IT 10 017
Examensarbete 30 hp
May 2010

Indexing nearest neighbor queries

Thanh Truong

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Indexing nearest neighbor queries

Thanh Truong

In database technology, one very well known problem is K nearest neighbor (KNN). However, the cost of finding a solution of the KNN problem may be expensive with the increase of database size. In order to achieve efficient data mining of large amounts of data, it is important to index high dimensional data to support KNN search.
Xtree, an index structure for high dimensional data, was investigated and then integrated into Amos II, an extensible functional Database Management System (DBMS).The result of the integration is AmosXtree, which has showed that the query time for KNN search on high dimensional data, is scale well with both database size and dimensionality.
To utilize the functionality of AmosXtree, an example is given on how to define an index structure in searching pictures.

Handledare: Tore Risch
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 017
Tryckt av: Reprocentralen ITC

TABLE OF CONTENTS

Acknowledgements

I am gracefully thankful to my supervisor, Tore Risch, who did give me encouragement, guidance and valuable comments from the initial to the final state of the Thesis work. I have learned not only the subject itself but also more knowledge about Database technology.

I would like to take this opportunity to offer my regards and blessings to my parents for all of theirs support and love for me.

Lastly, many thank to my friends: Guillermo, João, Seif, Xiaoyu, Quý and Thảo who supported me in any the respect during the Master program and the completion of the Thesis.

Thành Trương

1. Introduction

There is a growing need for searching and mining multi dimensional data in large data sets. For example, Global Positioning System (GPS) has become an anchor of transportation systems, which provides reliable positioning, navigation for aviation, ground, and maritime operations. GPS also supplies locating and timing services, which are widely used in rescue missions. In such applications, geographic (multi dimensional) data are stored and queried over. Another example is mining gene data expression, which aims at classifying, and analyzing gene data in order to predict disease and to find a cure for illness. The study is increasingly attractive to bio-informatics researchers [5]. The mining gene data is one aspect of a broad field known as data mining.

For mining multi dimensional data, a very well known problem is locating data objects in a data space closest to the given query data object. Such problem is called a *K nearest neighbor problem (KNN)*. Given a high dimensional data space and a query data object, it requires to find K nearest data objects in relation to the query one [3] .

However, with the increase of database size, the cost of finding a solution of the KNN problem may be expensive. Technically, it might result in scanning through the whole database one or several times, which becomes a scalability problem if the database is large. In order to achieve efficient data mining of large amounts of data it is therefore important to index high dimensional data points to support KNN searches.

In order to tackle the barrier, this Thesis aims at studying state of art index techniques on high dimensional data indexing and then integrating one such index technique into an extensible Database Management System (DBMS), Amos II [14]. Amos II is an extensible functional DBMS where data is represented as functions over which object-oriented and functional queries can be specified using the query language *AmosQL*. In Amos II, multi dimensional data can be represented by a datatype *Vector* and the KNN problem can be easily expressed by AmosQL queries of vectors. Moreover, Amos II has interfaces to external programs written in programming languages such as ANSII C, Lisp, and Java [18]. This ability of Amos II is used in this Thesis to extend the system with an index structure for high dimensional data. The chosen index structure is the X+-tree [19] and is implemented in C. Compared to the X-tree [15], the X+-tree aims to have better performance because it tries to avoid regenerating the tree by limiting the maximum number of children nodes of a non-leaf node.

After extending Amos II with the X-tree index structure, some experiments were conducted to empirically show that query time for KNN searches on high dimensional data with such an index on data scales well when both the dimension and the size of the database are increasing.

2. Background.

This chapter first discusses the conventional relational database model with a focus on data representation and indexing. Then there is an overview of what multi-dimensional data is all about. The next subject to be discussed is the scalability problem when the number of dimensions increases. Then the applicability of multi-dimensional indexing on a common data mining problem is discussed: similarity search and K nearest neighbor search.

The chapter is ended with an overview Amos II [14] the functional and objected oriented DBMS used in this Thesis for implementation and experimentation.

2.1 Database

The relational database model invented by Codd in 1970 [2] is the most common database model used today. It offers a mechanism for storing, manipulating, and retrieving information organized as a collection of *tables*.

Each table contains a number of *records* representing rows. A record has a number of *attributes* to represent column values. The attributes (columns) are identified by names and associated with a *domain* specifying of the data representation of attribute values.

The attributes constituting the *primary key* of a table is used to uniquely identify a row in the table. Tables are linked by a set of common key values also known as *foreign keys*. A *foreign key* is a reference from a tuple attribute to a key in another table inside the same database. Keys are fundamental for expressing relationship between rows in different tables.

In order to retrieve data, *queries* specify how information is searched from a relational database. Relational queries are expressed using the query language Structured Query Language (SQL). SQL is used not only for searching data but also to create tables, index data, or to insert, update, and delete data, i.e. SQL is a general language for interacting with a relational DBMS.

To speed up database queries and updates requires a data structure defined as a database *index* that enables the DBMS to find the particular data rows specified in a query or an update quickly. A database index is similar to the index of book, which is a reference to a topic associated with page. Instead of scanning the book (until the last page in the worst case), the book reader can use an index to find a specific page to read. Analogously

a DBMS finds the rows specified by a query by using database indexes.

The most common implementations of indexes in relational databases are B-trees [13] or dynamic hash tables [9]. Without index, queries require expensive linear table scan proportional to the table size. With index, instead of scanning (in the worst case) the whole table, particular rows matching the query can be found by traversing an index tree (or hash table) down to specific rows referenced by the index. The index search scales much better than linear scanning all data.

2.2 High dimensional data

Dimension

Mathematically, the dimension of a data space or object is defined as the number of coordinates needed to specify a point in the space.

For example, a point P in one dimension data space (1-D) can be represented by one coordinate x : P (x), In two dimension data space, the point P is specified by two coordinates x and y : P(x,y).

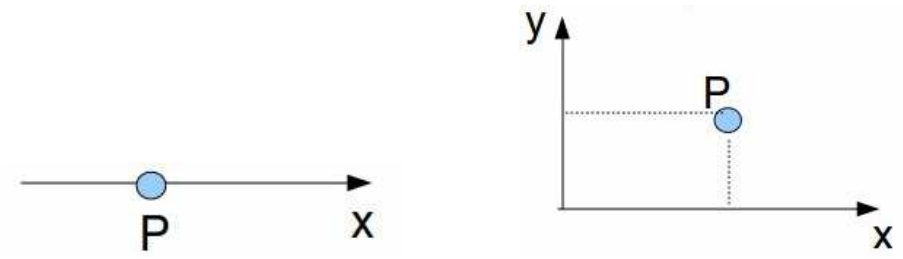

Figure 1 A point P in one dimension and two dimension data space respectively

Preliminaries

Given data space S where d is the number of dimension of S, n is the number of data objects within the space, a data object x_i *in S can be defined as following:*

$$S = \{x_i \mid 1 \le i \le n\}; x_i = [x_{i1}, x_{i2}, ..., x_{xd}]$$

In data mining, the dimension of a data set is defined as the number of attributes, which all data point in that data set must have [12]. For example, the dimension of spatial data is three implying that a physical object in spatial data can be represented with three coordinates (also known as Cartesian or 3-D system). In other words, an object in a 3-D

system can be specified by x, y, and z values. Extending that, by adding one attribute (dimension) such as time to the 3-D space to form 4-D space the new data space is able to describe the movement of physical objects.

Many data sets are of dimensionality larger than one and the understanding the nature of such high dimensional data is important to solve data mining problems [10].

2.2.1 Similarity search

❖ Similarity and Dissimilarity

Similarity is a quantity that expresses the strength of a relationship between two objects. In contrast, dissimilarity measures the difference between the two objects [6]. The difference or dissimilarity can be measured by distance function for example: Euclidean, Minkowski, Maxnorm.

❖ Relationship between similarity and dissimilarity

Let denote SI_{ij} , DI_{ij} *as the similarity and dissimilarity between two objects i and j respectively. The similarity and dissimilarity are normalized so that their values fall in between 0 and 1.*

$$SI_{ij} + DI_{ij} = 1$$

From definition 2, it is natural to use the distance function (measuring dissimilarity) implicitly such as Euclidean to evaluate the similarity too.

In order to apply the distance function or measurement on complex data objects such as images [4][17], sequences, video, and shapes, it is quite common to approximate these data with high dimensional vectors describing properties of the objects, or also known as *feature vectors.* Color intensity, sharpness, and face geometry are examples of feature vectors. A numerical *feature vector* of *n* dimensions is the result of transforming selected properties of a complex object. By doing the transformation, it makes the processing and statistical analysis on these data easier [4]. The reason is that the feature vector is supposed to approximate some properties to be searched for. For example, one may want to allow efficient search in a database for persons whose face is similar to a given face. What one does is to use some relatively expensive face recognition algorithm to compute the face geometry in a picture as a feature vector. Then one stores the image together with

the feature vector in the database and index the feature vector using a high dimensional indexing structure. To find a face in the database similar to a give one first finds the faces with similar feature vectors and then applies a more careful face comparison algorithm on the found objects. This will scale much better than having to scan the entire database to apply the face comparison algorithm.

In general, the idea of similarity search is to search in the given data space to find data objects that are similar to a given object. Given a data space *S* of *n* objects represented by feature vector *v* with *d* is the dimension, and a query point *p*, the similarity search is expected to find the points closest to *p*. The similarity is measured by a distance function D calculated as the distance between the given data point and a point in S. For example, the value of the distance function D between two data objects is *dist* means that they are *dist* similar with each other. *Range search* is defined as a search that returns points in the given range - the given distance from the query point. *Range search* is sometimes ambiguously used with the term similarity search. It is defined as follows:

❖ **Range search - [20]**

Given a data space S, a search that returns all data objects, which are "dist" similar to the given search object, is defined as similarity search. The returned data make up S'*:*

$$S' = \{\forall x_i : D(p, x_i) \leq dist\}$$

In other words, range search finds all points within distance *dist* from the query point *p*.

From the general concept of similarity search, we have another case in which we want to find the K (K > 1) data points closest to the given query point p. It is also known as K Nearest Neighbor Search problem (K-NNS) defined as follows:

❖ **K nearest Neighbor Search (K - NNS)**

K nearest neighbors search is defined as: given a set of S of n data objects and a query point p, find a subset $S' \subseteq S$ *of* $k \leq n$ *data objects such that for any objects* $x_i \in S$ *and* $h \in S - S'$, $D(p, x_i) \leq D(p, h)$.

S' is the data space making up from the K data objects that are nearest to the given query point p.

$$S' = \{x_i \mid \forall h \in S - S' : D(p, x_i) \leq D(p, h)\}$$

Sometimes K nearest neighbor search is given together with a classifier that tries to classify the given data point (or a set of unknown classes) by major voting among its K nearest neighbors [12]. This is called *K nearest neighbor classification* in which the given object is assigned to the most common class among its neighbors. Intuitively, all neighbors equally contribute to determine the class of the query data point. On the other hand, the result might be influenced by votes of distant neighbors rather than the nearer neighbors. This problem can be limited by weighting the contributions of the neighbors, so that the contribution of the nearer neighbors is much more than one from the far neighbors. This Thesis does not include such classical classification or the weighted distance approach. Instead, it is able to query for K nearest neighbors only.

❖ **Characteristics of KNN search**

It is common to use the nearest neighbor search to classify a given data point based on the major voting of its nearest neighbors. However, one may argue on choosing an optimal number of neighbors. Which number of neighbors provides enough certainty about the class of the given object?

One drawback of the nearest neighbor search is that the search result does not reflect any knowledge of the data space, which could be used to save time in the next search. In other words, the search method does not build up any model, which gives valuable information about the data space.

2.3.2 K nearest neighbor algorithm

The very naive algorithm of K nearest neighbor search is based on first calculating distances from the given point *p* to all other points. The second step are sorting the distance and determining the nearest neighbors based on K-th minimum distance.

Here are steps on how to compute K-nearest neighbors KNN algorithm:

Determine parameter K = number of nearest neighbors

For *each* $x_i \in S$ ***do***

    Calculate $dist_{x_i,p} = D(xi,p)$

End for

Sort the data objects based on theirs distance to p

```
Return K nearest neighbors based on K-th minimum distance.
```

A faster alternative is to use a list of nearest neighbors, which is accumulated and updated when a possible nearest neighbor is found. This algorithm does neither store all the distances nor sort them.

```
Start with Minimum Distance MINDIST =∞
An empty queue size of K
For each x_i ∈ S do
    Calculate dist_{x_i,p} = D(xi,p)
    If dist_{x_i,p} < MINDIST then
        Push x_i to the queue
        Update MINDIST = dist_{x_i,p}
    End if
End for
Return the queue
```

2.3.3 When is the nearest neighbor meaningless?

It is normal to ask about the nearest neighbor like "the nearest restaurant in town?" or "the most similar picture?". However, a meaningful answer does not always exist and in some cases when the answer of such a query is meaningless. The answer is said to be meaningless if the distances to the nearest neighbors and distances to other objects are very close in resemblance [8].

Let $S^{'}$ denote a subset of *S* in which the nearest neighbor of *p* is located. $S^{'}$ can be formalized as :

$$S^{'} = \{x_i : D(p,x_i) \leq (1+\varepsilon) * D_{min}\}$$

With $D_{\min}$ is the distance between *p* and it's the nearest neighbor, and ε is the enlarged factor. The Figure 2 shows the nearest neighbor data space enlarged by a given ε.

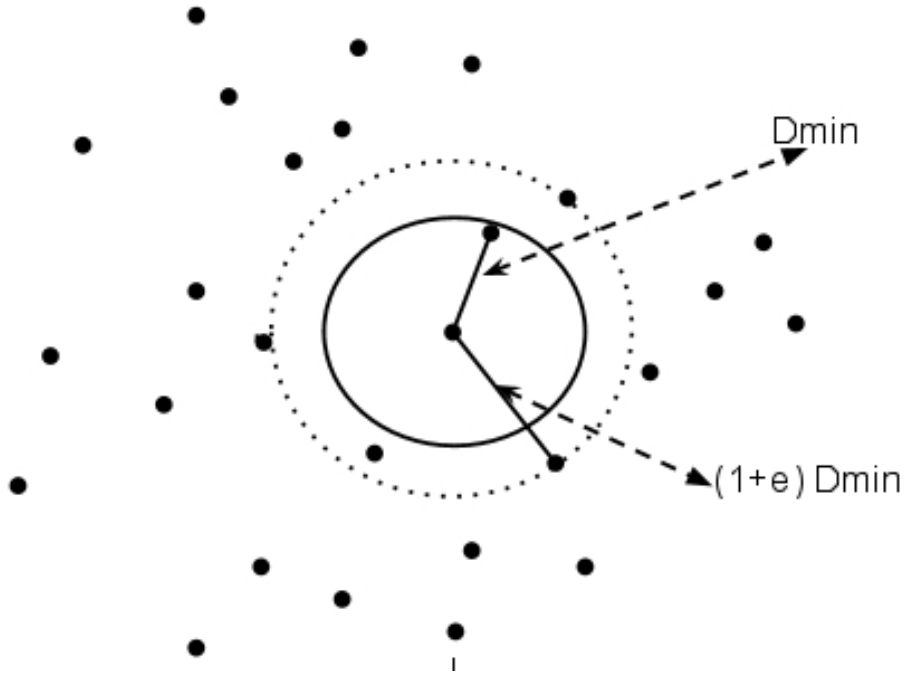

Figure 2: The nearest neighbor's data space enlarged by ε factor

If too many data objects are locate inside $S^{'}$, it means that almost all data points are close to the query point even when ε is small. Hence, it is said that the nearest neighbor search is meaningless. The search is meaningless if the query retrieves the nearest neighbors from the given query point while many others near neighbors are not far away from the nearest one.

In [8] the author introduced the *instability* of the nearest neighbor search. They stated that the nearest neighbor is unstable for a given ε if the distance from the query point to the most data points is less than $(1+\varepsilon)$ times the distance from the query point to its nearest neighbor.

2.3 Index structure for high dimensional data

2.3.1 R-tree - the foundation

As extension of the classic B-tree index method, the R-tree and its variations are used for indexing multi dimensional data. The original R-tree, R+-tree, R*-tree, etc are for low dimensional data, which the TV-tree, X-tree, etc for high dimensional data [15].

An R-tree works based on the concept of the *Minimum Bounding Rectangle (MBR)* of an object, which is the bounding determined by the upper bound and lower bound in each dimension of the object. For example, in 2-D, the MBR of the trapezoid ABCD (see Figure 3) is in form of $(x_{low}, y_{low}, x_{high}, y_{high})$ or precisely (x_A, y_A, x_D, y_C) representing for the lower-left and the upper-right corner of the rectangle.

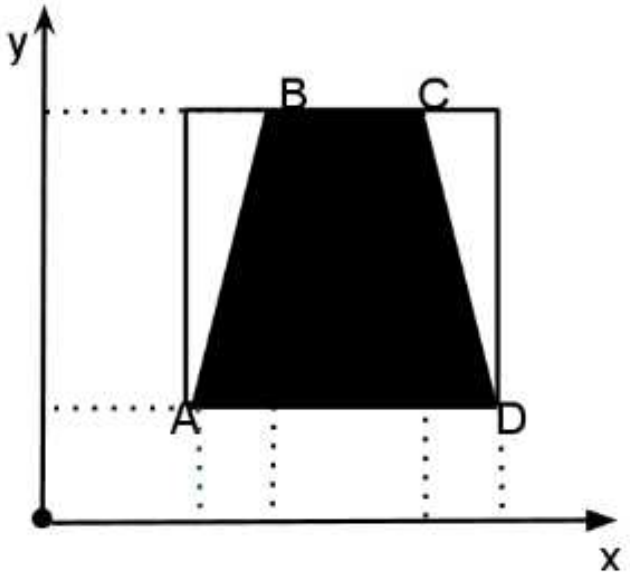

Figure 3 Trapezoid's MBR

The idea of the R-tree is to partition the high dimensional data space into a hierarchical data structure representing MBRs that are indexed instead of it inner data objects themselves. The R-tree can be seen as a hierarchical tree of MBRs.

The R-tree is similar to the B-tree with the ability to remain balanced but the contents of its nodes are different from the B-tree. The R-tree's leaf nodes contain entries, which are in the form of (*MBR*, *object pointer*) where *object pointer* is a pointer to a data object and *MBR* is bounding the corresponding object.

Besides, there are non-leaf nodes (directory nodes) which contain entries of the form

(*MBR*, *p*) where *p* is a pointer to a descent node in the next level of the tree.

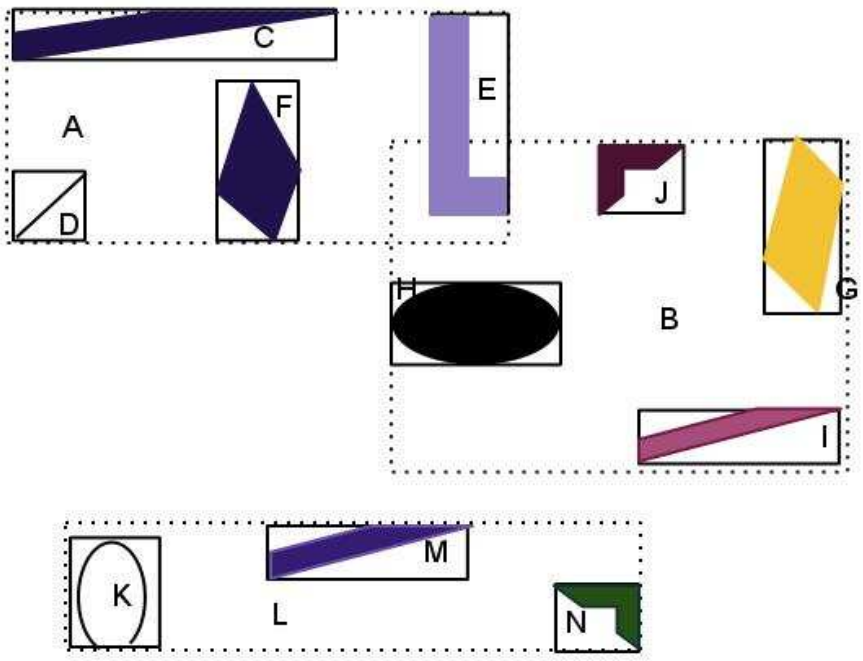

Figure 4: A collection of rectangles

Figure 4: A collection of rectangles depicts a collection of MBRs of objects. The corresponding tree is shown in Figure 5.

Intuitively, there is often the case that some data objects in the data space are covered by more than one bounding rectangle. For example in Figure 4: A collection of rectangles, the L-shape E is in two rectangles A and B. This phenomenon is called *overlap*, which is a result of the splitting phase during the insertion into an R-tree. When inserting a new data object into an R-tree, a leaf node is chosen that has the smallest MBR containing the MBR of the data object. If the leaf node is not full then an entry (*MBR, object pointer*) is inserted. Otherwise, the leaf node needs to be split. The split then might cause overlap between two node's MBRs.

Experiments observed that the overlap in the directory nodes (meaning the rectangles) is increasing with the growth of dimensionality of data [15]. Unfortunately, the degree of overlap influences on the performance of the search since it might have more than one path to traverse down the tree. The cost increases if the search has to follow several

paths. The R-tree therefore is efficient on low dimensional data but higher ones (where overlaps occur more often). The X-tree tries to avoid the overlap between MBRs of nodes to improve performance for higher dimensionality in order to support efficient processing of queries of high dimensionality [15].

2.3.2 X-tree.

Inspired from the original R-tree, the X-tree also has data nodes and normal directory nodes. A data node contains minimum bounding rectangles (MBRs) together with pointers to the actual data objects: (*MBR, p*) while the directory node consists of MBRs together with pointers to sub-MBRs. In addition, the X-tree introduces another type of nodes: *super-nodes*. A *super-node* is large directory node of variable size (a multiple of the usual block size). *Figure 7* shows an example of an X-tree structure with three kinds of nodes: directory node, leaf node, and super node.

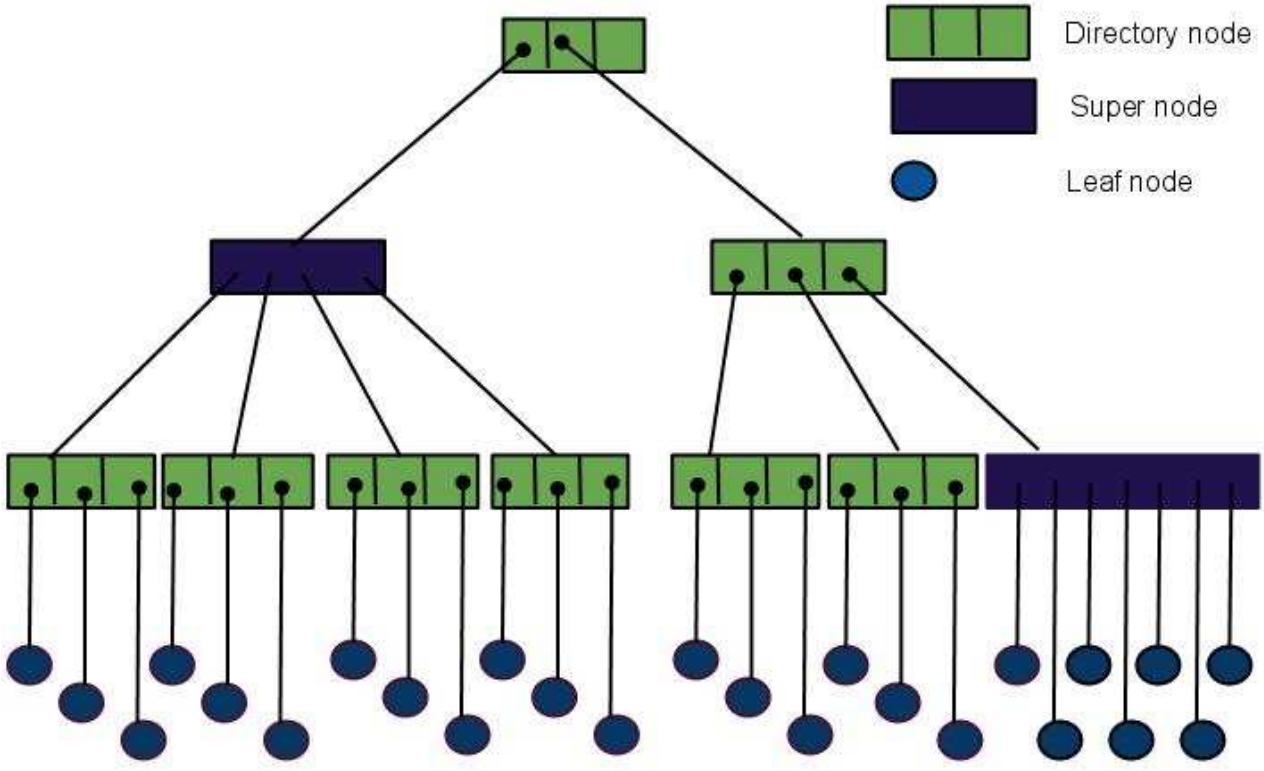

Figure 5: Structure of R-tree

As mentioned above, when inserting a new entry into the R-tree, if there is no space left in a node then the R-tree splits nodes. Like the R-tree, the X-tree also splits when a new node is inserted. However, splitting in the tree introduces overlap that would result in an inefficient directory structure. Therefore, in the X-tree a super-node is created during insertion only if there is no other possibility to avoid overlap extending a directory node over the normal directory size. Consequently, the super-node may be large and a linear scan on a large super-node can be a problem. In many cases, it is possible to avoid creating or extending a super-node by choosing an *overlap-minimal split axis* [15]. The overlap minimal split axis taken over a directory node tries to partition the set of MBRs of the node into two small sets such that the two subsets has no or minimal overlap.

When a new entry is added into a leaf node in an R-tree, if the leaf node splits, it causes its parent to have to also update its MBR and list of children. If the parent node still has space left, it will add a new split node as a new child. Otherwise, the parent node has to be split too. It is said that splitting from the leaf node triggers the splitting upwards to the root of the tree. In contrast, with the X-tree, if there is no room left in a directory node the directory node can extend to be a super-node instead of splitting. Therefore, other nodes are intact. Insertion into an X-tree makes the tree change mainly at super-nodes.

Since the original X-tree was proposed [15], there have been several implementations of X-trees. One of them is the X+-tree [19]. The X+-tree allows the increase of the size of super-nodes in the X-tree to some degree. Technically, in order to avoid overlap, which is bad for performance, a super node might grow during the insertion. However, the linear scan of a large super node can be a problem. In the X-tree, the size of a super-node can be many times larger than size of a normal node. In the X+-tree, the size of super-node is at most the size of a normal node multiplied by a given user parameter MAX_X_SNODE. When the super-node becomes larger than the upper limit, the super-node has to be split into two new nodes.

Credits of the X+-tree implementation goes to Cheung Ka Leong and Wong Chi Wing from the Database Research Group at the Chinese University of Hong Kong [19].

2.3.3 Improved KNN-indexes

Most of the existing approaches centered on solving the nearest neighbor search on a prior built index while expanding the neighborhood around the query point until the number of desired closest points is reached. Furthermore, it has been theoretically proven that those approaches access a large portion of the data space in high dimensions. Therefore, it is still expensive when the dimensionality increases [1].

Stefan Berchtold and et al. have introduced another approach, which pre-calculates and indexes the result of any nearest neighbor search. It is done by computing the *Voronoi cell* of each data point. The Vornonoi cell of a data point *q* consists of all potential query points *p*, which has *q* as the nearest neighbor. Therefore, finding the nearest neighbor of the query point *p* is turned to locating the Voronoi cell in which *p locates*. However storing explicitly the Voronoi cell is complex. The authors proposed to approximate the Voronoi cell as an MBR, [16] that can be stored in a multi dimensional index structure. The approximation over the data space results in a set of MBRs. The search, which is to in find which cell a query point is located, becomes finding in which MBR the query point *p* is located. The search is efficient thanks to the multi dimensional index structure. Experiments have shown that the execution time of the new approach is 4 times faster than the X-tree [16].

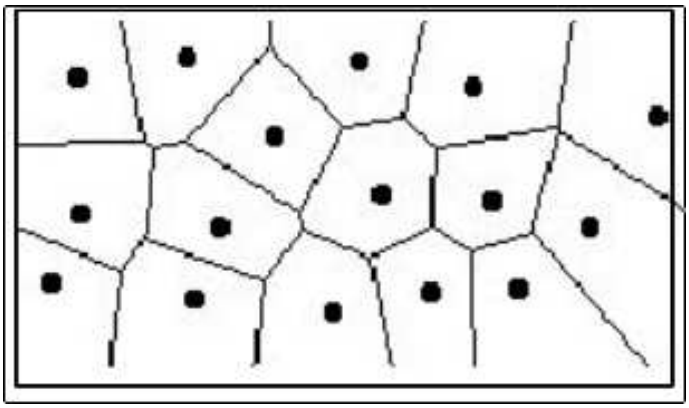

Figure 6: Voronoi cells

This approach is not a subject to be investigated here, which focuses on the X+-tree with some modifications to make it integrated into Amos II. The pre-calculation solution space might be considered as a future work.

2.4 Amos II

Amos II (Active Mediator Object System) [14] is an extensible database system, which has a functional database model. The functional database model used in Amos II consists of *objects, types*, and *functions*. An Amos II database is stored in main memory until it is flushed to disk by explicit commands. It is interacted through an object relational and functional query language called AmosQL. The database system can be used as a stand-alone database, client-server database, or a collection of several distributed Amos database peers.

Amos II also supports mining data by providing functions for data analysis, grouped aggregation, and data visualization. These all functions center on the datatype *Vector* that can be used to express multi dimensional data typically used in data mining.

Furthermore, it is possible to access external program from Amos II through a *callout* interface where Amos II functions are implemented in some regular programming language (e.g. Java or C). The *AmosXtree* system presented in this Thesis takes advantages of the callout interface of Amos II. In particularly, high dimensional data stored in Amos II is indexed with index data constructed, queried, and stored by an X-tree implemented as a foreign function in C.

2.4.1 Types and Objects

Types in Amos II correspond to *entity types* in the Entity-relationship (ER) model. The types are also related to *classes* in object oriented (OOP) programming languages. The nature of types is arranging and ordering objects into categorizes which have their own set of behaviors described by *functions*.

Amos II allows the user to declare *user defined types* besides those included in the system as built-in types such as *Integer* or *Real*. A *user defined type* can be declared with *create type* command. It is possible to construct properties along with a new type. Analogous with a *class* in OOP a subtype inherits all properties of its super type. An instance of u*ser* defined type is and object stored in the database. An object is associated with an *object identifier* (OID). For example, the following code defines the types *Album* and *FamilyAlbum* followed by the creation of an instances and setting a function value:

```
create type Album properties (name Charstring);
create type FamilyAlbum under Album;
```

```
create FamilyAlbum instances :pa;
set name(:pa) = 'Summer-2009';
```

User defined types and *objects* are deleted from database with the *delete* statement.

2.4.2 Functions

Properties and relations between objects are expressed as *functions*. For example, the *Album* type has an attribute, *name*. It is modeled as a function *name* in which an object of type *Album* is argument and returns a string

```
create function name(Album) → Charstring as stored.
```

The above function is stored in the database describing the relation between an *Album* and its name (argument and result). Since it is placed in the database, it is also known as a *stored function*, which is one kind of functions. Other types of functions are *derived functions*, *foreign functions*, *stored procedures,* and *overloaded functions*. *Derived functions* are parameterized views defined by a single query*. Foreign functions* are implemented in some programming languages (ex. Java, C, or Lisp) and provide external interfaces, which make Amos II extensible. A foreign function is linked to a symbolic object in an Amos II database. The details of external interfaces are addressed in the following section. Stored procedures are composed of a sequence of AmosQL statements. In most cases, store procedures are used as functions that update the database. Finally, *overloaded functions* are functions having the same symbolic names but different signatures.

2.4.3 External interfaces

External interfaces allow to call Amos II from other applications (the *callin* interface) or to extend the kernel system (the *callout* interface). There are supported external interfaces to several programming languages, e.g. C, Lisp, and Java. The C interface is on the lower level, which is recommended when high performance is required, at the expense of a lower programming level.

In this Thesis work, the X+-tree index structure is implemented in C as foreign functions through the *callout* interface. In section 3.4.1.3, the implementation of the X+-tree

extension shows an example of how to use the *callout* interface.

3. The AmosXtree system

The extension of Amos II with the X+-tree index is called the *AmosXtree system*. The system architecture is composed of these following components:

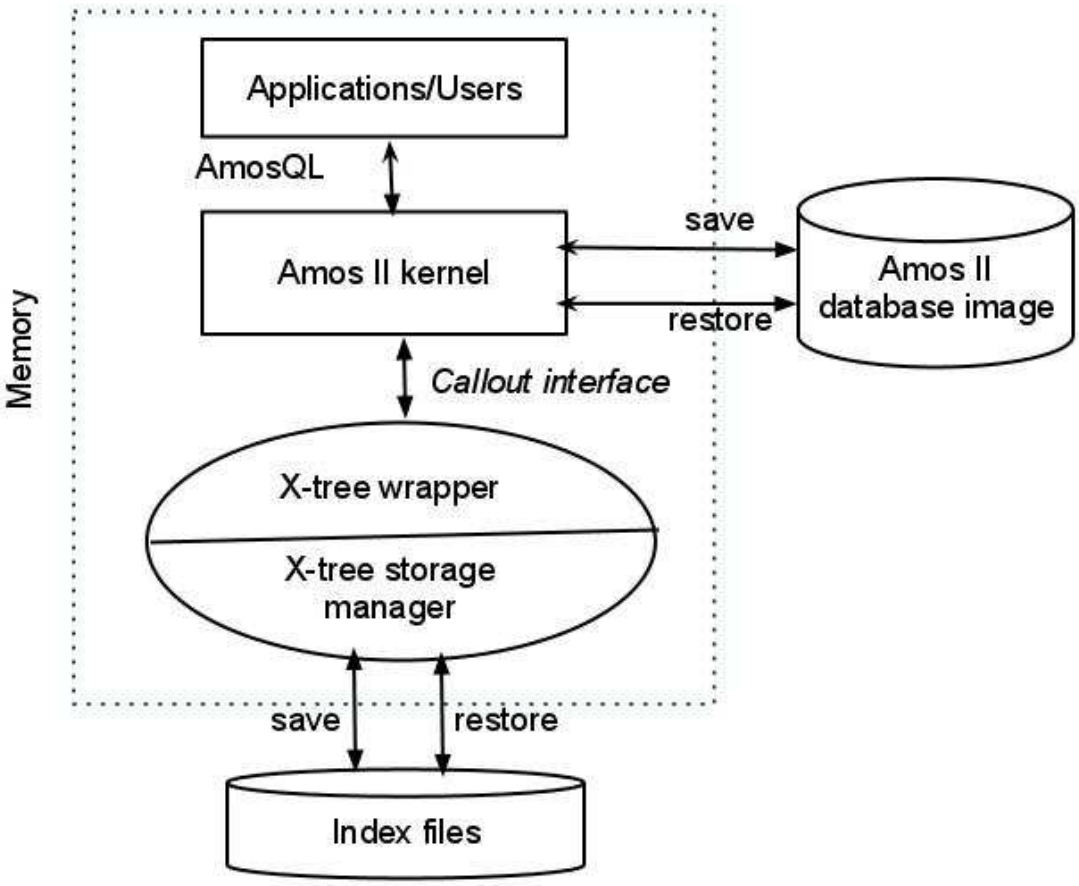

Figure 7: System architecture of AmosXtree

The *application* is some application program or a console application where the user can interact with system by issuing AmosQL and retrieve the result in return. Through the application the user can create data or import data from somewhere into the database.

The Amos *II database image* is a disk area where all database objects, types, and functions are stored. The Amos II kernel loads the corresponding Amos database image into main memory during the initialization. Changes made during the session are saved to disk with an explicit *save* command.

The AmosXtree consists of two sub-components the *Xtree wrapper* and the *Xtree storage manager*. The former contains the implementation of the foreign functions provided by AmosXtree. The latter contains code for building an index tree, searching the X-tree,

saving the tree, and re-loading it from disk.

High dimensional data such as feature vectors are stored in the database and indexed by calling out to *Xtree wrapper*. Accordingly, an index structure is built up in main memory and an identifier (ID) of the index tree is returned to Amos II. Using that ID, the system can determine which index structure it wants to search in a user query.

The AmosXtree represent the X-tree using a main memory data representation. All data can be saves on the disk as a *database image* file. Each X-tree index is saved in a separate *index file*. All data, including the index trees, are flushed to disk by the *save* command in Amos II. Inversely, the index trees are read from the index files into main memory when Amos II is started. The synchronization of the image and index files is controlled by a hook facility provided by Amos II. The details of the implementation of restoring and saving are described later.

3.2 Example of use – Indexing picture database

To illustrate the functionality of AmosXtree an example is given on how to define an index structure in searching pictures. Without indexing, searching similar pictures among 100.000 pictures stored in a database or a directory would require linear scan of tables or files. The example shows how to utilize an X-tree index for indexing multi-media objects stored in regular files.

3.2.1 Schema of picture database

Figure 8 shows an Entity Relationship (ER) schema of the photo database, which is used as example in this section.

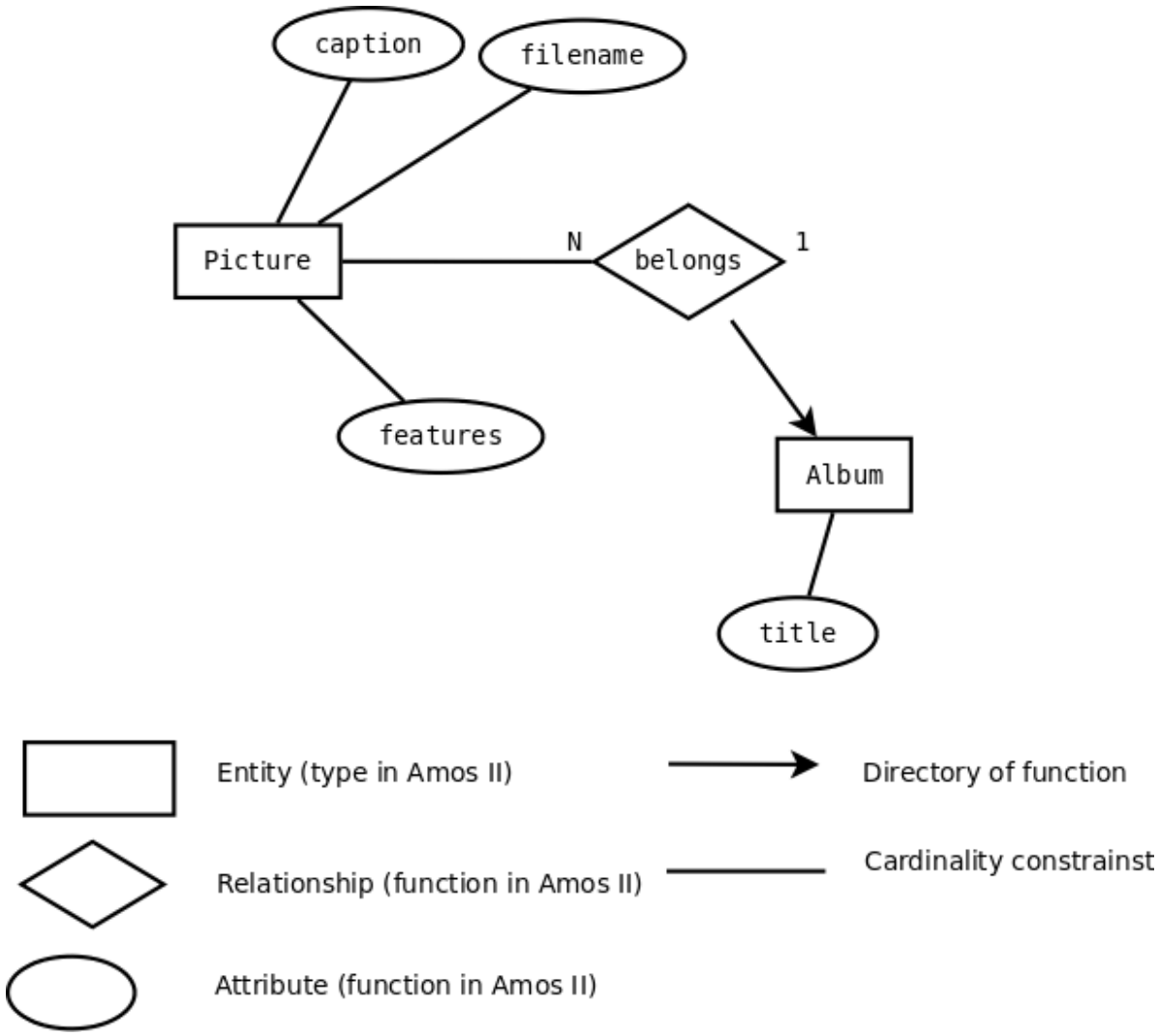

Figure 8: Entity Relationship schema describing photo database

People often take pictures during the vacation and then categorize them in an album. Analogously, there are two types of objects in the database: *Picture* and *Album*. In short, a picture in the database represents a real picture located in a disk file, e.g. a JPEG file. A picture in the database does not contain a large bitmap. Instead, the database stores only the name of the JPEG file along with some interested properties (features) such as brightness or face geometry extracted from the real picture. These features are extracted by a *featureExtractor* function, which is defined as a foreign function. In the Thesis work, the feature extractor is *colorHistogramExtractor* that simply calculates the distribution of color in an image. It is implemented in Java. A stored procedure stores the result of an extraction as *features* of a picture in the database. However, the search over the features will not scale up when the database size is large or when the feature vector is complex. Therefore, it is important to index the features. The example shows how one can make an

index over feature vectors of pictures with AmosXtree.

In addition, a picture has also a *caption* property, which is used to describe a picture, and a *filename* property that holds the name of the disk file where it is stored.

Pictures are categorized in albums where each album has a descriptive *title*. The relationship between pictures and albums is also shown in Figure 8. It is a function *album* storing the album of a given picture.

The details of the implementation of the schema in Amos II can be seen in Appendix B. It includes examples of how to populate the photo database (i.e. importing data from a file, extracting features of an image, and indexing a new inserted picture).

3.2.2 Indexing pictures with AmosXtree

To create a global feature index represented by an X+-tree the following function is called:

```
create function createFeatureIndex() -> Integer xtID as …
```

After creating the feature index, the application defines a picture insertion function to first extract the feature vector. In our application a color histogram is extracted from each new picture by calling a function *colorHistogramExtractor* and then insert the returned feature vector into the index. These two functions with theirs signatures are declared as:

```
create function colorHistogramExtractor (Charstring filename)->
Vector of Number as … ;

create function addFeatureIndex(Integer xtID, Vector of Number,
Picture p) -> Boolean as …
```

The complete implementation of these two functions and the picture insertion function can be seen in the Appendix B: The Photo-Album database.

AmosXtree also provides ways to write similarity search and KNN search on indexed data. The following functions are implemented to search for pictures (see Appendix B: The Photo-Album database):

```
create function similar (Picture  p, Real distance)-> Bag of
Object as …

create function knearest(Picture p, Integer k)-> Bag of Object as
…
```

After loading pictures into the database one can search the contents of pictures based on some criteria. For example, the following query searches for the six most similar pictures compared to a picture with caption “Last summer”:

```
select caption(p)
from Picture p, Picture q
where p in knearest(q, 6) and caption(q)= "Last summer";
```

The following query finds the pictures that are similar to a favorite one where the similarity is 0.65:

```
select caption(p)from Picture p, Picture q
where p in similar(q, 0.65) and caption(q)="Last summer";
```

The following query selects titles of albums that have pictures similar to a given one:

```
select distinct title(album(p))
from Picture p, Picture q
where p in knearest(q,10) and caption(q)="Last summer");
```

Because feature vectors have been indexed, the search is dramatically faster than without index, as will be shown later.

3.3 Implementation

Since the original X-tree was proposed, there have been several implementations of X-trees. Some of them are written in Java, some are in C++, and few of them are in C. Therefore, if possible, the core implementation of our X-tree should use existing implementations rather starting from scratch. To make it easy to integrate with Amos II, the choice of external program should be in C or Java.

Moreover, regarding the performance, a C program often outperforms Java programs since Java byte-codes are compiled or interpreted by the Java Virtual Machine resulting in performance penalty.

The core of the X+-tree implementation in [15] is reused, with changes and additions made to data structure and functions.

3.3.1 Modules

The existing modules of the X+-tree implementation were re-organized to simplify the introduction of the new code. The AmosXtree system is structured as follows:

- xtree.h

This module contains AmosXtree's interfaces creation, insertion, and search. Furthermore, it hosts all macro, data structures, and constants.

- amosXtree.c

This module implements the AmosXtree front-end defining the implementation of Amos II foreign functions, binding function, and the main function of the AmosXtree system.

- xtree.c

This module consists of implementations of basic X+-tree functionalities such as creation (*xtree_make*) or insertion (*xtree_put*).

- xpbuild.c

This module hosts utility functions used mainly to build new index trees.

- xsearch.c

This module implements functions for searching X+-tree indexes.

- xfile.c

This module contains function to save the X+-tree index trees to disk and function to reload saved X+trees from disk to main memory.

Besides, there is a module written in Java to extract color histogram form an image.

- ColorHistogramExtractor.java

This module contains functions to extract color histogram of an image. The choice of programming language is Java because it has built-in functions for processing images such as Java.2D package, which allows encoding, and decoding from and to several image formats: JPG, BMP, PNG, etc. Converting from other formats to BMP implies producing an array of pixels, which is can be used for implementing follow up image processing algorithms.

The Java code of course must be run on the Java Virtual Machine (JVM). Amos II can be run with JVM by calling the script *javaamos*. However, AmosXtree is a C program compiled

on Visual C ++ 6.0 compiler. Therefore, in this application Amos II has to callout to both C and Java. The solution here is that Amos II callouts to Java through C. In other words, Amos II callouts to the foreign function *colorHistogramExtractor* in C. It in turn redirects to a real Java implementation. The following module helps calling Java from C. It is suggested by Michael Havey in his article on Java Developer Journal [11]

- cj.c

This module contains functions to call Java from C. Technically; it wraps some functions and data structures of Java Native interface (JNI), which allows creating Java Virtual Machine (JVM), destroying JVM, and manipulating Java's classes and objects. On Windows, the JVM is included in Java SDK as a dynamic library "jvm.dll" which exposes JNI as public exports. Therefore, an executable C program linking jvm.dll makes it possible to call from C to Java.

3.3.2 Node data structure

The X-tree has three different kinds of nodes: directory nodes, data nodes, and super nodes [15]. In the X+-tree implementation these three kinds of nodes are represented through a data structure *node_type* with definition:

```
struct node {
    float *a;
    float *b;
    int id;
    int attribute;
    int vacancy;
    struct node *parent;
    struct node **ptr;
    int snodeSize;
    oidtype object; // to hold object handle of Amos II
    int valuable;
};
typdef struct node node_type;
```

- The two fields **a*, **b* are used to determine an MBR of the current node.
- *id* is the unique identifier of the node.
- *vacancy* holds the number of available entries a node can have. The possible number of entries of a node is in between *m* (min) and *M* (max). Both *m* and *M* can be configured through a configuration file. (see 3.5.3 Configuration).
- *parent* is a pointer to the node's parent. The parent of a node is needed when changes to a node low in the tree trigger changes upward the tree. For example, inserting a new entry to a node causes its parent's MBR to be updated.
- *ptr* holds a pointer to the list of sub-nodes.
- *snodeSize* holds the size of the node if the current node is a super node.
- *object* holds a reference to an Amos II database object stored in the node. It is represented as an Amos II *object handle* [18]. It is a new attribute added to the original node structure of X+-tree.

The *object* is assigned when a new node inserted to the index tree. This is defined as the Amos II foreign function *xtree_put* (see Appendix A: The XtreeWrapper interface). It is called to index a feature vector of an Amos II object. An entry of a leaf node in the index tree references the object in Amos II by holding its object handle.

```
a_let(oid_obj, a_getelem(t, 2, FALSE));
```

A node's *object* is released with *a_free(node->object)* when the node is deleted. The node marked as deleted shall be ignored in the search.

3.3.3 Search result structure.

Search functions are implemented as tail recursive functions in which a search result is passed along with the call. The search result is defined as a linked list of structures named *NN_type* where NN means 'nearest neighbor'. The structure *NN_type* is used for KNN search and similarity search also.

```
typedef struct NN {
    double dist;
    int oid;
    struct node *pointer;
```

```
    struct NN *next;
    oidtype object;

} NN_type;
```

NN_type has the following fields:

- *pointer* is a pointer to a node in the index tree which matches with the given search criteria.
- *dist* holds the distance between the found node and the given query data object.
- *next* holds the next matching node.
- *object* holds the Amos II object stored in the node. See 3.3.2

3. 4 The Xtree wrapper.

The Xtree wrapper contains the implementation of foreign functions defined in Amos II. The implementation invokes the X+-tree functionalities. The implementation code of a foreign function has the following steps:

- Extract parameter values
- Call the corresponding X+-tree C function
- Emit the result tuple as a tuple to Amos II.

Below is a synopsis of the implementation of the foreign function *xtree_knn_search*.

```
void xtree_knn_search(a_callcontext cxt, a_tuple t)
{   . . .
    dcl_oid(oid_iVector);
    dcl_tuple(tpl_feature);
    . . .
    a_newtuple(tpl_feature, size, FALSE);
    a_getseqelem(t, 1, tpl_feature, FALSE);
    {
         queryPoints = (float *) malloc(sizeof(float) * size);
```

```
          for(i = 0 ; i < size; i++) {
               queryPoints[i] =
                    a_getdoubleelem(a_elt(oid_iVector,i));
          }
          root = a_getintelem(t,0, FALSE);
          k = a_getintelem(t, 2, FALSE);
         NN = k_NN_search(root, queryPoints, k);
         while (NN != NULL && NN->oid != UNDEFINED) {
                a_setobjectelem(t, 3, NN->object , FALSE);
                a_emit(cxt,t,FALSE);
                NN = NN->next;
            }
        }
    }
    . . .
    return;
}
```

Then implementation of a foreign Amos II function must be bound to a corresponding symbolic name. The function *a_extfunction* binds a C function *fn* to an Amos II symbol *name*:

```
a_extfunction(char *name, external_predicate fn);
```

The bindings are placed in one C function named *registerXfunctions* in file *amosXtree.c*

```
void register_X_functions() {
     a_extfunction("xtree_make",xtree_make);
     a_setpredparam("xtree_make","create a new Xtree");

     a_extfunction("xtree_put",xtree_put);
     a_setpredparam("xtree_put",
                    "put a pair object/value into Xtree");
```

```
            . . .
    a_extfunction("xtree_save",xtree_save);
    a_setpredparam("xtree_save","save Xtree");
    a_extfunction("xtree_load",xtree_load);
    a_setpredparam("xtree_load","load Xtree");
         . . .
    a_extfunction("xtree_similarity_search",
                  ”xtree_similarity_search”);
    a_setpredparam("xtree_similarity_search",
                  "find node(s) that closes in a distance
                  to the given node");
    a_extfunction("xtree_knn_search", xtree_knn_search);
    a_setpredparam("xtree_knn_search",
                  "find node(s) as KNN to the given node");
}
```

3.5 The Xtree storage manager.

The Xtree storage manager is responsible for saving and restoring the index structure. This section mainly addresses handling the identifier of the index tree, its configuration, and the index files.

3.5.1 Handling Xtree identifiers

An index tree is created and manipulated in main memory unless it is explicitly flushed into a file on disk. In order to keep track of indexes of a database image, there is a table to store the identifiers (IDs) of the index tree:

```
create function xtreeIds()-> Bag of Integer as stored;
```

The ID of an index tree is generated in Amos II and it is then passed to foreign function *xtree_make* to create a new index tree in the main memory. The code below shows the creation of a new Xtree's ID in Amos II.

```
create function lastest_xtreeIds() -> Integer l
as
```

```
begin
    if (notany(xtreeIds())) then  set l = 0
    else
         set l  = maxagg(xtreeIds());
         result  l;
    end;

/*create index*/
create function createIndex() -> Integer p
as
begin declare Integer id;
    set id = lastest_xtreeIds() + 1;
    if xtree_make(id) then
    begin
         add xtreeIds() = id;
         result id;
    end;
end;
```

The ID will be passed to other foreign functions as a parameter to identify a particular index tree. The foreign functions receive a call from Amos II with an ID of the index tree, for example insertion into the index tree. The Xtree storage manager needs to find the memory address where the corresponding index tree is allocated for the given ID. Therefore, there is a relationship between an identifier of a tree and its address in main memory. In order to locate the tree in main memory from its ID, the Xtree storage manager maintains a linked list in which an entry represents a relationship between a unique ID and the corresponding tree address in main memory. The entry has type of *ListTree_type* described below

```
typedef struct ListTree {
    struct ListTree_type *next;
```

```
    int address;
    int id;
} ListTree_type;
```

- *next* : a pointer to another entry of a linked list
- *address: holds* memory address of the tree. It is useful to locate the tree.
- *Id:* ID of the tree.

The synopsis code below shows the update of list index trees when an index tree is created:

```
node_type *root;
. . .
// assigns the ID
(*root).id = a_getintelem(t, 0, FALSE);

// add one entry to the linked list
updateListTree((*root).id, (int)root);
// emit the ID to Amos II
. . .
```

In addition, there is a requirement that the index tree can be saved to disk. One can later reload the saved index tree back to main memory and it must be possible to query the loaded index tree. Therefore, it is necessary to save the ID of the index tree in the index files. This makes the ID of the tree stored in the database image and IDs stored in index files synchronized.

```
// load index tree from file
root = o_xtree_load(filename);

// add an entry to the linked list
updateListTree((*root).id, (int) root);
// emit the ID to Amos II
```

```
...
```

3.5.2 Saving and restoring index files

- Hooking functions

An index tree is created in main memory and is saved to disk with an explicit *save* command. To make database persistent, a function *index_rolloutfn* to store index trees should be automatically called before the database image is saved. In the other direction, a function *index_rollinfn* that loads index files should be implicitly invoked when the database image is initialized.

```
create function index_rolloutfn(Charstring image) -> Boolean;
create function index_rollinfn(Charstring image) -> Boolean;
```

Amos II provides a way to hook these two functions into the database system so that the *index_rolloutfn* is called before the database imaged is saved and the *index_rollinfn* is called when the database image is initialized. The hooking is done by using a built-in function *register_saver* in Amos II.

```
register_saver(#'index_rolloufnt', #'index_rollinfn');
```

The complete AmosQL code can be seen in Appendix C: Saving and restoring index files

- Naming index filename.

  Filename of an index tree is made up by concatenating the database image name, index tree's ID, and extension "xt".

```
Filename = image name + ID + ".xt"
```

  Example: foo1.xt, foo2.xt . . .

- Managing list of index trees to be saved / loaded.

When saving, all index trees in main memory should be saved to disk. Amos II goes through a list of index tree's IDs returned by function *xtreeIds()* (see 3.5.1 Handling Xtree identifiers) and calls a foreign function to save the corresponding index tree.

When initializing the database, each index file whose filename formed with ID in the list *xtreeIds()* is loaded from disk to main memory.

3.5.3 Configuration

The index tree is highly customizable by being configured with parameters. A data structure is defined to hold these parameters: *typdef struct config*.

```
typedef struct config {
    int dim;
    int m;
    int M;
    debugMode;
    ,,,
} config_type;
```

- *m* : Minimum entries of a node
- *M* : Maximum entries of a node
- *dim* : Number of dimensions
- *debugMode* : Debug mode of AmosXtree. If the Debug Mode is off (debug *Mode* = 0) there is no logging to console.

These parameters are placed in a configuration file named as *xtree.config*, which can be modified easily. When AmosXtree start ups, the configuration file is read into a global variable named as *master_config*. This is a default configuration for all new index trees.

```
void initialize(config_type *config) {
 . . .
fp = fopen(CONFIG_FILE, "r");
fscanf(fp, "m=%d\n", &master_confg->m);
fscanf(fp, "M=%d\n", &master_confg->M);
fscanf(fp, "dim=%d\n", &master_confg->dim);
. . .
}
```

The master_config is used to create a new index tree.

```
// Makes an Xtree with master config
```

```
root = o_xtree_make(master_config);
```

One can modify the configuration file after index trees were saved to disk. Therefore, a loaded index tree might have a different configuration compared to others. In other words, each index tree can have its own configuration. Technically, AmosXtree maintains a list of index tree's configurations. Recall that, we did manage a list type of *ListTree_type* in order to keep track of index trees. The struct *ListTree_type* (see 3.5.1 Handling Xtree identifiers) is added another field *config* to host the configuration of the corresponding tree.

```
typedef struct ListTree {
    . . .
    config_type config;
} ListTree_type;
```

After a new index tree is created, its information (configuration, ID, address) is added to the list

```
updateListTree((*root).id, (int)root, master_config);
```

When the index tree is saved to disk, its configuration is also saved in the index file. In the other way around, the configuration stored in the index file is read first to construct the index tree from the index file. In this way the existing index files can be loaded, and modified even if the configuration file *xtree.config* was modified after the index trees was saved. The synopsis code below shows how the configuration is saved in the index file.

```
fp = fopen(filename, "w");
getConfig((*root).id, &fconfig);
// Write the ID of the tree
fprintf(fp, "%d\n", root->id);
// Write the configuration of the tree to the index file
fprintf(fp, "m=%d\n", fconfig.m);
fprintf(fp, "M=%d\n", fconfig.M);
fprintf(fp, "dim=%d\n", fconfig.dim);
```

```
fprintf(fp, "no_histogram=%d\n", fconfig.no_histogram);
. . .
write_inter_node(root, fp, fconfig);
```

3.5.4 Index files

For each tree in main memory, there is a corresponding file on disk. Several index files can be created after a save command is executed in Amos II.

The index file starts with the ID of the index tree in the first line, it then is followed by its configuration (see 3.5.3 Configuration). After the configuration, all nodes are stored. Each node is fully stored along with all its attributes. Non-leaf nodes are saved by the *write_inter_node* (in file *xfile.c*) C function and leaf nodes are saved by the *write_leaf_node* function (in file *xfile.c*)

4. Performance evaluation

This section describes the performance tests in two cases having index on high dimensional data and not having index on data. The experiments are conducted in order to prove that the query time of KNN search scales up with the increase of both database size and dimensionality. The data used in the experiments is uniformly. The experiments are conducted on Microsoft Windows XP SP1, Intel CPU 1.6GHz, 192 MB RAM using the Visual C++ 6.0 compiler.

4.1 Setting up the experiment.

After *colorHistogramExtractor* was implemented, the application runs with real pictures. However, for performance evaluation purpose it requires a large amount of data and therefore these data are simulated. Therefore, the experiments assume having feature vectors as the result of extracting interested properties from pictures stored in test data files. One test data file consists of many rows where each row is an integer vector representing a (simulated) picture. A row is in format <id x1 x2 x3 . . . xd> with d being the number of dimensions. The test data provides data files with d from 4 to 20. Data is generated at random.

Setting up the test environment is compulsory to create a clean test environment. For example, the command *setup_test.cmd* is used for setting up the test environment. In both cases, the data types *Picture* and *Album* and all relevant functions are pre-created and saved into the Amos II database image.

The query times are measured with and without the X+-tree index and then plotted as graphs to visualize the scalability.

4.2 Evaluation

Two experiments show the scalability of AmosXtree with dimensionality and with database size.

- **Scalability with database size**

The first experiment investigates the scalability of the index when the size of the database is increased. Let define K is a number of nearest neighbors and D is the number dimensions of the database. The parameters are assigned as K = 5, D = 8, and the database size is scaled from 20 to 100 Mbs.

Figure 9 compares the query time with index and without index when the database size increases. The x-axis shows the increase of database size while y is the search time. The query time without index to find K = 5 nearest neighbors increases linearly while the query time with index is very close to the x axis.

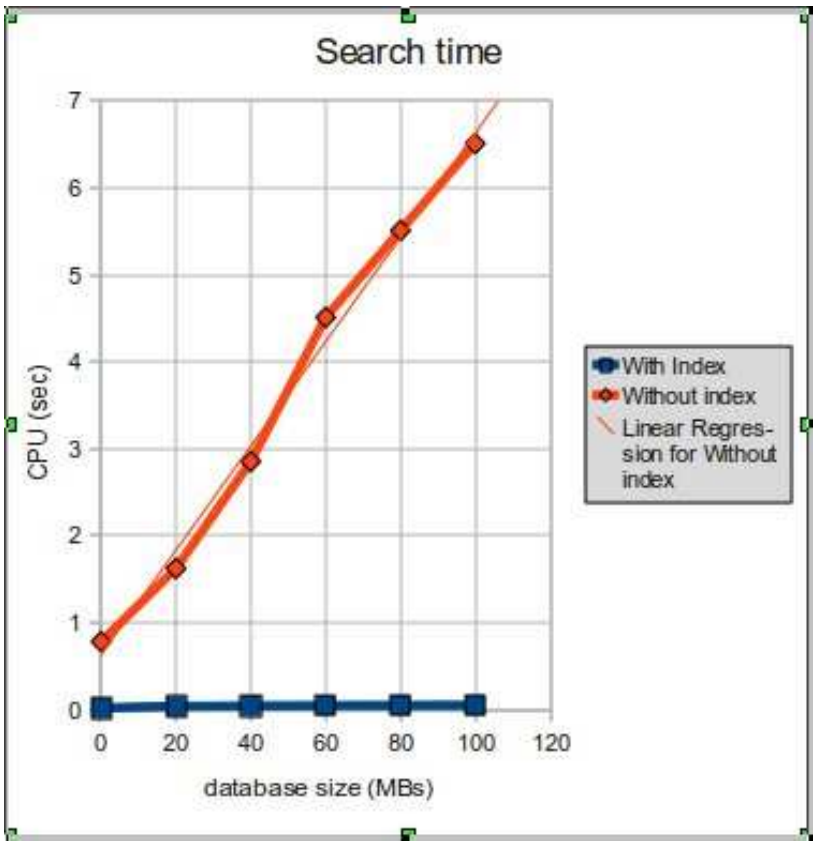

Figure 9 Query time of KNN queries (using and not using index)

Figure 10 plots the query time with index to have a clear observation about its scalability. The y-axis has linear scale on search time while the x-axis has logarithmic scale on size of database.

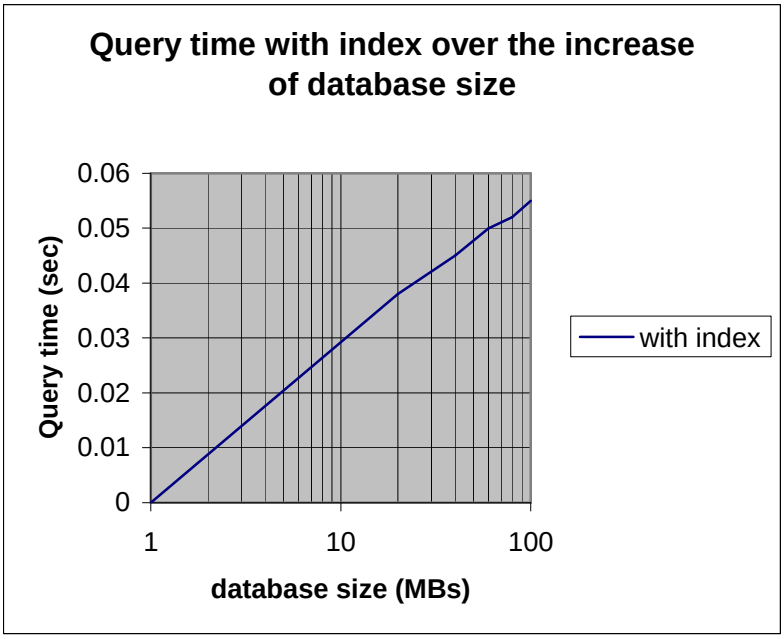

Figure 10 Query time with index over the increase of database size

As expected, the trend of query time in case of using index is close to linear with the logarithm of database size which means that it scales well for large databases. By contrast, without the index when the database size increases, the query time needed to perform the search (KNN) grows linearly.

❖ **Scalability with dimensionality**

The second experiment evaluated KNN queries in two cases using the same number (100.000) of data vectors over the different dimensions. Using the same amount of data implies that the size of database is linearly increasing with the dimensions.

The experiment's result presented in Figure 11 used uniformly distributed point data to search for K = 5 neighbors. Figure 11 Query time of KNN queries (with and without index) on 100000 objects shows the query times with index and without index, respectively. The number of dimension varies from 5 to 20. The x-axis presents the dimension while the y-axis presents the query time in millisecond. In case there is no index on high dimensional data (the red line), the query time it took to find K = 5 nearest neighbors is linearly increasing with the dimensionality. In contrast, the query time for performing the KNN search with indexed data very close to the x-axis, which shows that the X+-tree provides good performance compared to without index search over the database size.

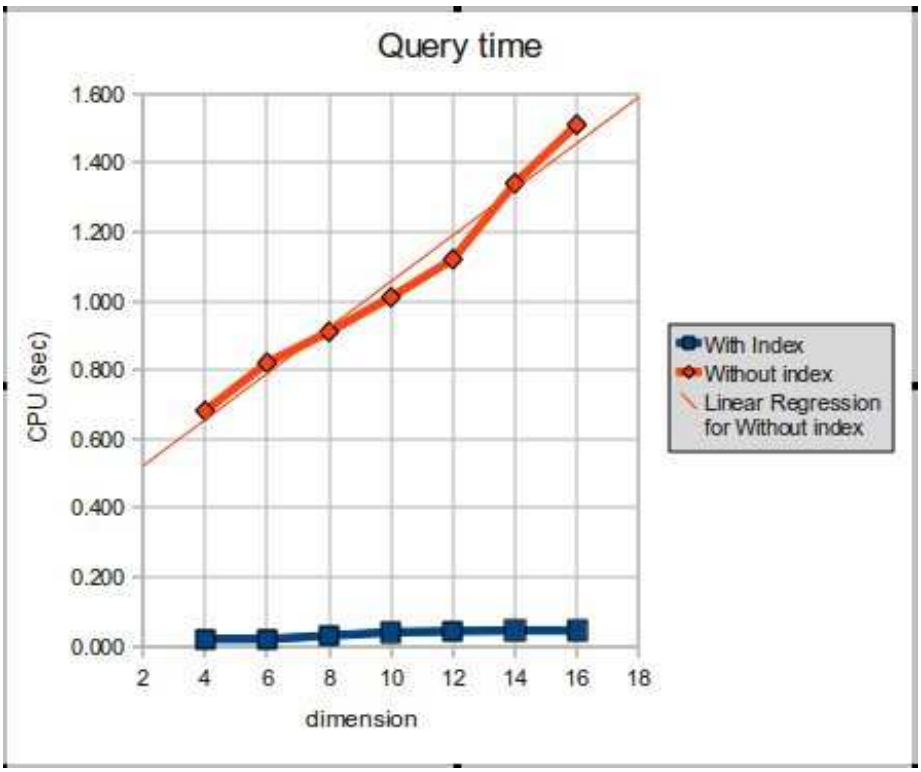

Figure 11 Query time of KNN queries (with and without index) on 100000 objects

From Figure 11, it is difficult to see the increase of the query time with index. Figure 12 shows only the query time with index when dimension increases. In this figure, x-axis has logarithmic scale and y-axis has linear scale. While number of dimensions is increasing, the query time also increases. Its increase trend is almost linear. On other words, it shows that Xtree scales very well when the number of dimensions is less than or 16 (the highest dimension used in this experiment).

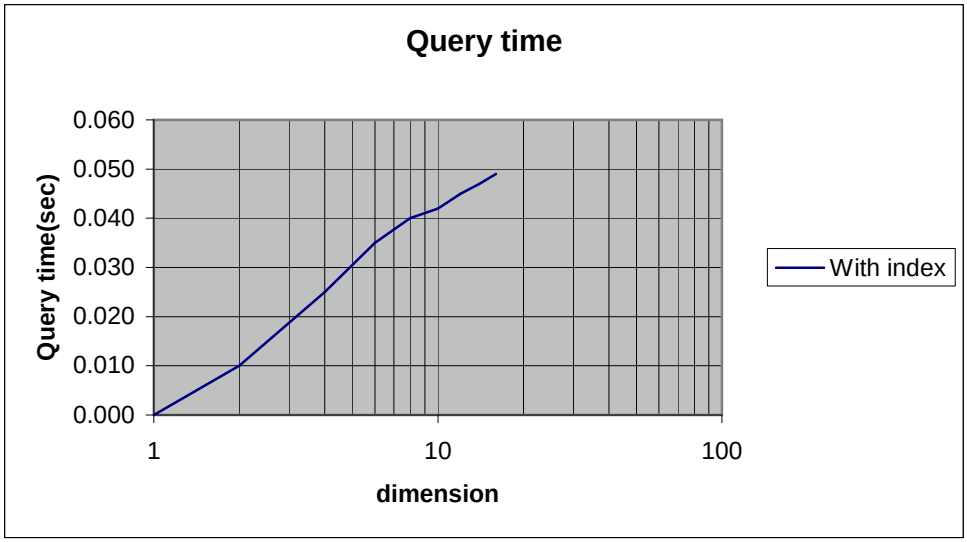

Figure 12 Query time of KNN query using index over 100000 objects

Empirically, the experiments draw a conclusion that having index on high dimensional data can improve the performance of searching on it, particularly in KNN queries.

5. Conclusions & Future work

This Thesis work first studied different index structures for high dimensional data. Then a chosen index implementation, the X+-tree, was integrated into the extensible functional database system Amos II. The result of this work is a system named as AmosXtree, which adds X+-tree indexes to Amos II.

AmosXtree makes it efficient to search similarity in high dimensional data by using the X+-tree index structure to index data. AmosXtree uses foreign functions to implement the index interfaces. The index structure is implemented in C, which has advantages of good performance compared to other languages. Furthermore, the X+-tree index structure was shown experimentally to provide very good scalability over both the database size and dimensionality.

By equipping Amos II with X+-tree, an index structure, on high dimensional data, it fosters the creation of applications based on searching high dimensional data stored in Amos II. The Thesis work has demonstrated the skeleton for searching similar pictures. This provides a useful tool for picture management. However, The Thesis work needs to be extended in order to make the demonstration fully complete by using real feature extractor function on real picture files rather than simulated ones.

Other works can be followed up to get then X+-tree index fully integrated into the query language. What needed here is *query transformation* that transparently transforms queries to utilize the X+-tree index. Furthermore, it should be possible to put an X+-tree index on any parameter of a stored function by a simple system command. Finally, a cost model for X+-tree should be provides as functions to aid the cost-based query optimizer.

References.

[1] Berchtold S., Böhm C., Keim D., Kriegel H.-P: *A Cost Model for Nearest Neighbor Search in High-Di-mensional Data Spaces*, Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS), Tucson.

[2] Codd, E.F : *1981 Turing Award Lecture - Relational Database: A Practical Foundation for Productivity*

[3] David Hand, Heikki Mannila Padhraic Sym : *Principles of Data Mining,* The MIT Press, 1 edition, 2001

[4] Faloutsos, C., et al.: *Efficient and Effective Querying by Image Content*. In Journal of Intelligent Information Systems, Vol.3, No.3 (1994) 231-262

[5] Jiyuan An; Yi-Ping Phoebe Chen: *Finding Rule Groups to Classify High Dimensional Gene Expression Datasets*, ICPR 2006. 18th International Conference, 2006

[6] J. G. Liu - Permanent address: Remote Sensing Laboratory, China University of Geosciences, Wuthan, People's Republic of China.a;J. McM. MOORE

[7] Kardi Teknomo's tutorial: *What is Similarity and Dissimilarity?* http://people.revoledu.com/kardi/tutorial/Similarity/WhatIsSimilarity.html#Distance

[8] Kevin Beyer, Jonathan Goldstein, Raghu Ramakishnan, and Uri Shaft: *When Is "Nearest Neighbor" Meaningful?* , CS Dept., University of Wisconsin-Madison.

[9] Litwin, Witold (1980): *Linear hashing: A new tool for file and table addressing*, Proc. 6th Conference on Very Large Databases: 212–223

[10] Mohamed J. Zaki : *High Dimensional Data Analysis* - http://www.cs.rpi.edu/~zaki/index.php.

[11] Michael Havey: *Calling Java From C, Java Developer Journal*. Accessed on 15 th May 2010.

[12] Pang-Ning Tan, Michael Steinbach, Vipin Kumar : *Introduction to Data Mining*

[13] Rudolf Bayer : *The universal B-tree for multidimensional indexing: general concepts*, Lecture Notes in Computer Science, Springer, ISBN 978-3-540-63343-3

[14] S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, and M. Sköld, *: Amos II User's Manual*.

[15] Stefan Berchtold , Daniel A. Keim , Hans-Peter Kriegel: *The X-tree : An Index Structure for High Dimensional Data,* 1996.

[16] Stefan Berchtold, Bernhard Ertl, Daniel A. Keim, Hans-Peter Kriegel, Thomas Seidl: *Fast Nearest Neighbor Search in High-dimensional Space*, Institute for Computer Science, University of Munich, Germany.

[17] Swain, M.J, Ballard D.H: *Color Indexing*. In Inter. Journal of Computer Vision, Vol. 7, No.1 (1991) 11-32

[18] Tore Risch, D. Elin: *Amos II External Interfaces*, Uppsala Database Laboratory (2000)

[19] X+-tree, *http://appsrv.cse.cuhk.edu.hk/~kdd/program.html,* Database Research Group, Department of Computer Science and Engineering, Hong kong

[20] Xiao Gao Yu, Xiao Peng Yu: *A new K-nearest neighbor searching algorithm based on angular similarity*, Wuhan Institute of Technology, China, ISBN**:** 978-1-4244-2095-7, 2008

Appendix A: The XtreeWrapper interface

This section explains the interfaces provided by the Xtree wrapper. First, it introduces User interfaces containing functions to create a new X+-tree and to use it. At the end of the section, internal interfaces are listed.

❖ User interfaces

- Creating a new X+-tree index:

```
createIndex() -> Integer p
```

The *createFeatureIndex* function takes no input parameter and returns ID of the index tree.

- Adding a feature vector into the index

```
 addFeatureIndex(Integer xtId, Vector of Real f, Object o) ->
Boolean
```

The *addFeatureIndex* function takes ID of the index tree, a feature vector of real, and an object. If the object is added to the index tree, the function returns TRUE. Otherwise, it returns FALSE.

- Removing an object from an X+-tree index:

```
deleteFeatureIndex(Integer xtId, Object o) -> Boolean
```

Its input parameters are the index tree's ID, and an object to be deleted from the index. If a node is deleted successfully, the function returns TRUE. Otherwise, it returns FALSE.

- Searching similar object given the degree of difference in an X+-tree index:

```
similar_search(Integer xtId, Object o, Real distance) ->Bag of
Object
```

Input parameters of the *similar_search* function are the index tree's ID, the object to be searched for, and the distance specifying how similar it should be. The search returns a bag of objects similar to the given object.

- Searching K nearest neighbors in an X+-tree index:

```
knearest_search(Integer xtId, Object o, Integer k) -> Bag of
Object
```

The *knearest_search* function takes the index tree's ID, the object to be searched for, and the number of neighbors. The output is a bag of objects nearest to the given object.

- Saving X+-tree indexes:

```
save “filename”
```

The index trees are saved together with the *save* command of Amos II. See more in Appendix C: Saving and restoring index files.

- Loading X+-tree indexes:

```
amosXtree  “filename”
```

The index trees are loaded when one starts AmosXtree with a database image. If there are index files associated with the image, then they will be loaded into main memory and be ready to use. See Appendix C: Saving and restoring index files

❖ **Internal functions.**

```
/*------------------------------------------------------------------

Internal functions

------------------------------------------------------------------*/

/*Create an index tree*/

create function xtree_make(Integer xtId) ->Boolean  as foreign
'xtree_make';



/*Add a feature vector and its associated object to the index
tree*/

create function xtree_put(Integer xtId, Vector of Number f, Object
o) -> Boolean as foreign 'xtree_put';



/*Delete a node from the index tree*/

create function xtree_delete(Integer xtId, Vector of Number f) ->
Boolean as foreign 'xtree_delete';
```

```
/*Save the index tree to disk*/

create function xtree_save(Integer xtId, Charstring filename) ->
Boolean as foreign 'xtree_save';


/*Load  the index tree from disk*/

create function xtree_load(Charstring filename)

-> Integer xtId as foreign 'xtree_load';


/*Internal function */

create function xtree_knn_search_fn(Integer xtId,

Vector of Number f, Integer k) -> Bag of Object

as foreign 'xtree_knn_search_fn';


create function xtree_similarity_search_fn(Integer xtId, Vector of
Number f, Real distance)

-> Bag of Object as foreign 'xtree_similarity_search_fn';


/*Internally manage list ID of Xtree*/

create function xtreeIds()-> Bag of Integer as stored;

create function lastest_xtreeIds() -> Integer l

as

begin

    if (notany(xtreeIds())) then  set l= 0

    else

          set l = maxagg(xtreeIds());

    result  l;

end;
```

- **Implementation of user interfaces.**

```
/*-------------------------------------------------------------
```

```
Implementation of user interfaces
------------------------------------------------------------*/
/*Create an index tree*/
create function createIndex() -> Integer p
as
begin
     declare Integer id;
     set id = lastest_xtreeIds() + 1;
     if xtree_make(id) then
     begin
          add xtreeIds() = id;
          result id;
     end;
end;


/*Define function to insert feature vector f and corresponding
object o in feature index */
create function addFeatureIndex(Integer xtId, Vector of Real f,
Object o) ->Boolean as xtree_put(xtId , f, o);


/*Define function to delete feature of an object from index*/
create function deleteFeatureIndex(Integer xtId,  Object o)
->Boolean as xtree_delete(xtId , o);


/*Similar search*/
create function similar_search(Integer xId, Object o, Real
distance) -> Bag of Object
as xtree_similarity_search_fn(xtId, features(o), distance);


/*K nearest neighbor search*/
create function knearest_search(Integer xtId, Object o, Integer k)
```

```
-> Bag of Object
as xtree_knn_search_fn(xtId, features(o), k);
```

Appendix B: The Photo-Album database

❖ Definitions of schema in AmosQL

Definition of data type *Album* and its properties as functions:

```
create type Album;
create function title(Album al)-> Charstring as stored;
```

- Definition of data type Picture and its associated functions

```
create type Picture;
/* feature vector of a picture stored in database*/
create function features(Picture p)->Vector of Number fv as
stored;


/* assume each image has a filename to disk location*/
create function filename(Picture p)->Charstring fn as stored;


/* assume each image has a caption*/
create function caption(Picture p)->Charstring cap key as stored;


/* each picture belongs to an album*/
create function album(Picture p)->Album a as stored;


/* create a new album */
create function newAlbum(Charstring t) -> Album al
as
begin
    create Album(title) instances al (t);
    result al;
end;
```

❖ Indexing pictures

- There is a picture index to store the identifier of the index tree associated with data field *feature*.

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

set picture_index() = createIndex();
```

- Extracting color histogram of a given image:

```
create function colorHistogramExtractor(Charstring filename)

-> Vector of Real as foreign "colorHistogramExtractor
featureExtractor";
```

The real implementation of *colorHistogramExtractor* is in Java. It first reads the image given the filename to memory and then calculates the distribution of colors. In this function the HUE color, which is the pure spectrum color, is used instead Red, Green, and Blue color model. HUE is said to give more accuracy in comparing the distribution of colors in an image [6]

The result is a feature vector of dimension 20.

```
// ------------------------------------------------
// Read image file into the BufferedImage
// ------------------------------------------------
BufferedImage image = ImageIO.read(new File(m_filename));


// ------------------------------------------------
// Grab pixels from the image
// ------------------------------------------------
int height = image.getHeight();
int width = image.getWidth();
PixelGrabber pGrab =
    new PixelGrabber(image, 0, 0, width, height, true);
pGrab.startGrabbing();
```

```
int [] pixels = new int [height* width];
pixels = (int[]) pGrab.getPixels();

// ------------------------------------------------
// Calculate the distribution of colors
// ------------------------------------------------
Color col = null;
int   hue = 0;
for (int i = 0; i < (height * width); i++) {
     col = new Color(pixels[i]);
     hue = rgb_to_hsv(col.getRed(), col.getGreen(), col.getBlue());
     m_histogram[hue / m_bins] ++;
}
. . .
```

- Inserting a new picture

```
create function newPicture(Album al, Vector im)-> Picture p
as begin
     create Picture(filename,  caption , album) instances p
          (im [0], im[1], al );
     set features(p) = colorHistogramExtractor(filename(p));
     addFeatureIndex(picture_index(), features(p), p);
     result p;
 end;
```

- **KNN queries with index**

```
/*KNN search*/
create function knearest(Picture p, Integer k) -> Bag of Picture
as knearest_search(picture_index(), p, k);


/*Example of KNN search*/
create function q1(Charstring caption, Integer k) -> Bag of
Charstring
as  select caption(p)
     from Picture p, Picture q
     where p in knearest(q,k) and caption(q)= caption;
```

- **KNN queries without index**

```
/* Basic functions for kNN */
create function kct() -> <Vector of Real x, Picture pic>
as   select cc, p from Picture p, vector of real cc
     where cc = features(p);


/* Calculate distances*/
create function distances(Vector of Real x,
     Bag of <Vector of Real, Picture > P)->
     Bag of <Real distance, Picture pic>
as
     select euclid(x, v), pi
     from Vector of Real v, Picture pi
     where <v, pi> in P
;
/*KNN queries without index*/
```

```
create function q4(Charstring caption, Integer k) ->
    Bag of Charstring
as  select caption(p1) from Picture p1, Picture p2, Real dis
    where
          caption(p2) = caption
          and <dis, p1> in
          leastk(distances(features(p2), kct() ), k);
```

- **KNN queries with and without index in related to the measurements**

A KNN query without index calls the function *distances* to calculate the distances from the features of the given picture to features of all other pictures in the database. It will not scale when the number of pictures increases (meaning the increase of database size) or when the feature vector of picture is complex (meaning the increase of dimension of feature vector). In these cases, the function *distances* take times to perform the distance calculation to each feature of other pictures. It results in the linear increase of the search time (see Figure 9 Query time of KNN queries (using and not using index) and Figure 11 Query time of KNN queries (with and without index) on 100000 objects).

On another hand, the query with index calls function *knearest* that takes advantage of the Xtree index structure (hierarchal structure of MBRs). It does not perform the distance calculation to all other pictures. Instead, it always selects pictures whose distances are minimal distances to the query picture. Therefore, it can find K nearest neighbor faster without visiting all others. Technically, KNN search on index tree avoids visiting unnecessary nodes by choosing a node who's MBR has a minimal distance to the query point to visit first. As expected with the increase of database size and the increase of dimension, KNN search with index takes less time than KNN search without index (see Figure 9 Query time of KNN queries (using and not using index) and Figure 11 Query time of KNN queries (with and without index) on 100000 objects).

Appendix C: Saving and restoring index files

❖ Saving index files

The name of an xtree index file is constructed from the ID of the Xtree and the name of the database image by the function *gen_indexfile*:

```
create function gen_indexfile(Charstring image, Integer id) ->
Charstring as
begin
     declare Charstring filename, Charstring indexfile;
     /*suppose image = foo.dmp. The index filename = foo1.xt*/
     set filename = substring(image, 0, char_length(image) -
                              5);
     set indexfile = filename + stringify(id) + ".xt";
     result indexfile;
end;
```

When the *save* command is issued, the *index_rolloutfn* is used to save index trees. All index trees are saved to disk.

```
create function index_rolloutfn(Charstring image) -> Boolean
as
begin
     for each Integer id where id in xtreeids()
          xtree_save(id, gen_indexfile(image, id));
     result TRUE;
end;
```

In the function *index_rolloutfn,* a function *xtree_save* is called to save each index tree. The function *xtree_save* is defined as

```
create function xtree_save(Integer xtId, Charstring filename)
-> Boolean as foreign function ‘xtree_save’
```

The internal function *xtree_save* takes index tree's ID and a chosen name. It then saves the index tree from main memory to the disk. TRUE is returned if no error occurs, otherwise FALSE.

- **Restoring index files**

After the initialization, Amos II calls *index_rollinfn* to load all index files associated with the database image. The function *index_rollinfn* loops through a list of Xtree's IDs and calls the foreign function *xtree_load* for each ID of them to load the corresponding index file if the index file exists on disk.

```
create function index_rollinfn(Charstring image) -> Boolean as
begin
    declare Charstring indexfile;
    for each Integer id where id in xtreeids()
    begin
        set indexfile = gen_indexfile(image, id);
        if file_exists(indexfile) then
        xtree_load(indexfile);
    end;
    result FALSE;
end;
```

And the *xtree_load* is defined as follow

```
create function xtree_load(Charstring filename) -> Integer xtId as
foreign function 'xtree_load'
```

The internal function *xtree_load* takes the file name of an index file as input parameter. It loads the intdex tree from the file and returns the ID of the loaded index tree.

Finally, saving and restoring functions are registered to Amos II. They are automatically called from Amos II when Amos II saves or the database is initialized respectively.

```
register_saver(#'index_rolloutfn', #'index_rollinfn');
```