Chapter 74
The Research of Embedded Database Hybrid Indexing Mechanism Based on Dynamic Hashing

Huijie Chen and Jianwei Li

Abstract For the sake of improving the retrieval and insertion efficiency of embedded database when dealing with large-scale data, this paper was proposed a hybrid indexing structure based on dynamic hashing and designed a hybrid indexing mechanism combined with the characteristic of extendible hashing, linear hashing and red black tree. After retrieving and inserting large amounts of data, the results show that this hybrid indexing mechanism can make the time complexity of embedded database retrieval and update operation almost to be O(1). At last, the hybrid index mechanism provides for embedded database with real-time index insertion and record querying.

Keywords Embedded database · Hybrid indexing mechanism · Dynamic hashing

74.1 Introduction

With the developing of information and electronic technology recent years, the increasing sharply of information that need to manage by the embedded systems prompted the embedded device requires have more effective data management capabilities. The combination of embedded systems and database technology can greatly enhance the data management capability of embedded devices. Besides, the

H. Chen · J. Li (✉)
School of Computer Science and Technology,
Taiyuan University of Science and Technology,
Taiyuan 030024, China
e-mail: ghhong2004@163.com

H. Chen
e-mail: chenhuijie666@163.com

W. Lu et al. (eds.), *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*, Lecture Notes in Electrical Engineering 211,
DOI: 10.1007/978-3-642-34522-7_74,

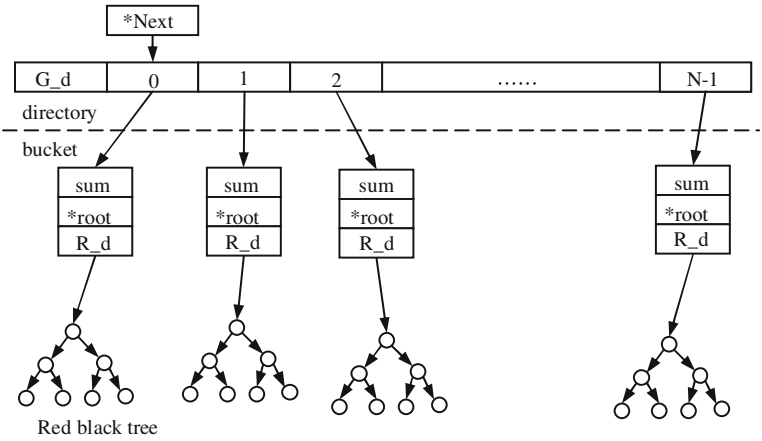

Fig. 74.1 Hybrid indexing structure

index mechanism is an important part of database system and determine largely the database operate performance. Therefore, the design of an efficient indexing mechanism used in embedded environments is essential.

However, most of the existing embedded database may make some real-time operation cannot be guaranteed. It is because of that the embedded database uses the dynamic growth tree structure as their index structure.

In order to solve the problem above mentioned, many hybrid indexing structures, such as Hybrid-TH tree [1], HT trees [2], H-Tail [3] and H-T*tail [4], was designed out. These hybrid index structures are divided into two parts, the upper is the static hash structure, and the lower part of it is the tree structure. However, with the amount of data increasing, this hybrid index structure needs to deepen the tree depth to accommodate the more data. Therefore, the data retrieval and insertion efficiency also will be further decreased.

This paper will present a new dynamic hashing splitting methods taking account of the characteristic of extendible hashing [5] and linear hashing [6], then design a hybrid indexing mechanism combined with red black tree [7]. This mechanism can solve the problem of the embedded database processing efficiency is continuing decrease when dealing with massive data.

74.2 Hybrid Indexing Structure

Hybrid indexing structure is shown in Fig. 74.1.

1. Directory. Figure 74.1 shows that the upper part of the hybrid indexing structure is directory. The directory consists of some directory entries. Directory entry numbers vary from 0 to N−1. The directory entry stored the corresponding bucket addresses. G_d [8] is the global depth of directory that

means that the number of the current directory need at least G_d bit to be coded. Next [9] is a pointer that pointing the directory entry that will be split. The changes rules are as follows:

$$Next = \begin{cases} Next++ & Next! = 2^{G\_d} - 1 \\ 0 & Next == 2^{G\_d} - 1 \end{cases} \tag{74.1}$$

2. Bucket. Figure 74.1 shows that the lower part of the hybrid indexing structure is bucket. The bucket is composed by the total of keyword stored in this buckets, the local depth of this bucket and the pointer to the red black tree root nodes. The local depth [10] means that the trailing bit of every keyword stored in the buckets are same. The relationship between the local depth B_d and global depth is $0 < B\_d \leq G\_d$. If two directories entry are point to one bucket at the same time, the bucket is called with shared bucket.

74.3 Hybrid Indexing Mechanism

The explanation of symbols:
N: the total number of directory entries in the directory.
H(x): the hashing function, input x and produce a integer.

- Search Operation

1. K: get the keyword of record
2. $M = \lfloor H(k) \rfloor_{G\_d}$. $M = H(k)\,\%2^{G\_d}$
3. get the directory entry number B(K) by the following formula:

$$B(K) = \begin{cases} \lfloor H(K) \rfloor_{(G\_d)} & (M < N) \\ \lfloor H(K) \rfloor_{(G\_d-1)} & (M \geq N) \end{cases} \tag{74.2}$$

4. Search the nodes of keyword is K in the bucket that id = B(K).

- Insert Operation

1. K: get the keyword of record
2. $M = \lfloor H(k) \rfloor_{G\_d}$. $M = H(k)\,\%2^{G\_d}$
3. get the directory entry number B(K) by the formula (74.2).
4. Insert the record into the bucket that id = B(K). At last check whether the barrel overflow. If do not meet the split conditions, this operation is over, otherwise, do the split operation.

- Split Operation

Get the local depth B_d of the bucket the corresponding directory entry M that need to split.

1. If the bucket that need to split is not shared bucket, B_d = = G_d
   - j = G_d; G_d ++; the global depth should plus 1.
   - Get the directory entry number that the brother bucket of M. B_M = M|(1 < < j)
   - Expand the directory entry number to B_M.
   - Split the nodes of bucket to M and B_M bucket.
   - The local depth of bucket M and brother bucket B_M should plus 1.
   - The directory entry from N to B_M-1 should pointer to their brother bucket.
2. If the bucket that need to split is shared bucket, B_d < G_d
   - if G_d = B_d + 1
     (a) The smallest directory entry number pointer to bucket M is mini_M = M & ((1 < < B_D) − 1), and its brother bucket is max_M = mini_M|(1 < < B_d)
     (b) If max_M < N, it means the bucket max_M have already in the directory. The bucket M should split the bucket mini_M and bucket max_M. The B_d of mini_M and max_M should plus 1.
     (c) If max_M ≥ N, it means the bucket max_M is not in the directory. Then expand the directory entry number to max_M. The bucket M should split the bucket mini_M and bucket max_M. The depth of both of the bucket should plus 1. The directory entry from B to B_M - 1 should pointer to their brother bucket.
   - if G_d > B_d + 1
     (a) The smallest directory entry number pointer to bucket M is mini_M = M & ((1 < < B_D) − 1), and its brother bucket is max_M = mini_M|(1 < < B_d).
     (b) The bucket M should split the bucket mini_M and bucketmax_M. The depth of both of the bucket should plus 1. The directory entry from B to B_M − 1 should pointer to their brother bucket.

74.4 Simulation

74.4.1 Search Operation

The retrieval operation of the hybrid indexing mechanism is composed by hash operation and search operation in bucket. For the part of Hash, the time-consuming by calculating the location of the bucket can be negligible. These sections mainly discuss the average time trend of search operation in bucket. The time trend of average retrieval shown in Fig. 74.2.

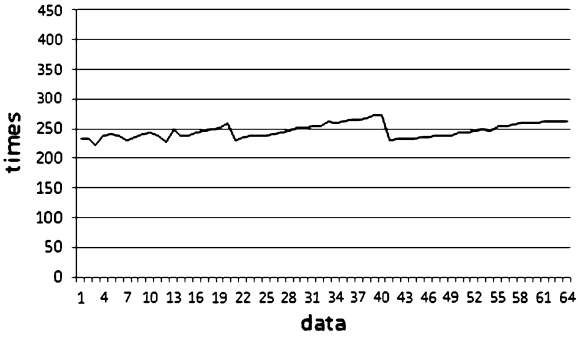

Fig. 74.2 The time trend of average retrieval time

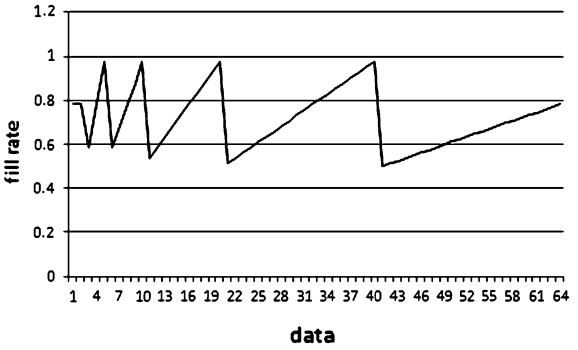

Fig. 74.3 The trend of fill rate

Overall, with the amount of data increasing, the average retrieval time trend tends to parallel. And some periodic fluctuations can see in the figure. The reasons for the retrieval time tends to parallel depends mainly on the tree depth of the red–black tree structure in the lower part of the hybrid index structure. And the bucket depth is strictly limited. For some fluctuations in the growth trend of the retrieval time, it is mainly affected by the bucket fill rate in hybrid index structure. The trend of fill rate is shown in Fig. 74.3 and the fill rate is calculated as follows:

$$\text{fill rate} = \frac{\text{the sum of data}}{\text{the number of bucket} \times \text{bucket volume}} \tag{74.3}$$

74.4.2 Insert Operation

The Insertion operation of the hybrid indexing mechanism contains Hashing, the operation of insert nodes into the bucket and adjusting. For the part of Hash, the time-consuming by calculating the location of the bucket can be negligible. These sections mainly discuss the average time trend of inserting nodes into the bucket and adjusting.

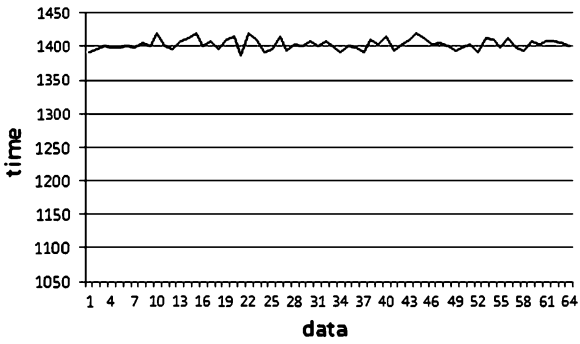

Fig. 74.4 The average insert time if the bucket depth have a limit

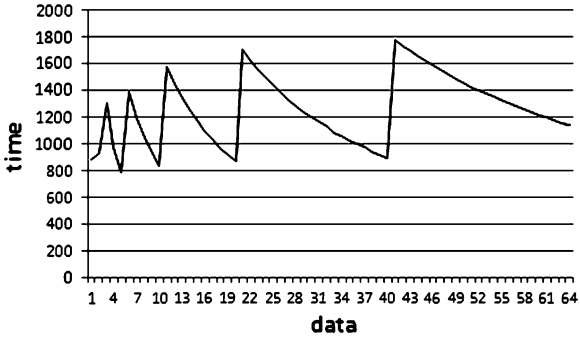

Fig. 74.5 The trend of average adjustment time

74.4.2.1 The Operation of Insert Nodes into the Bucket

The Insert operation efficiency will be decreased with the tree depth to be deeper. However, if the tree depths have an upper limit, the insert operation efficient will not increase. Figure 74.4 shows the average insert time if the bucket depth have a limit.

74.4.2.2 Adjusting

The adjustment process is composed by the directory expansion and the bucket split. If a bucket is overfilled, it needs to split into two partially filled buckets. The average adjustment time is shown in Fig. 74.5. The average adjusting time is calculated as follows:

$$\text{average adjustment time} = \frac{\text{the total of adjustment time}}{\text{the total of data}} \tag{74.4}$$

The trend of average inserts time shown in Fig. 74.6. The average insertion time is calculated as follows:

$$\begin{aligned}\text{the average insertion time} = {} & \text{the average insert a node time} \\ & + \text{the average adjustment time}\end{aligned} \tag{74.5}$$

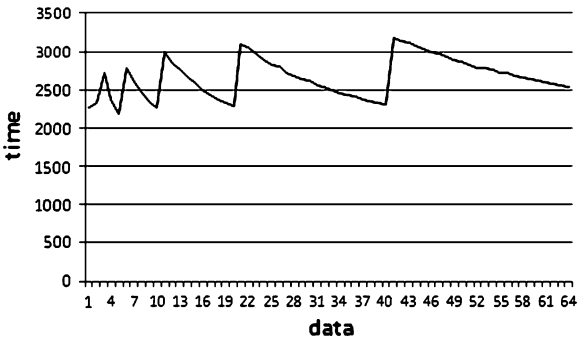

Fig. 74.6 The trend of average insert time

74.5 Conclusions

The experiment shows that the search operation complexity of hybrid indexing mechanism is O(1). The insert operation is affected by the fill rate of bucket and show a periodic fluctuations trend. And the time of insert operation have an upper limit. However, the drawback is that in the normal data insertion process, there will be a certain period of adjustment operations, the insert operations cannot continue during this time. It has to need use other techniques to improve.

Acknowledgments This work is under the support of the "Shanxi Province nature Foundation (No. 2012011027-3)", authors hereby thank them for the financial supports.

References

1. Ryu C, Song E, Jun B, Kim Y-K, Jin S (1998) Hybrid-TH: a hybrid access mechanism for real time main-memory resident database systems. In: Real-time computing system and applications 1998 proceeding, fifth international conference on 1998, pp 303–310
2. Jiali X (2003) H-T: an index mechanism for real-time main memory database systems. Comput Appl Softw 20(6):61–63 (in Chinese)
3. Lin P, Li H, Xu X (2004) Optimization of T-tree index of main memory database in critical application. Comp Eng 30(17):75–76 (in Chinese)
4. Chen J, Zhu W (2009) A new index mechanism fitted for the embedded database. Microcomput inform 25(3–2):84–86 (in Chinese)
5. Fagin R, Nievergelt J (1979) Extendible hashing—a fast access method for dynamic files. ACM Trans Database Syst 4(3):315–344
6. Litwin W (1980) Linear hashing: a new tool for file and table addressing. In: Proceedings of the 6th international conference on very large data bases, Montreal, pp 212–223
7. Hinze R (1999a) Constructing red-black trees. In: Workshop on algorithmic aspects of advanced programming languages, 89–99
8. Rammohanrao K, Lloyd JK (1982) Dynamic hashing schemes. Comput J 25 478–485
9. Rathi A, Lu H (1991) Performance comparison of extendible hashing and linear hashing techniques. ACM Press New York. Doi:10.1145/122045.122048.19-26
10. Larson P (1985) Performance analysis of a single-file version of linear hashing. Comput J 28:319–326