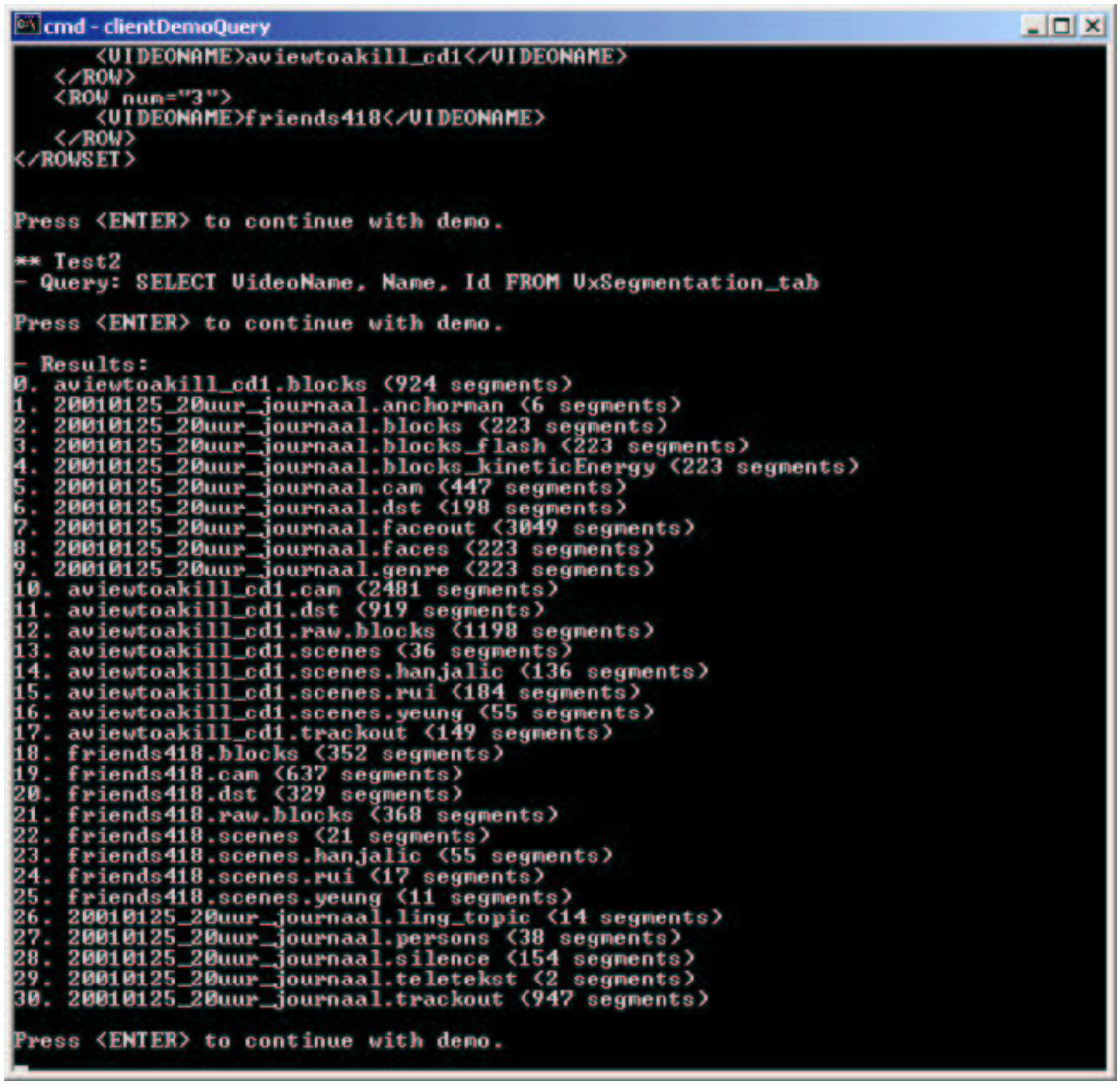

Figure 2.8: ClientQueryDemo

ClientQueryDemo is a C++ console app that demonstrates how to use the `queryXML` and `queryDBData` operations from `XMLSession` (with these operations users can query the database and get back any kind of result).

Chapter 3

Oracle maintenance

3.1 Adding users to Oracle

This appendix shows how to add new users to Oracle with the capability to store segmentations. Enter SQL∗Plus with system privileges (*sys* or *oracle* user) and execute the following statements:

```
SQL> CREATE USER newuser IDENTIFIED BY userpassw;
```

This creates a user named *newuser* with password *userpassw*.

```
SQL> GRANT VxUser, UNLIMITED TABLESPACE TO newuser;
```

This command grants to the new user the role *VxUser* (the user receives all the privileges granted to the role). This role contains the next system privileges:

- ALTER SESSION
- CREATE CLUSTER
- CREATE DATABASE LINK
- CREATE INDEXTYPE
- CREATE OPERATOR
- CREATE PROCEDURE
- CREATE SEQUENCE
- CREATE SESSION
- CREATE SYNONYM
- CREATE TABLE
- CREATE TRIGGER
- CREATE TYPE
- CREATE VIEW

The UNLIMITED TABLESPACE privilege is also granted to the user. This privilege allows to use space in all the tablespaces (the default tablespace is SYSTEM). No limit is specified.

To specify a disk quota use the following commands:

```
SQL> REVOKE UNLIMITED TABLESPACE FROM newuser;
SQL> ALTER USER newuser QUOTA 5M ON system;
```

Once the user is created, he has to create the tables to store segmentations: **VxSegmentation_tab** (p. 4), **VxSegment_tab** (p. 5), **VxSegmentStringFeature_tab** (p. 6), **VxSegmentIntFeature_tab** (p. 8) and **VxSegmentDoubleFeature_tab** (p. 8). The sequences VxSegmentation_id and VxSegment_id also need to be created.

You can do it with the `CREATE` statements shown in this documentation or through the makeTables script:

```
SQL> START makeTables.sql
```

3.2 Creating the ”demoweb” user

This appendix shows how to create a user that has read-only access to data on oracle. A user like this can be used for demos published on the web.

Our *demoweb* user will have read-only access to the data of the *demodata* user.

Enter SQL∗Plus with system privileges (*sys* or *oracle* user) and execute the following statements:

```
SQL> CREATE USER demoweb IDENTIFIED BY demoweb;
SQL> GRANT VxUser TO demoweb;
```

The `UNLIMITED TABLESPACE` privilege is not granted, so *demoweb* will have a disk quota of zero. That’s enough because *demoweb* doesn’t add any data to the database.

Next we have to grant *demoweb* with privileges of reading *demodata* tables:

Connect to oracle as *demodata*:

```
SQL> CONNECT demodata@csys
```

Grant read access:

```
SQL> GRANT SELECT ON VxSegmentation_tab TO demoweb;
SQL> GRANT SELECT ON VxSegment_tab TO demoweb;
SQL> GRANT SELECT ON VxSegmentStringFeature_tab TO demoweb;
SQL> GRANT SELECT ON VxSegmentIntFeature_tab TO demoweb;
SQL> GRANT SELECT ON VxSegmentDoubleFeature_tab TO demoweb;
```

Now, *demoweb* has read access on *demodata* tables. But the schema name has to be specified (”demodata” schema):

```
SQL> SELECT * FROM demodata.VxSegmentation_tab;
```

It would be better if we can avoid specifying the ”demodata” schema. We will do it through views:

Connect to oracle as *demoweb*:

```
SQL> CONNECT demoweb@csys
```

Create views to access *demodata* tables transparently:

```
CREATE VIEW VxSegmentation_tab AS SELECT * FROM demodata.VxSegmentation_tab;
CREATE VIEW VxSegment_tab AS SELECT * FROM demodata.VxSegment_tab;
CREATE VIEW VxSegmentStringFeature_tab AS SELECT * FROM demodata.VxSegmentStringFeature_tab;
CREATE VIEW VxSegmentIntFeature_tab AS SELECT * FROM demodata.VxSegmentIntFeature_tab;
CREATE VIEW VxSegmentDoubleFeature_tab AS SELECT * FROM demodata.VxSegmentDoubleFeature_tab;
```

Now, *demoweb* user can just type:

```
SQL> SELECT * FROM VxSegmentation_tab;
```

to access *demodata* tables.

3.3 Some administrative commands

List tables created by the user:

```
SQL> SELECT table_name FROM user_tables;
```

List roles granted to user:

```
SQL> SELECT * FROM session_roles;
```

List privileges granted to user:

```
SQL> SELECT * FROM session_privs;
```

Current username:

```
SQL> SELECT username FROM user_users;
```

List table privileges where the grantee is the current user:

```
SQL> SELECT * FROM user_tab_privs_recd;
```

List privileges granted on tables owned by the current user:

```
SQL> SELECT * FROM user_tab_privs_made;
```

3.3.1 Commands for system user:

List roles granted to user 'newuser':

```
SQL> SELECT * FROM dba_role_privs WHERE grantee='newuser';
```

List system privileges granted to all roles:

```
SQL> SELECT * FROM role_sys_privs;
```

All this information is being retrieved from Data Dictionary Views. user_tables, session_roles, dba_role_privs, etc... are all data dictionary views. You can get a list and description of all data dictionary views through the table `DICTIONARY` and the `DESCRIBE` command.