ReadMe of the Code of GiMP

Beta version 1.1

Rui Zhang

rui@csse.unimelb.edu.au

http://www.csse.unimelb.edu.au/∼rui

Department of Computer Science
and Software Engineering,
The University of Melbourne,
Carlton, Victoria, Australia, 3053
2006

1 Paper related to this code

Rui Zhang, Panos Kalnis, Beng Chin Ooi, Kian-Lee Tan: **Generalized Multi-dimensional Data Mapping and Query Processing**, *ACM Transactions on Data Base Systems (TODS)*, 30(3): 661-697, 2005.

2 Copyright

3 Implementation Notes

The code of the GiMP is based on a B^+-tree coded by Dai Haoyu, modified by Rui Zhang. The delete operations are coded by Panos Kalnis. Anything else such as the GiMP operations, customization functions, supportive functions of different mappings, etc, are coded by Rui Zhang.

4 How to Use the Code

4.1 Compilation

The code was compiled on Fedora 2 and can be compiled on most Linux and Unix systems. Use the command "make" to compile.

4.2 Follow the following steps to run the code

1. **The mapping method:** The code already implemented 5 mapping methods: the B^+-tree, the iMinMax, the UB-tree, the Pyramid technique and the iDistance. Implement one of them by defining one of the 5 constants BTREE_ , IMINMAX_ , UBTREE_ , PYRAMID_ and IDISTANCE_ in the file "customize.h".

2. **Dimensionality:** Define the dimensionality of the data space by defining the constant D in the file "btree.h". E.g. "#define D 4" means dimensionality is 4. For the B^+-tree implementation, D must be set as 1.

3. **Data domain and type:** The code assumes that data are of type float, and normalized to $[0,1]^d$ (even for the UB-tree). The data type is defined by the constant DATA_TYPE in file "btree.h". E.g. "typedef float DATA_TYPE;" means the data are of type float. The current implementation of the Z-value calculation, that is, the ZV() function defined in the file "support.c", inputs a data point of type float and returns its Z-value. However, other ways of defining the ZV() function can be used to input other types of inputs such as integers. If the data inputs are integers, DATA_TYPE should be defined as integer.

4. **Key type:** The type of the key is defined by the constant KEYTYPE in the file "btree.h". E.g. "typedef float KEYTYPE;" means the key is of type float. For the UB-tree implementation, KEYTYPE should be defined as unsigned long (or unsigned long long if needed) no matter what type the input is. For other mapping methods, KEYTYPE should be defined as float (or double).

5. **Ad hoc settings:** If the GiMP is implemented as the UB-tree, the order of the Z-curve is defined by the constant ORDER in the file "support.h". E.g. "#define ORDER 4" means the order of the Z-curve is 4. The data type of the Z-value is defined by the constant "Z_TYPE" in the file "support.h". E.g. "typedef unsigned long Z_TYPE;" means a Z-value is represented by an unsigned long integer. Z_TYPE should be defined as the same type as KEYTYPE.

   If the GiMP is implemented as the iDistance, the number of reference points is defined by the constant NumRef in the file "customize.h". E.g. "#define NumRef 64" means the number of reference points is 64. The reference points used for the iDistance are stored sequentially in the file "reference" in text format.

6. **Data and Queries:** In the test program (the main() function in the file "gimp.c"), both data and queries are read from files. The data and query files are referred by the pointers "fp_data" and "fp_query", respectively. Both data points and queries are stored sequentially one by one in binary format. A data point or a kNN query is stored one dimension by one dimension. A range query is stored in the format of $\{l_1, u_1, l_2, u_2, ..., l_d, u_d\}$, where l_i and u_i are the lower and upper bounds of the query range in dimension i. For convenience, we have used the pointer "fp_knnquery" to refer to the kNN query file, while "fp_query" is used to refer to the range query file. By default, the names of the data file, the range query file and the kNN query file are "data", "rangequery" and "knnquery", respectively.

7. **Query parameters:** Two parameters can be defined in the file "gimp.c": "numquery" and "k". numquery specifies how many queries to execute. k specifies the number of NN to find in kNN search. It's easy to change the code to get these parameters from inputs.

8. **Compile the code:** After the above settings, compile the code by "make". The executable "gp" is generated.

9. **Run the executable:** There are three choices to run "gp": "gp b", "gp r", or "gp n". "gp b" builds an index from the data file. "gp r" runs the range queries from the range query file. "gp n" runs the kNN queries from the kNN query file. Note the type of queries each mapping method supports. Asking the wrong query type would generate errors.

10. **Results:** Results are written to the file "result" in text format. ID's of the answer points are listed for each query.

5 Optimization Notes

1. Internal nodes of the B^+-tree are loaded into memory before the queries. The internal nodes are less than 0.3% of the whole index, which can fit into memory. To test the effect of buffer, code needs to be touched up.

2. Nodes of the B^+-tree can be compacted by the function compact( ) to a higher node utilization rate, which is set by the constant "compactrate" in file "btree.h". E.g. "#define compactrate 0.9" means the utilization rate is 0.9. Another way to achieve high utilization is by using a B*-tree style insert/delete, which is not implemented in this B^+-tree yet. If compactrate is set less than 0, no compaction would be conducted.

3. For kNN search, the initial search radius and the increasing amount of the search radius are set by two constants "r0" and "dr" in the file "btree.h". By default, they have been set as 0.1 and 0.005 respectively, although r0 can be optimized by estimating the final search radius and dr can be set accordingly, such as 5% of r0.