Uppsala University
Department of Information Technology
Thanh Truong

DATABASE DESIGN II - 1DL400
Assignment 3

Extensible database indexing

1. Introduction

The goal of this exercise is to give a practical experience in database system index structures. The exercise consists of writing Java code to implement the new index structure *KD-tree* and add KD-tree based indexes to Amos II. You will also investigate the scalability of using KD-tree in queries.

1.1. Extensible database indexing

Assume a relation *R* that has a large number of tuples and the SQL query *Q*:

```
select * from R where A < 'constant'
```

Listing 1: A query to find tuples in a given range

Without indexing query Q requires a full scan of the entire table R to select the tuples that match the predicate *A < 'constant'*. Such a full scan is expensive to perform if R has many rows. In order to speed up the search, one can employ a *B-tree* index on attribute A, which will find the matching tuples fast. B-trees provide scalable range queries, i.e. scalable matching of an interval of single values in a query against an indexed set of single values stored in the database.

However, applications often require scalable search based on other criteria than indexed interval search. One example is finding those vectors of numbers (e.g. coordinates) in the database that are close to a given vector of numbers in a query. There are other kinds of indexes than B-trees for this; in the exercise, we are going to use two such index structures

called *KD-trees* and X-trees. You are not implementing the KD-trees themselves, but instead you are provided an already complied Java jar package implementing KD-trees. X-trees are included in the system already.

1.2. Adding and removing indexes on stored functions

Amos II has three kinds of built-in indexes: Linear Hashing, main memory B-trees, and X-trees. A specific kind of index can be defined for a parameter of a stored function by the system function:

```
create function pname(Person p) -> nm as stored;
doc(“create_index”); /* Get documentation */
create_index("pname", "nm", "hash", "unique");
```

Listing 2: How to make unique hash index on stored function

See http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html#indexing for documentation of the function *create_index*. The following illustrates how to index stored Amos functions:

```
/*Define a B-TREE index on the second argument */

create function winequalitysamples(Number sp)-> Vector of Number wq as
stored;
#[OID 1361 "NUMBER.WINEQUALITYSAMPLES->VECTOR-NUMBER"]

/*Define a B-TREE index on the second argument */

/*Examine all defined indexes */

create_index("winequalitysamples", "wq", "MBTREE", "multiple");

indexes(#'winequalitysamples');
{#[OID 1362 "P_
NUMBER.WINEQUALITYSAMPLES->VECTOR-NUMBER"],0,"hash","unique"}
{#[OID 1362 "P_
NUMBER.WINEQUALITYSAMPLES->VECTOR-NUMBER"],1,"mbtree","multiple"}

/*Drop a HASH index*/

drop_index('winequalitysamples', 'sp');
0

drop_index('winequalitysamples', 'wq');
Trying to remove last index on NUMBER.WINEQUALITYSAMPLES->VECTOR-NUMBER

/* There last index cannot be removed. There is one B-tree index left: */
```

```
indexes(#'winequalitysamples');
{#[OID 1362
"P_CHARSTRING.WINEQUALITYSAMPLES->VECTOR-NUMBER"],1,"mbtree","multiple"
}
```

Listing 3: Example of using indexes on a stored function

The system function call *`indexes(#'winequalitysamples');`* returns the indexes defined for the function *`winequalitysamples`*. It first shows that a unique hash index (i.e. a key) by default is added by the system on the first argument *sp* (position 0). The B-tree index is then added on the second argument (position 1). An index can be removed by calling *`drop_index(Charstring functionname, Charstring argname);`*. However, the system needs at least one index defined on a stored function. Therefore, one cannot remove the last remaining index. In the example the index on parameter *sp* is removed, and instead the new index on parameter *wq* is the remaining index.

1.3. Physical algebra operators in Amos II

With the system function `pc(Charstring functionname)` you can inspect the execution plan of an Amos function. The execution plan is a program in an internal language called the *physical Amos algebra*. The physical Amos algebra is a very simple language having the following *execution plan operators*.

In general, if an index type is named *index* (e.g. HASH) there will be the following physical execution plan operators on such an index:

- *index*-FULL-SCAN (e.g. HASH-FULL-SCAN): will iterate over the entire index, i.e. all rows in the tables storing the function. It returns all tuples in the stored function where one input parameter is indexed by this index. If there are many objects in the index this may result in many iterations and may not scale. Full scan should therefore be avoided for large tables.

- *index*-INDEX-SCAN (e.g. HASH-INDEX-SCAN): will find the tuples in a stored function matching an index key for a 'multiple' index. This will scale if there are not very many such matches, that is if the index is selective. It is up to de DBA not to make indexes with too high multiplicity.

- *index*-INDEX-GET (e.g. HASH-INDEX-GET): will find the single tuple matching the key directly. This scales if the index implementation is sound.

In addition there is also a general execution plan operator to call the implementation of a *foreign function* implemented in some external programming language (C, Lisp, Java):

- CALL: calls a foreign function, e.g. to do special index search.

All physical algebra operators take some bound variables as input and iteratively emit a stream of bindings of unbound variables. In the execution plans printed by *pc()* bound variables are marked '-' and unbound variables '+'.

1.4. Foreign functions

Amos II users can define *foreign functions* implemented in a conventional programming language such as Java, C, or Lisp. For example, one might want to use C to manage index structures efficiently and Java to visualize query results. Foreign functions allow Amos II to access specialized storage managers, main memory data structures, computational engines, and so on. They provide the basic primitives to extend the DBMS.

Foreign function are documented in section 6.1 ''*Foreign and multi-directional functions"* in [1]. There is more documentation of Amos II Java interfaces in sections 3.1 and 3.2 of [5]. The following example is a skeleton of the implementation of a foreign function in Java:

```
/*Import Amos II Java interface packages*/

import callin.*;

import callout.*;

public class KDTreeIndex{
. . .

  /* PUT puts <key, val> into a KDTREE given its Id. The first PUT always
  constructs the KDTREE */

  public void jkdtree_put(CallContext cxt, Tuple tpl) throws
  AmosException, KeySizeException, KeyDuplicateException {

  /*Get the id*/

  int id = tpl.getIntElem(0)

  /* convert from sequence of numbers to array of floats*/

  double [] key = toArray(tpl.getSeqElem(1));

  Oid val = tpl.getOidElem(2);

  . . .

  /*Set the result*/

  tpl.setElem(3, val);
```

```
  cxt.emit(tpl);

  . . .

  }

}
```

Listing 4: Implementation of the foreign function *kdtree_put*

After a foreign function has been implemented in Java as in Listing 4, the signature of that foreign function is defined in AmosQL and mapped to the Java implementation as in Listing 5.

```
create function kdtree_put(Integer kdId, Object f, Object o)-> Object
as foreign 'JAVA:KDTreeIndex/jkdtree_put';
```

Listing 5 Definition of foreign function signature of *kdtree_put* in AmosQL

A foreign function implementation in Java always has two arguments *cxt* and *tpl.* The argument *cxt* is an instance of class *CallContext*. It is a data structure managed by Amos II to pass information about the caller. The argument *tpl* is an instance of class *Tuple* and represents bindings of the argument and result parameters of the foreign function call. The order of parameters in *tpl* is the same as in the AmosQL function signature.

Knowing the data type and position of a bound parameter in a foreign function allows us to access tuple elements by using some Java interface functions. For example, the first input parameter *Id* is accessed using *tpl.getIntElem(0)* since its position is 0 and its type is *Integer*. The implementation in Listing 4 binds the result value in position 3 by calling *tpl.setElem* and then emits the result bindings *tpl* by *cxt.emit(tpl).*

The following are the methods of Amos II Java interfaces used in the exercise. The complete documentation of foreign functions in Java is in [5].

- *int Tuple.getIntElem(int pos)* fetches an integer element from position *pos* of a tuple.
- *void Tuple.setElem(int pos, int i)* sets the tuple element at position *pos* to integer *i*.
- *Oid Tuple.getOidElem(int pos)* fetches a arbitrary *Amos object o,* representing any kind of data stored in the database, from a tuple.
- *void Tuple.setElem(int pos, Oid o)* stores Amos object *o* in element *pos* of a tuple.
- *void CallContext.emit(Tuple)* emits a result tuple from a foreign function. *emit* is called iteratively when a bag of values is returned.
- *Tuple Tuple.getSeqElem(int pos)* fetches an object of database type *Vector* from position *pos* of a tuple. A vector is represented in Java as a tuple object.
- *void Tuple.setElem(int pos, Tuple tpl)* stores the tuple *tpl* as a vector in position *pos* of a tuple.

1.5. **Mexima**

Meta External Index Manager (Mexima), a subcomponent of Amos II, provides a generic mechanism to extend the system with new index structures. It allows the index manager in Amos II to call an *external index manager (Exima)* implemented as a stand-alone program in C or Java. An external index manager handles one specific kind of index structure. To connect an external index manager to the Amos II kernel, one needs to implement a few foreign functions in, e.g., Java. These foreign functions reside in a program called a *Mexima Driver*.

For example, the library code for assignment 3, *kd.jar*, contains a Java package that provides an API to interface the *KD-tree* index structure. The main program *KDTreeIndex.java* contains skeletons of the foreign function implementations required to access the *KD-tree* package.

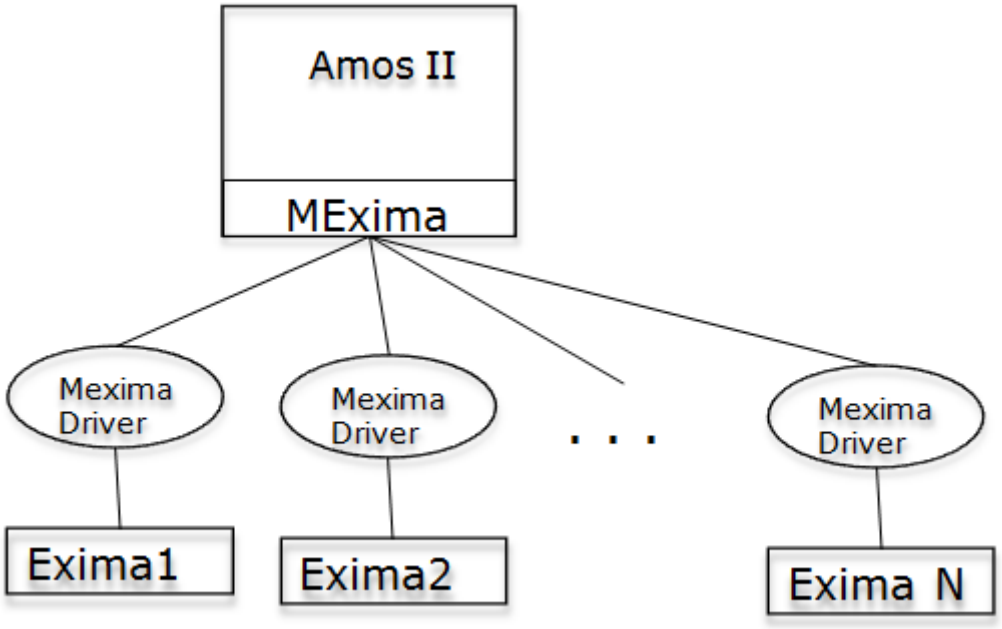

Figure 1 Mexima architecture

A Mexima driver implements and maintains an identifer per index, `xid`. The driver is responsible for generating a new unique index identifier *xid*, as an integer when Mexima requests to make a new index managed by the the driver. Consequently, the Mexima internally maintains a list of `xid` identifiers.

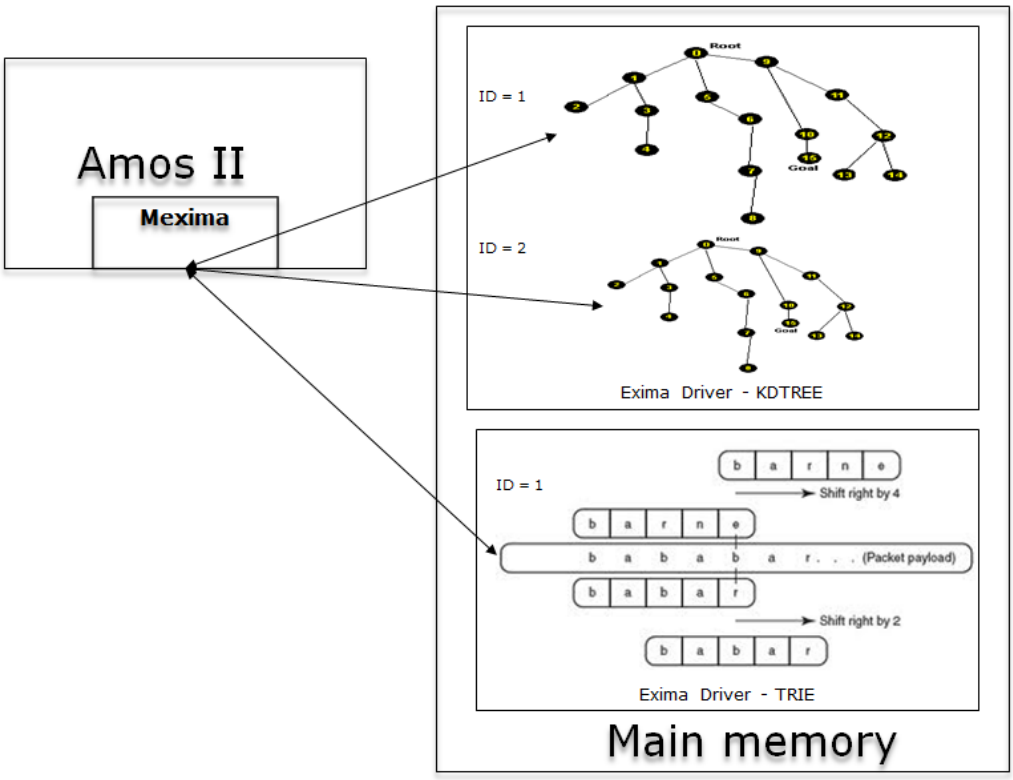

Figure 2 Mexima connects Exima(s) to Amos II

The developer defined a new Mexima driver implementing some foreign functions. First, the new index type is registered to Mexima using the built-in function:

```
register_exindextype(Charstring indextype, Charstring langpath,
Boolean manualsaver) -> Boolean;
```

Listing 6: Definition of a new index type

Input:

- `indextype` : Name of new index type
- `langpath` : In this assignment, as always the string ``FOREIGN''
- `manualsaver`: In the assignment always specified as *false*.The user has the option to write special programs to save or restore the external index managed by a Mexima driver by setting the parameter *`manualsaver`* to true. If *`manualsaver`* is *false* Mexima will automatically provide saving and restoring the index.

Output: Success (true) or no result.

The next step is to redefine a set of foreign functions to update and search the new index structure named *`indextype`*, with signatures:

- *`create function indextype_make() ->Integer as foreign...`*
  creates a new index of the given index type and returns its index identifier.

- *`create function indextype_put(Integer id, Object key, Object val) -> Object as foreign ...`*
  stores a value *`val`* associated with *key* into an index structure whose identifier is *`id.`* The value is stored as an Amos object (class *`Oid`*).

- *`create function indextype_get(Integer iid, Object key) -> Object val as foreign...`*
  retrieves the value *`val`* associated with *`key`* from an index structure whose identifier is *`id`*.

- *`create function indextype_delete(Integer id, Object key) -> Object as foreign...`*
  deletes a value *`val`* associated with *`key`* from an index structure whose identifier is *`iid`*.

- *`create function indextype_clear(Integer id) -> Object as foreign...`*
  removes an entire index whose identifier is *`id`*.

- *`create function indextype_save(Charstring filename) -> Boolean as foreign...`*
  This function is optional and not used in the assignment. It is required only if the parameter *`manualsaver`* of function *`register_exindextype`* is set to *true*. It saves all index data named *`indextype`* on disk.

- *`create function indextype_load(Charstring filename) -> Boolean as foreign...`*
  This function is optional and not used in the assignment. It is required only if the parameter *`manualsaver`* of function *`register_exindextype`* is set to *true*. It restores all index data of named *indextype* from disk.

- *`create function indextype_mapper(Integer id) -> Bag of Object as foreign...`*
  This function is optional and not used in the assignment. It iterates over the index and emits pairs of keys and values. This function is optional as not all index structures allow iteration over all its data items.

Mexima is transactional which means that it is possible to commit and rollback index updates.

1.6. K-dimensional Tree

A K-dimensional search tree (KD-tree) is a main memory data structure that extends binary trees to index multidimensional data. In this assignment, a KD-tree implementation is delivered as a Java package, *kd.jar*, which has API documentation available online at [3]. You need to go through that API to understand what functions to invoke to make a KD-tree index.

2. Preparation

Download skeleton code for assignment 3 from
http://www.it.uu.se/edu/course/homepage/dbastekn2/vt11/dbt2-vt2011-assignments.html
and unzip it into your home directory. There is a *readme.txt* file containing instructions on how to set up your environment variables and run the skeleton code.

For background reading, you are recommended:

- Chapters 20, 21 and 22 in [2].
- Chapter 16 in [4].
- All material from the lectures of this course. See also the assignment web page of this course for additional information.
- Section 7.1, 6.1 in [1].
- Section 3.1, and 3.2 in [5].

There is also a supervised introduction to the assignment.

3. Assignment

You will modify file *lab3_stub.osql* and file *KDTreeIndex_Stub.java* to complete the assignment.

In this assignment, you will work with the database *WineSample*. Each *WineSample* has an identifier number stored in the function *'wsId',* and a feature vector of numbers stored in function *'features'*.

```
create type WineSample;

/* Each WineSample has an identifier number stored in the function
'wsId' */

create function wsId(WineSample ws) -> Number id as stored;

/* Each WineSample has a feature vector of numbers  stored in the
function 'features': */

create function features(WineSample ws) -> Vector of Number f as
stored;
```

The assignment consists of several parts:

3.1. Exercise 1

a) Complete function *getSample* to get a wine sample for a given *wsID*. Then you should run the test some times to see the execution time.

```
/* Make derived function to get the wine sample for a given wsId */
create function getSample(Number wsId) -> Winesample ws
  /**TODO: Your code here*/;

/* Run getSample three times to see execution time: */
getSample(37);
getSample(37);
getSample(37);
```

b) Look at the execution plan of function *getSample*
Does it scale? Explain why.

```
pc("getSample");
/**TODO Give your comments here*/
```

Make hash index on *wsId* to speed up *getSample*. Does it scale? Why?

```
/* Make hash index on WSID to speed up getSample: */
/**TODO: Your code here*/
/* Reoptimize getSample: */
recompile("getSample");

/* Run getSample three times to see execution time: */
getSample(37);
getSample(37);
getSample(37);
/* Inspect execution plan of getSample: */
pc("getSample");
/**TODO How will it scale? Why? */
```

3.2. Exercise 2

a) Make a function to find all wine samples whose feature vectors are within a given distance from a wine sample *ws*.

```
create function closeWineSamples(WineSample ws, Number distance)
                                  -> Bag of WineSample
 /* Find wine samples within distance from a give wine sample ws */
/**TODO : Your code here*/;

set :ws = getSample(37);
/* Run it three times to investigate speed */
closeWineSamples(:ws, 3);
```

Instruction: You should use the function *Euclid,* which returns a distance between two vectors of numbers.

b) Look at the execution plan of function *closeWineSamples*
Does it scale? Why? You should give your comments on the speed of the execution.

```
pc("closeWineSamples");
/**TODO: How will it scale? Why?*/
/**TODO: Give your comment on the speed*/
```

3.3. Exercise 3

a) Implement a foreign function in Java to speed up proximity search using the unknown KD-tree package in Java found on the web.

kdtree_make creates a new index type KD-tree. The function *kdtree_make* returns a unique id of the new KD-tree. It signature follows

```
create function kdtree_make() -> Integer id as foreign
'JAVA:KDTreeIndex/kdtree_make';
```

Instruction: Complete the implementation in *KDTreeIndex_Stub.java/kdtree_make*

b) Call *kdtree_make* to create new KD-tree index and store it in the function *'winesampleIndex'*

```
create function winesampleIndex()-> Integer as stored;
```

```
/**TODO: Your code here*/
```

c) Implement a foreign function *kdtree_put* to insert into a KD-tree a feature vector *f* for a given object *o*. Its signature follows:

```
create function kdtree_put(Integer id, Object fo, Object o) ->
Object as foreign 'JAVA:KDTreeIndex/kdtree_put';
```

Instruction: Complete the implementation in *KDTreeIndex_Stub.java/kdtree_put*

d) Define a procedural function to add a wine sample *ws* and its feature vector *fv* to KD-tree *wineSampleIndex* using *kdtree_put*.

```
create function addWineSampleIndex(WineSample ws, Vector of Number
fv) -> WineSample
 /**TODO: Your code here*/;
/** go through the database and add all wine samples to the
wineSampleIndex: */
for each Winesample ws
    addWineSampleIndex(ws, features(ws));
```

3.4. Exercise 4

a) Implement a foreign function *kdtreeProximitySearch* that calls a KD-tree whose identifier is *id* to search all feature vectors within a distance *dist* from a given feature vector *f*. Its signature is:

```
create function kdtreeProximitySearch(Integer id, Vector of Number
f, Number distance)-> Bag of WineSample
 as foreign 'JAVA:KDTreeIndex/kdtreeProximitySearch';
```

b) Define a *proximity search* for features of wine samples. It calls the foreign function *kdtreeProximitySearch*. It uses the KD-tree index to return a bag of wine samples whose features are within a distance *dist* from the features of *ws*.

```
create function closeWineSamples2(WineSample ws, Number dist)
                                -> Bag of WineSample
  /**TODO: Your code here*/;
```

```
/* What are the wine samples within distance 3 from: ws? */
closeWineSamples2(:ws, 3);
```

3.5. Exercise 5:

Investigate the speed and execution plans of *closeWineSamples* and *closeWineSamples2.* What is the difference in speed between the two functions? How do the execution plans differ? Explain difference in speed!

```
/*
Inspect execution plans of closeWinesample and closWinesamples2 */
closeWineSamples(:ws, 3);
closeWineSamples(:ws, 3);
closeWineSamples(:ws, 3);
closeWineSamples2(:ws, 3);
closeWineSamples2(:ws, 3);
closeWineSamples2(:ws, 3);
pc("closeWinesamples");
pc("closeWinesamples2");
/**TODO: How do they scale? Why?*/
```

3.6. Exercise 6:

a) Register a new index type *'KDTREE'* for transparent proximity indexing

```
/* Register a new index type KDTREE for transaparent proximity
indexing*/
/**TODO: Your code here*/
```

b) Implement the following functions to manipulate the KD-tree index. Their signature follows

```
create function kdtree_make() -> Integer id
  as foreign 'JAVA:KDTreeIndex/kdtree_make';


create function kdtree_put(Integer id, Object fo, Object o) ->
Object as foreign 'JAVA:KDTreeIndex/kdtree_put';
```

```
create function kdtree_delete(Integer id, Object f) -> Boolean
  as foreign 'JAVA:KDTreeIndex/kdtree_delete';

create function kdtree_get(Integer id, Object o) ->Bag of Object
  as foreign 'JAVA:KDTreeIndex/kdtree_get';

create function kdtree_clear(Integer id)-> Boolean
  as foreign 'JAVA:KDTreeIndex/kdtree_clear';
```

c) Index the parameter '*f*' of the stored function *'features'* using KDTREE index using built-in function *create_index*.

```
/*  Index the parameter 'f' of the stored function 'features' using
KDTREE index.
*/
/**TODO: Your code here*/
```

d) Investigate the execution plan of *closeWineSamples.*

```
recompile("closeWineSamples");
/* Now, you should see that the execution plan of 'closeWinesamples'
is transformed to utilize KD-tree proximity search. It is done
transparently since we have declared a new index type of KD-tree on
'features'*/
pc("closeWinesamples");
```

3.7. Exercise 7:

a) Replace index on the parameter '*f*' of the table *'features'* using *KDTREE* index to use the index kind *XTREE* instead. You should use function *create_index.*

b) Investigate the execution plan of *closeWinesamples* again. Run it to investigate the speed. Compare with the KD-tree index. **Discuss result!**

```
recompile("closeWineSamples");
pc("closeWinesamples");
/**TODO Your comments here*/
```

```
/* Run closeWineSamples three times to investigate speed: */
closeWineSamples(:ws, 3);
closeWineSamples(:ws, 3);
closeWineSamples(:ws, 3);
/**TODO Your comments here*/
```

3.8. Exercise 8:

a) Make function to find wine samples given features *fv*

```
create function featuredWineSamples(Vector of Number fv)->Bag of
Winesample
/**TODO Your code here*/;
```

b) Investigate the execution plan of *featuredWineSamples* and give your comments on scalability.

```
and set :fv = {7,0.31,0.26,7.4,0.069,28,160,0.9954,3.13,0.46,9.8};
/* Run it three times */
featuredWineSamples(:fv);
featuredWineSamples(:fv);
featuredWineSamples(:fv);

pc("featuredWineSamples");
/**TODO: Your comments on execution plan and scalability*/
```

4. Examination

4.1. Hand-in documents

Each exercise requires extending AmosQL code in file *lab3_stub.osql* and Java code in *KDTreeIndex_Stub.java*.

Adding descriptive comments is advised, in case your solution does not pass a validation script we use. Each group sends only one compressed folder (tar or zip) containing:

- lab3_stub.osql
- KDTreeIndex_Stub.java
- group.txt stating your group number, names and emails

4.2. Validation

- We test your solution with a validation script.
- Then we check your answers.
- Finally, we manually investigate the AmosQL and Java code.

Failure on a step means a K is given. You then have to re-submit your solution until it passes that step. G is given when everything is correct.

Bibliography

1. S.Flodin, M.Hansson, V.Josifovski, T.Katchaounov, T.Risch, M.Sköld, and E.Zeitler: *Amos II Release 13 Version 1*, Feb 20, 2011, http://user.it.uu.se/ udbl/amos/doc/amos_users_guide.html (accessed Feb 20, 2011)

2. R. Elmasri and S.B.Navathe: *Fundamentals of Databases, 6th edition.* Addison-Wesley, 2010.

3. Simon D. Levy: *KD-Tree Implementation in Java and C#.* 2011. http://home.wlu.edu/ levys/software/kd/

4. T.Padron-McCarthy, and T.Risch: *Databasteknik.* 2005.

5. D. Elin and T.Risch: *Amos II Java Interfaces*. http://user.it.uu.se/~torer/publ/javaapi.pdf Technical report, 2000.
