21st International Conference on Database
and Expert Systems Applications (DEXA'10),
Bilbao, Spain, 30 August - 3 September 2010

Hybrid In-memory and On-disk Tables for Speeding-up Table Accesses

Joan Guisado-Gámez[1], Antoni Wolski[2], Calisto Zuzarte[3], Josep-Lluís Larriba-Pey[1], and Victor Muntés-Mulero[1]

[1] DAMA-UPC, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Campus Nord-UPC, 08034 Barcelona
{joan, larri, vmuntes}@ac.upc.edu
[2] IBM Helsinki Laboratory, Finland
antoni.wolski@fi.ibm.com
[3] IBM Toronto Laboratory, Markham, Ontario, Canada L6G 1C7
calisto@ca.ibm.com

Abstract. Main memory database management systems have become essential for response-time-bounded applications, such as those in telecommunications systems or Internet, where users frequently access a table in order to get information or check whether an element exists, and require the response to be as fast as possible. Continuous data growth is making it unaffordable to keep entire relations in memory and some commercial applications provide two different engines to handle data in-memory and on-disk separately. However, these systems assign each table to one of these engines, forcing large relations to be kept on secondary storage.
In this paper we present TwinS — a hybrid database management system that allows managing hybrid tables, i.e. tables partially managed by both engines. We show that we can reduce response time when accessing a large table in the database. All our experiments have been run on a dual-engine DBMS: IBM®SolidDB®.

Keywords: Hybrid tables, Main memory databases, DBMS performance.

1 Introduction

As a result of cyberinfrastructure advances, the vast amount of data collected on human beings and organizations and the need for fast massive access by a large amount of users to these data pose a serious performance challenge for database management systems. For example, large *Domain Name Systems* (DNSs), that are used for translating domain names to IPs, may be constantly queried by a large amount of users per second that require the response to be as fast as possible even in peak situations where the amount of users increases. Efficiently detecting whether a domain exists or not must meet real-time requirements since this does not only speed up this query answer, but it also reduces the load of the system, accelerating other concurrent queries.

Queries in this type of applications are usually characterized for accessing data depending on a certain value of one of its attributes which are typically unique, such as the string containing the domain name in our examples. Also, it is very usual to find queries on data which do not exist in the database. For instance, domains that are not associated to any existent IP are very common. Looking for data which are not present in the database affects the overall capacity of the system to respond as fast as possible. In this situation, fast database management solutions such as main-memory database management systems (MMDBMSs) become essential.

Although MMDBMSs are efficient in terms of accessing or modifying data, they limit the total amount of data to the available memory. Some commercial MMDBMSs like IBM®SolidDB® or Altibase™ resort to a hybrid solution implementing a second storage based on disk. This requirement causes large tables to be necessarily classified as on-disk, not allowing them to benefit from main memory techniques. This restriction directly colides with the fact that the size of large databases is in the petabytes nowadays and, therefore, it is very common to find massive tables that do not fit in memory entirely. This situation demands for a coupled solution where tables that do not entirely fit in memory may partially benefit from MMDBMS advantages.

It is important to take into account that classifying a table as on-disk does not strictly mean that the whole table is on disk, since buffer pool techniques may be used to keep the most frequently accessed information in memory. However, using efficient memory structures as those in MMDBMSs is preferable to the use of buffer pool techniques in disk-resident DBMSs, as discussed in [1], since the latter might not take full advantage of the memory.

In this paper, we propose TwinS — a hybrid in-memory/on-disk database management system that allows managing *hybrid tables*, i.e. the same table is split into two parts kept on disk and in memory, respectively. This poses a challenge when it comes to decide the existence and location of a specific row, when performing an access by using the value of a unique attribute. Our main contributions are: (i) proposing a new architecture that allows for hybrid tables, (ii) providing an efficient data structure for avoiding useless accesses to both memory and disk, specially when the queried information is not in the database, (iii) comparing our approach to a real commercial MMDBMS, IBM®SolidDB®, that provides both an in-memory and a disk-resident engine. Our results show that, thanks to our approach, we are able to speed up the overall execution when accessing rows in the table intensively. This provides us with experimental evidence that allows us to state that using in-memory data structures outperforms classical buffer pool techniques for on-disk DBMS.

The remainder of this document is organized as follows. Section 2 describes the architecture proposed for our hybrid database and our proposal in order to reduce unnecessary accesses to both memory and disk. In Section 3, we present our results. Section 4 presents some related work. Finally, in Section 5, we draw the conclusions and present some future work.

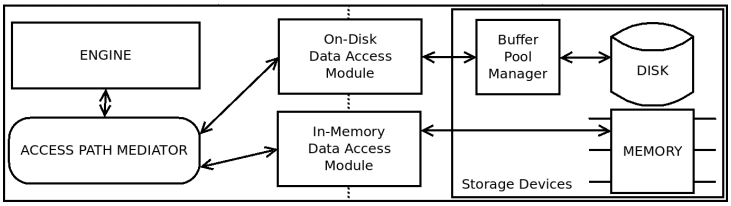

Fig. 1. TwinS architecture.

2 TwinS

TwinS is a hybrid database management system that aims at improving the performance of disk resident databases by enabling hybrid tables which store part of their rows in memory. This architecture naturally requires the coexistence of two modules that separately manage the accesses to each of the storages. Let $R = \{r_1, r_2, ..., r_n\}$ be a hybrid table of n rows. In TwinS, we assume that a hybrid table R is divided into two parts, R_M and R_D, such that $R = R_M \cup R_D$ and $R_M \cap R_D = \emptyset$, where R_M is the part of the table managed by the in-memory engine, and R_D is the disk-resident part.

Figure 1 shows the architecture of our system with the modules involved in the execution of a query in TwinS, which is basically divided into two parts: the *engine* and the *access path mediator* (APM). The APM is in charge of solving data access paths and it takes into account that the location of data in TwinS may be distributed among two storage devices. Depending on the data location information in the APM, it redirects the query to the in-memory or on-disk data access modules used to access the rows stored in main memory or on disk, respectively. One of the main contributions of this paper is to design the APM such that it keeps information in a compact way and maximizes the efficiency of the system by avoiding unnecessary accesses to both memory and disk.

The APM has to have information of data location at row level. We do not discuss in this paper about sequential accesses because using hybrid tables instead of in-memory or on-disk tables does not add any complexity in this operation, since it would imply traversing data in both memory and disk sequentially and, therefore, the APM would not be strictly necessary.

2.1 A first naive approach: Latency Priority (LP)

In order to start the discussion, we present a first approach that takes the latency of each storage device as the only information to decide the storage to be accessed, prioritizing the one with the lowest latency, i.e. memory. We call this approach *Latency Priority* (LP). Thus, a naive solution to solve value-based queries on a hybrid table R would be trying to find data in the in-memory storage first and, if not found, trying to find it on disk. This solution forces us to access memory at least once for each value and, if the value is not found, it also requires accessing disk.

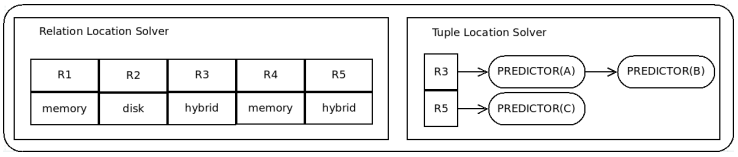

Fig. 2. Access Path Mediator design.

The LP APM has a data structure called Relation Location Solver (RLS), which classifies each table in the system as in-memory, on-disk or hybrid.

2.2 Prediction Based approach (PB)

Now, we propose a refined design of the APM which is depicted in Fig. 2. This version has two components: the Relation Location Solver (RLS)(left side of Fig. 2), and the Tuple Location Solver (TLS)(right side of Fig. 2), which is used to avoid useless accesses.

The main difference between LP and PB is that the latter incorporates a new type of structure that allows the system to know whether a row is in a specific storage or not. We call this new data structure *predictor*. The TLS is a structure in the form of a vector where each position points to a list of predictors. Predictors allow to know whether a row can be in memory or on disk depending on the values of its attributes. Thus, each hybrid table may have as many predictors as attributes in the table. Note that in Fig. 2, R3, identified as a hybrid table by the RLS, has one predictor for attribute A and another one for attribute B, and R5, also identified as a hybrid table, has only one predictor for attribute C. This way, for each queried value on a given attribute, the APM checks in the corresponding predictor (if the attribute has one associated) the location of the matching row.

Algorithm 1 describes the procedure to access a unique attribute. First PB gets the category of table R through the RLS (lines 2-3). If R is classified as a disk-resident or an in-memory table the system scans the table in order to find a certain value (line 2). If R is a hybrid table, the system checks in the TLS if there are rows of R fulfilling the condition related to that *value* in R_M (line 4), R_D (line 6) or in both of them (line 7), and then proceeds scanning the corresponding part. Note that *scan()* is not a sequential operation, but it consists in looking up the data through in-memory or on-disk indexes.

Using Count Filters for Prediction

This section discusses on the implementation of the predictors depicted in Fig. 2. From previous sections, we may infer several aspects that are crucial for predictors:

The access time to the predictor has to be as fast as possible. Therefore, a predictor has to be in memory. This also implies that it has to be as compact as possible in order not to reduce significantly the amount of memory needed to

Algorithm 1: Access by value

Input : Value '*value*' of the attribute '*Attribute*', Relation '*R*'
Output: *L*::List of rows of *R* that *R.Atr* = *val*

```
1  switch APM.checkRLS(R) do
2      case DISK or MEMORY: L ←Scan(R,Attribute,value);
3      case HYBRID:
4          switch APM.checkTLS(R,Attribute,value) do
5              case MEMORY: L ←Scan(R_M,Attribute,value);
6              case DISK: L ←Scan(R_D,Attribute,value);
7              case BOTH: L ←Scan(R_M ∪ R_D,Attribute,value);
8          endsw
9
10 endsw
```

execute the access operation. However, the accuracy of the predictor is crucial in order to be able to reduce the number of accesses to both memory and disk. Predictors may return false hits, i.e. predict that a row is in a certain storage when it is not true. This will result in an unnecessary access, but will preserve the correctness of results. However, they can never return a false negative [2], since that would imply not accessing the storage containing a row with the requested value. Although it is out of the scope of this paper to analyze the performance aspects regarding insertions, deletions and updates in the database, we need this structure to be ready to be updated if this happens.

The first two requirements are contradictory, since the first requires to keep predictors as small as possible, while the second needs keeping as much information as possible in order to make accurate predictions about the location of rows. A way to reduce the number of useless accesses is by implementing the predictor using two presence bitmaps in order to mark which values have been inserted in memory and on disk, respectively. A bit set to 1 in one of the bitmaps indicates that the value represented by that unique bit exists in the associated storage device. Using two bitmaps with as many positions as the size of the domain of the attribute that is linked to that predictor would allow us to have an exact predictor. With this structure we guarantee the four conditions stated above. However, in the presence of non-contiguous values or when the databases are updated very frequently, they would be very inefficient. The use of Bloom Filters [2] instead of bitmaps saves memory by means of applying hash functions at the cost of losing exact answers. The precision of a Bloom Filter depends on two different variables: the number of hash functions that are applied for each key and the size of the Bloom Filter. However, we would still have a problem when removing a value: we could not be sure that setting its corresponding bit to zero is correct, since other values could be mapped to the same bit. This is solved by using count filters. In order to implement the predictor we use Partitioned Dynamic Count Filters (PDCF) [3], a compact data structure that is able to keep approximate counters for large sets of values. As we see later, this structure requires little memory since it dynamically allocates and deallocates memory for the counters that need it. In our case, each predictor is composed by two PDCF, one for memory and another one for disk. Counters in the PDCF are

assigned to values using a hash function. Therefore, each time a value is inserted in memory the corresponding counter (or counters) in the PDCF assigned to memory is increased. When a value is removed the counter is decreased. Note that only when the counter is equal to zero, we are sure that the corresponding value is not in memory. Of course, the PDCF might return a false positive when more than one value is mapped on the same counter, but it will never return a false negative. It is also possible to allocate a single PDCF. For instance, if we wanted to avoid useless disk accesses, but we were not worried about memory accesses because it was the case that memory is very much faster than disk, we could consider creating a single PDCF. In general, it is worth to create both, since the space needed by these structures is not significant compared to the size of the database.

3 Experimental Results

This section analyses the effect of using hybrid tables. First, we start with the performance achieved by TwinS when accessing a table randomly by its primary key, using LP and PB with one or two predictors. Second, we analyze the impact of querying values which are not in the database.

Data have been generated with *DBGEN*, the database generator provided by TPC-H. We run the experiments on the TPC-H table *orders*. The size of the table after loading 10^6 rows into memory using IBM SolidDB is 550 MB. We test TwinS varying the amount of available memory, the percentage of the memory used for the in-memory engine and the TLS probability of false hit. Unless explicitely stated, we consider for all the experiments three different values for the amount of available memory: 100, 200 and 400 MB. For each of these three cases, we vary the portion of memory used by the in-memory engine (20%, 40%, 80% and 100%). Note that the remainder of the memory (0%, 20%, 60%, 80%) is used for the buffer pool. This will allow us to understand whether it is more useful to use memory to keep part of a table using in-memory data structures, or it is better to use it as a classical buffer pool. Also, we test several configurations of the TLS, with a probability of yielding a false hit of 0.001, 0.01 and 0.1.

Each experiment is run three times, all the values shown in this section are the average of the results of these executions. When stated, the buffer pool is warmed up. The experiments are run using an Intel® Core™ 2 Duo at 3.16GHz. The memory available by the computer is 3.5 GB. However, taking into account that during the execution of a query the scan operation may coexist with other operations in the QEP or even with concurrent queries from other users, we assign a maximum of 400 MB to table orders.

We test our architecture when accessing data at random using a unique attribute. Figure 3 shows the access time in seconds when using LP and PB with both one predictor and two predictors. With this experiment, we first aim at understanding the effect of the accuracy of predictors on the overall access time. Because of this, we have reserved an extra space of memory for predictors

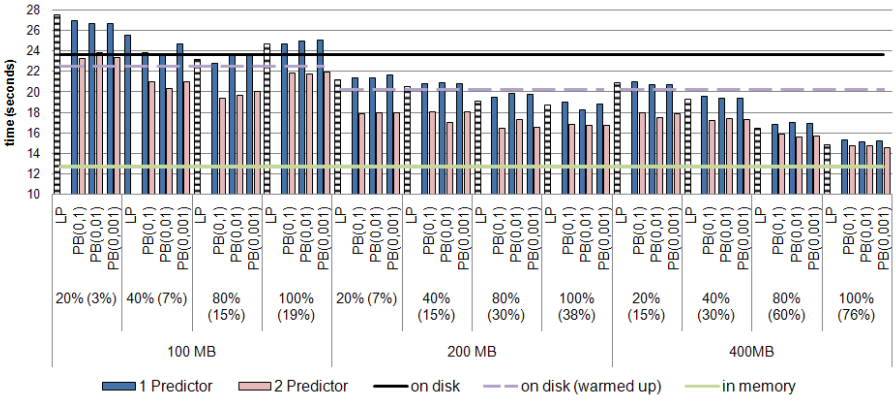

Fig. 3. Access time when random access by primary key is done.

which is not competing with the memory pool reserved for keeping data. The percentage of tuples that fit in the in-memory engine appears in brackets next to the percentage of memory used for the in-memory engine. It also shows the access time when the system has very little memory and almost the whole table is on disk, the time when the table is on disk, but we have previously warmed-up the buffer pool, and the access time when the whole table fits in memory.

As we can see from these results, LP is able to achieve better results, when the amount of memory available is large and allows to keep a large percentage of the table in memory, as opposed to the case where the table is completely on disk and even to the case where the table is on disk but we have previously warmed up the buffer pool. Note that warming up the buffer pool might not be realistic in several scenarios, but we have included it to stress our proposals and show that even in this situation, we can benefit from hybrid tables. However, as expected, when the amount of memory is limited, and a large part of the table is on disk, since LP requires accessing memory for any access by value, performance degrades, making it better to use an on-disk DBMS. Regarding PB, we can observe that using one predictor will not save time with respect to LP, since LP saves accessing disk whenever the value is on memory and, therefore, the predictor is useless. Taking this into account, LP is desirable to PB with one predictor. On the contrary, with two predictors, we save unnecessary accesses to disk when the value we are looking for is on disk, and the overall time is reduced with respect to all the other approaches. Another important result is that it is more beneficial to keep data in the in-memory data structures, than using classical buffer pool techniques, agreeing with the ideas presented in [1]. This can be seen clearly in the 200 MB scenario of Fig. 3: when 80% of the memory available is used by the in-memory engine, any configuration with two predictors is able to reduce significantly the access time compared to the reduction that the buffer pool is able to achieve. Finally, we can see that the accuracy of the predictor results in very similar time responses. The proper configuration of the TLS which, as it is shown in Fig. 3, consists in using two predictor with

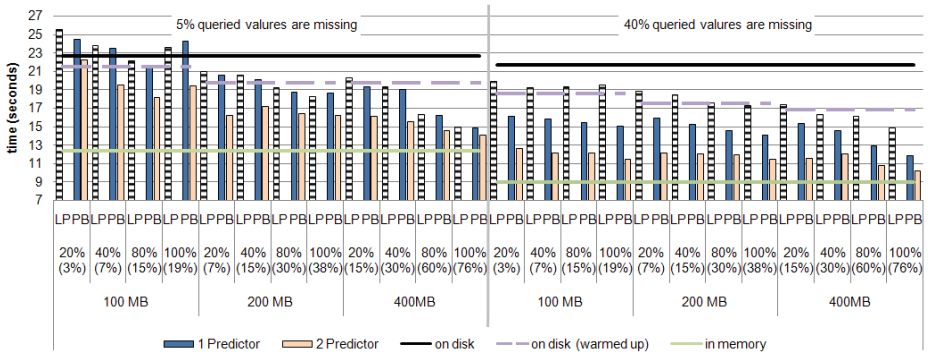

Fig. 4. Left: access time when 5% of the queried values are missing. Right: access time when when 40% of values are missing.

probability of false hit under 0.1, only requires 6-7 MB, which represents 1.3% of the total amount of memory needed to store table (550 MB).

Following, we repeat the same experiments adding a certain number of queries to values that are missing in the database. We run the experiments taking into account two different situations: a) 5% of the queried values are missing (leftmost side in Fig. 4) and b) 40% of the queried values are missing (rightmost side in Fig. 4). Results focus exclusively on the case where predictors have a probability of false hit equal to 0.1 which has been shown to be the best case taking into account both response time and memory requirements.

Again, we show three baseline scenarios: the whole table fits in memory, the whole table is stored on disk, and the table is on disk and the buffer pool has been warmed up. LP is in general better than having the whole table on disk, except when the amount of memory available is not very large. If the buffer pool is warmed up, then buffer pool based tables obtain similar results to those obtained by LP, except if almost the whole table fits in memory in which LP achieves better response times. Regarding PB, it achieves response times that are in general shorter than those obtained by on-disk DBMSs. This difference is increased as the number of queried values which are not in the database increases. The difference between using one or two PDCFs is again significant, and saving useless accesses to memory reduces the response time, as expected. The reasons for this are explained in Fig. 5, where we show the number of unnecessary accesses to both memory and disk, in logarithmic scale, when the percentage of queried values which are not in the database is 5%. Thanks to the first PDCF, we save most of the unnecessary disk accesses made by LP. The clear benefit of using the second PDCF is that the number of accesses to memory is reduced.

4 Related Work

Since the 1980s, with the availability of very large, relatively inexpensive, main memory, it became possible to keep large databases resident in main

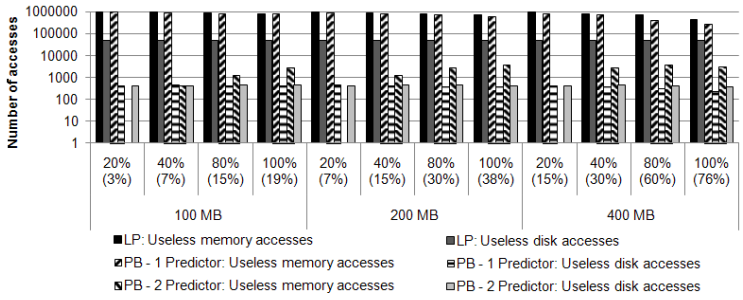

Fig. 5. Analysis of useless disk accesses.

memory [4], and concepts such as main memory database management systems (MMDBMS) became usual. As described by Hector Garcia-Molina in [1], MMDBMSs store their data in main physical memory providing very high speed access. MMDBMSs use optimizations to structure and organize data. Because of the volatile characteristics of the main memory, reliability on MMDBMS has been one of the main concerns in the area and a lot of work has been done concerning the recovery, logging and checkpointing of such systems [5]. The interest on MMDBMS has increased [6] and many commercial MMDBMSs have been developed. MonetDB [7], Altibase [8], IBM-SolidDB [9] or Oracle-TimesTen [10] are just some examples. SolidDB and Altibase allow MMDBMS and conventional DBMS to coexist although they do not allow for hybrid tables in the way that we propose.

The use of structures such as PDCF is usual for several purposes such as analyzing data streams [11], summarizing the content of peer-to-peer systems [12], reducing file name lookups in scale ditributed systems [13], etc. In databases, these structures have been used to answer queries regarding the multiplicity of individual items such as in [14].

5 Conclusions and Future Work

The results in this work show that using hybrid tables is a good solution when the whole table does not fit in memory. To our knowledge this is the first approach towards a hybrid in-memory and on-disk table allowing to reduce reading time by splitting the table into two parts. The use of predictors is essential in order to reduce useless accesses to both memory and disk. In the case of accessing values which are not in the database, foreseeing their non-presence and completly avoiding the access to data improves the overall performance of the system, making our proposal important for real-time applications, where a large number of users might be querying the same table.

The experiments have been done following a random pattern for accessing rows. However, it is still necessary to test the benefits of a hybrid table when the access pattern is skewed and a few rows are accessed very frequently, while the remaining rows are seldom accessed. Taking this into account, future work in-

cludes, among other possibilities, onsidering reallocation of data between devices in order to favor the system response time reduction.

Acknowledgments

The authors would like to thank the Center for Advanced Studies (IBM Toronto Labs) for supporting this research. We also thank Generalitat de Catalunya for its support through grant GRC-1087 and Ministerio de Ciencia e Innovación of Spain for its support through grant TIN2009-14560-C03-03.

References

1. H. Garcia-Molina, K. Salem, Main memory database systems: An overview, IEEE Transactions on knowledge and data engineering 4 (6) (1992) 509–516.
2. B. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 426.
3. J. Aguilar-Saborit, P. Trancoso, V. Muntés-Mulero, J.-L. Larriba-Pey, Dynamic adaptive data structures for monitoring data streams, Data Knowl. Eng. 66 (1) (2008) 92–115.
4. D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, D. Wood, Implementation techniques for main memory database systems, in: Proceedings of the 1984 ACM SIGMOD international conference on Management of data, ACM New York, NY, USA, 1984, pp. 1–8.
5. A. Liedes, A. Wolski, SIREN: a memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases, in: Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on, 2006, pp. 99–99.
6. M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, P. Helland, The end of an architectural era:(it's time for a complete rewrite), in: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment, 2007, pp. 1150–1160.
7. P. Boncz, M. Kersten, S. Manegold, Breaking the memory wall in MonetDB.
8. K. Jung, K. Lee, H. Bae, Implementation of storage manager in main memory database system altibase TM, in: Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems, Lecture Notes in Computer Science, 2004.
9. soliddb, `http://www-01.ibm.com/software/data/soliddb`.
10. C. Team, In-memory data management for consumer transactions the timesten approach, ACM SIGMOD Record 28 (2) (1999) 528–529.
11. S. Wang, X. Hao, H. Xu, Y. Hu, Finding frequent items in data streams using ESBF, Lecture Notes in Computer Science 4819 (2007) 244.
12. A. Broder, M. Mitzenmacher, Network applications of bloom filters: A survey, Internet Mathematics 1 (4) (2004) 485–509.
13. J. Ledlie, L. Serban, D. Toncheva, Scaling filename queries in a large-scale distributed file system, Tech. rep., Citeseer (2002).
14. G. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proceedings of the 28th international conference on Very Large Data Bases, VLDB Endowment, 2002, p. 357.