Multi-Indices - A Tool for Optimizing Join Processing in Main Memory

Dmitry Shaporenkov

University of Saint-Petersburg, Russia
dsha@acm.org

Abstract. In this paper we revise a well-known technique for optimizing joins - join-indices. We propose a variant of join-indices, multi-indices, which are specifically tailored for main-memory databases. We discuss trade-offs for creating multi-indices, outline implication of multi-indices on update and insert procedures and describe their usage in query processing algorithms. For some important particular kinds of queries involving selections we also propose a further optimization that places a pointer to the shared index record into the data record and thereby avoids search in the index structure altogether.

Keywords. Access methods, join algorithms, main-memory databases

1 Introduction

In the last decade advances in computer hardware allowed to store relatively large databases in main memory, attracting great attention to main-memory databases in research community. It has been shown that main-memory databases provide performance on order of magnitude better than traditional, disk-based databases [3]. A main-memory database system (MMDBMS) can store all the data and support structures (such as indices) in main memory, using disk only for logging and recovery and avoiding inefficient random access to the mechanical device [2].

Changing primary data storage from disk to main memory does not free database systems developers from the problems peculiar to traditional databases. In particular, efficient data access is still very important for MMDBMS. However, MMDBMS bring new criteria for evaluating access methods - CPU cache utilization. Recent research has shown that CPU cache misses are the biggest performance bottleneck for MMDBMS [7, 1]. Many algorithms and data structures traditionally used in DBMS have been revised in last years from the viewpoint of cache behavior [8, 5, 11, 1].

In this paper we focus on performance of equi-join operation, which is widely used in query processing algorithms for relational DBMS. We propose *multi-indices* - a sort of indices mapping an attribute value to all records of different relations containing this value. Multi-indices employ the same idea as the well-known join indices - precomputing information required during join operation. Using multi-index, one can find all pairs of matching records without scanning or preprocessing relations. We propose a layout of index structures that seems to be optimal with respect to cache behavior. For queries involving selections, performance can be further improved by placing a pointer to the index record into the data record. We expect this modification to significantly speed up execution of such queries.

2 Related work

The CSB+-tree proposed in [8] features a layout of tree nodes with only one child pointer. This structure frees extra space in a node for keys, increasing tree branching factor and decreasing height of the tree. This reduces number of cache misses during tree traversal at the cost of more complex node split algorithm. One of the latest works in the field, [11], uses a buffering technique for optimizing search in B+-tree. Buffering helps to avoid cache miss occurring when walking down from a parent to a child in the tree. An algorithm is described that distributes buffers among the tree nodes during query processing thus accommodating buffering strategy to the workload.

Shatdal, et al [5] improve cache behavior of several widely used algorithms for join and aggregation. The work also contributes a categorization of cache-conscious methods. Methods of the first category try to exploit temporal locality and reuse the data previously loaded in the cache by reducing the working set of the algorithm, while methods of the second category use spatial locality and partition the process in such a way that each part works with a relatively small block of memory that fits in the cache entirely. Kersten et al. [7] employ radix-clustering scheme to attain even better cache performance of join operation.

Our work is based on the idea of join indices proposed by Valduriez in [10]. Given two relations R and S and a join criterion, binary relation of pairs (r, s) (where r and s are RIDs of records of R and S which can be joined using the join criterion) is built. The index on r for this relation is then used to find all RIDs of records of S joined with the given record of R. While on the logical level our multi-indices resemble those proposed by Valduriez, we focus on efficient physical representation of index records under specific circumstances of main-memory database system. At the same time we do not bind the idea with the particular index structure. Like domain indices [6], multi-indices may index more than two relations. We also propose further development of the idea and claim that in many cases significant speed-up can be gained by placing pointer to the index record into the data record and avoiding the index search altogether.

3 Creating and using multi-indices

The main idea of multi-indices is to store information about all records of different relations containing given value of an attribute in the same place. Let $R_1, ..., R_k$ is a set of relations with common attribute A. *Multi-index* of the relations $R_1, ..., R_k$ on the attribute A is a mapping

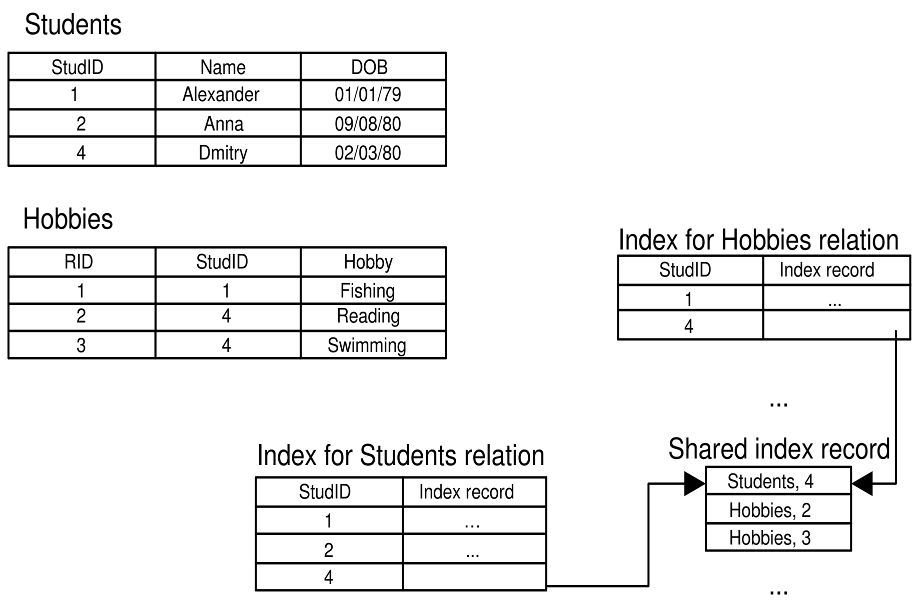
$$I^A_{\{R_1,...,R_k\}} : Domain(A_{R_1}, ..., A_{R_k}) \to \{r_i\},$$

where $Domain(A_{R_1}, ..., A_{R_k})$ is a union of attribute A's domains in relations $R_1, ..., R_k$ and r_i is a record of a relation R_l for some $l = 1, .., k$. Notice that the multi-index $I^A_{\{R_1,...,R_k\}}$ can serve as a conventional index for any of the relations $R_1, ..., R_k$ on the attribute A - we only need to filter out records of other relations. We call k the *degree* of the multi-index.

We also define *index record* as a structure containing RIDs of records with the given value of the attribute. The nature of RID depends on the implementation and the storage model being used. We investigate methods for efficient organization of index record later in this work.

3.1 Organization of multi-indices

There can be different ways for organizing multi-index $I^A_{\{R_1,...,R_k\}}$. For simplicity, let us assume we have two relation R_1 and R_2 with the common attribute A. We can then build two indices on A for R_1 and R_2 with shared index records, so that a value of the attribute can be mapped to records of R_1 and R_2 with this value of the attribute by single lookup in either index for R_1 or R_2. As an alternative, we can use single index structure for both R_1 and R_2. Either approach has its advantages, and choice between them should be made taking into account several factors.


Students

StudID	Name	DOB
1	Alexander	01/01/79
2	Anna	09/08/80
4	Dmitry	02/03/80

Hobbies

RID	StudID	Hobby
1	1	Fishing
2	4	Reading
3	4	Swimming

Index for Hobbies relation

StudID	Index record
1	...
4	

Index for Students relation

StudID	Index record
1	...
2	...
4	

Shared index record

Students, 4
Hobbies, 2
Hobbies, 3

Figure 1. An example of relations {Students, Hobbies} and multi-index on StudId attribute

First, distributions of A's values in relations R_1 and R_2 can differ significantly, so if a single index structure is used for both relations, index lookup is less efficient than it would be if two separate index structures were used. Second, a single index structure for a set of keys occupies less memory than two index structures mapping the same set of keys even if two structures share index records. Besides possible duplication of keys in both structures, there are also auxiliary elements like link pointers etc. Third, as we will show, implementation of multi-indices in the form of several index structures with shared index records degrades performance of insert and delete operations, so for environment with intensive modifications we may prefer a single index structure.

Choice among different variants of implementation of multi-index is therefore a trade-off between storage space, search and update efficiency. If there are three relations with common attribute, the situation becomes even more complicated, as we now have 5 options for constructing multi-index. It is known that the number of all possible partitions of a set (and therefore the number of options for constructing multi-index) is described by Bell numbers [9]. Generally, the issue of choosing optimal implementation of multi-index can be treated as an optimization problem where we are trying to minimize average search cost under restrictions on the amount of main memory available and overhead for insert and update operations. Space limits preclude us from detailed description.

In Figure 1 an example of two relations {*Students*, *Hobbies*} and a multi-index on the *StudId* attribute is depicted. In the example an index on *StudId* is built for both relations, and indices share common index records. Assuming that only some of students have hobbies, such an organization seems to be better than the single index structure.

We propose the following index record layout. Index record header contains the length of the index record and a table $(RelId \rightarrow offset)_{i=1,\ldots,m}$ that maps unique relation identifier to the offset of the first record identifier from the relation in the index record. Following the record header, record identifiers from the relations $RelId_1$, ..., $RelId_m$ are stored. The main idea is therefore to group record identifiers from the same relation together. Compared with simple sequential layout, this organization incurs some overhead during update of the index record, because we may need to shift part of the record to accommodate to new relation boundaries or even reallocate the record if no free space available inside the record. If, however, modifications to the database are rare as compared with queries, this overhead is well rewarded by the resulting cache efficiency. Fetching record identifiers for relation l costs $1 + \frac{size(I_l)}{size(cache\_line)}$ cache misses in the worst case (where I_l is the size of the block of records identifiers from relation l); fetching records identifiers for relations l and m costs $1 + \frac{size(I_l)+size(I_m)}{size(cache\_line)}$ cache misses.

3.2 Maintaining and using multi-indices

An algorithm for equi-join of two relations R_1, R_2 on common attribute A using multi-index $I^A_{\{R_1,R_2\}}$ is straight-forward. We simply traverse the multi-index collecting matching records. Notice that an optimization is feasible in the index record layout: if we have a multi-index $I^A_{\{R_1,\ldots,R_k\}}$ and there are expectations that two particular relations R_l and R_m will often be joined in queries, we may place record identifiers from these relations contiguously in the index records thereby decreasing the probability of the extra cache miss. Another optimization is to place record identifiers from the most frequently using relations closer to the beginning of the index record, since this may reduce the number of cache misses occurring during processing of the index record to 1.

We compare the multi-index join with the partitioned hash-join which has been proved to be one of the most efficient equi-join algorithms for MMDBMS [5]. Partitioned hash-join requires preprocessing step that divides relations (say R and S) being joined into partitions based on hash codes of values of the join attribute. This step involves scanning both relations and therefore exhibits poor cache performance because of large number of compulsory cache misses. Each record of the relations should be accessed in order to compute its hash code. The number of cache misses depends on the underlying storage model (one can lay out attributes in such a way that values of the join attribute occupy contiguous region in memory, thereby decreasing number of cache misses required to read them all), but in general for conventional N-ary storage model we may expect one cache miss per each record. After preprocessing step, hash-join builds hash table for each partition of relation R and probes all records in each partition of relation S using this hash-table.

Multi-index does not require any preprocessing, as it already has all the necessary information to perform the equi-join. We only need to traverse the index structure and for each index record form pairs (r,s) of matched records. Notice that conventional index structures like B+-trees and bucket-chained hash tables all support efficient traversal. Of course, we still experience large number of compulsory cache misses. However, we expect this number to be substantially smaller than that during preprocessing step in partitioned hash join, because index records better utilize cache lines and do not pollute them with irrelevant information (such as values of attributes not participating in join). For multi-indices of the smallest degree 2, the cache utilization is especially high, since all the information contained in an index record is required during join. For multi-indices of higher degrees, we have mentioned a couple of optimizations aimed to increase cache utilization. In the multi-index join we also do not need to perform hash code computation, which can be a CPU-bound operation. Given all the above, there are good prerequisites for multi-index join to perform better than the partitioned hash-join.

During insertions, deletions and updates to the database, multi-index, like any other index, should reflect changes in the data. We need to find the index record corresponding to the value of the attribute in the data record being inserted or deleted. If the multi-index consists of the single index structure, we only need to lookup in this structure. If, however, multi-index is a union of several index structures with shared index records we generally need to search in all these indices until the index record is found or the search is exhausted. This may seem expensive but notice that in practice we almost always know which part of the multi-index should definitely contain the shared index record corresponding the data record being inserted or deleted. For example, when we insert data record into or delete from details relation in 'master / details' scheme, we do know the RID of the corresponding master record, so we only need to search in the part of the multi-index for the master relation. The same applies to the situation when we add or delete fact record from facts relation in the star scheme; in this case we know the RIDs of corresponding records in dimension relations.

Multi-indices can be useful during modifications to the database for enforcing referential integrity constraints [10, 6]. Let us suppose we have three relations: *Students (StudId, Name)*, *Marks (StudId, Subject*, Value) and *Hobbies (StudId, Name)* where *Marks.StudId* and *Hobbies.StudId* are foreign keys for which CASCADE DELETE constraint is enforced. If we need to delete a record from the *Students* relation, multi-index $I^{StudId}_{\{Students,Marks,Hobbies\}}$ enables us to find all records about student's hobbies and marks using single index lookup. In case of two conventional indices on *StudId* for *Marks* and *Hobbies* two lookups would be required.

3.3 Connecting multi-indices and data records

In the above multi-indices were just a generalization of conventional indices; the data records remained unchanged. Unfortunately, such an approach limits usefulness of multi-indices. To demonstrate this, we return to the Students example. Let us suppose we have to find all marks of a student given the student's name: SELECT Subject, Value FROM Students, Marks WHERE Students.Name = 'Anna' AND Marks.StudId = Students.StudId. Good query execution plan first makes use of index *Students.Name* (which we assume to exist), and only then performs the join. The join is therefore used to find all records about student's marks given the *StudId*. But this just as well could be done with conventional index on *StudId* for the *Marks* relation. An alternative execution plan could first join *Students* and *Marks* using multi-indices and then filter out all records which bear nothing to Anna, but this plan looks unreasonable because of low selectivity of the query.

We can make multi-index more useful if we directly connect data records with the corresponding index records as illustrated in Figure 2. If we place a pointer to the index record into data record of the *Students* relation, we will be able to perform the query quite efficiently : we will only need to find a record about Anna in the *Students* relation and then follow the pointer to the index record where we fetch RIDs of corresponding records from Marks relation. This query execution plan does not involve any index search for performing the join. Notice that such a modification of data record's structure is especially simple for MMDBMS, because in MMDBMS we keep relations and multi-index in main-memory, and we do not have to perform any additional address translation when moving indices and relations from disk to memory and vice versa.

Of course, placing a pointer to the index record into the data record has some negative consequences, most obvious of which is that the pointer occupies extra

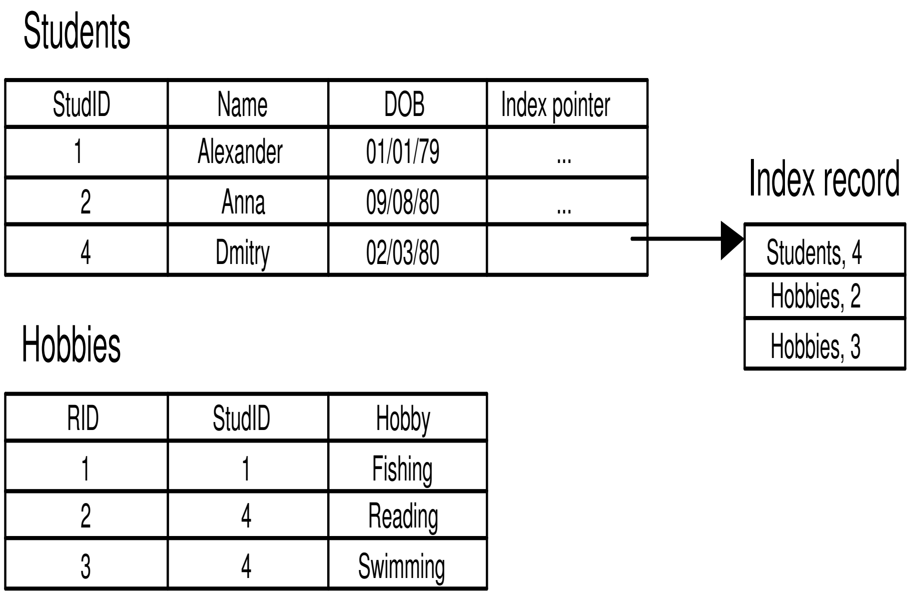


Figure 2. Direct references from records of the *Students* relation to index records

storage. Another unwanted side effect is possible decrease of cache utilization by data records. For those queries which do not include join, the pointer will simply waste valuable cache space. A possible solution to this problem is an adaptive data layout technique like one discussed in [4], which lay out contiguously attributes frequently occurring together in queries. Third issue is that after inserting a new data record the corresponding index record can be reallocated, and we will need to update all pointers to it. However, this can be done efficiently, because all data records holding pointers to the index record are referenced in the index record itself, so we can easily update them having the index records in hand. Generally, we expect that in many cases the proposed modification pays off and significantly increases effectiveness of multi-indices.

4 Implementation and experimental study

We are currently working on implementation and tuning of multi-indices in Memphis, our main-memory research database kernel. The primary goal of Memphis project is to provide a flexible architecture that would allow to support various storage models and index structures, easily plug compression algorithms and include new data types extending the predefined set of built-in types. The salient feature of Memphis is that it is written entirely in a managed language C# with performance-bound places using unsafe code [12]. This greatly facilitates integration of Memphis with applications targeted .NET framework. We are going to use Memphis as an underlying storage system for the repository containing large amount of information about source code of software projects (this information includes such code objects as classes, methods, properties and relationships among them).

We have implemented some conventional join algorithms used in MMDBMS (partitioned hash join, blocked nested loops, index) as well as multi-index joins. We use bucket hashing for multi-indices; for better efficiency we provide specialized representations of index records for multi-indices of several small degrees. These specialized representations contains fixed number of offset fields in place of general offset table described in section 3.1. We have not implemented references from data records to index records yet.

Our preliminary experiments show that multi-indices work well in important situations, in particular, for joins involving one master and several details relations (which is a very common case in practice). We present some experimental results here. These results only concern performance of join operation; measurements of insert and update overhead caused by multi-indices as well as a more detailed study of joins will be given in the extended version of the paper. We tested our implementation on several datasets, including the real dataset and synthetic ones. Synthetic datasets were generated by a tool that is a part of the Memphis project. This tool takes such properties of the dataset as relations cardinalities, types of attributes, and distribution of attribute values and produces relations in the form of plain text files that can be bulk-loaded into memory of the Memphis system. Each experiment was performed on the cold system; that is, the system was started, all the relations were loaded, then the join was performed. The presented measurements are average values obtaining by running each test several (5-6) times. The experiments were conducted on a conventional workstation (Intel P4 2.8 GHz, 1 Gb RAM). In our tests we compared the performance of the join algorithm using multi-index with in-memory partitioned hash-join algorithm ([5]). The number of partitions in the hash-join algorithm was chosen to provide the optimal performance. Results of experimental runs for some datasets are summarized in the Figure 3.

$Dataset$	$\lvert R\rvert$	$\lvert S\rvert$	$T_{HashJoin}$	$T_{CreateMultiIndex}$	$T_{MultiIndexJoin}$	$MultiIndexSize$
Classes	8000	6000	1.6 sec	0.8 sec	1 sec	0.4 Mb
SynthI1	30000	50000	1.2 sec	1 sec	0.85 sec	0.6 Mb
SynthI2	100000	300000	7.6 sec	4.8 sec	7.7 sec	1.8 Mb
SynthS1	100000	300000	43 sec	24 sec	8 sec	2.1 Mb

Figure 3. Results of experimental runs

For each dataset $|R|$ and $|S|$ are cardinalities of the relations being joined, $T_{HashJoin}$ is the running time of the partitioned hash-join algorithm, $T_{CreateMultiIndex}$ is the time required to build the multi-index, $T_{MultiIndexJoin}$ is the running time of the join algorithm using multi-index, and $MultiIndexSize$ is the amount of memory occupied by the multi-index. Since in this experiment we assumed that the multi-index is built during the join processing, the total running time of the multi-index join is $T_{CreateMultiIndex} + T_{MultiIndexJoin}$. In practice, however, the multi-index is maintained up-to-date, so the first term $T_{CreateMultiIndex}$ disappears from the latter formula.

Dataset *Classes* contains information about classes (basically, names of classes and their members) defined in source code of a large object-oriented system. This information is stored in the normalized form. The relation R ('Classes') contains *ClassID* and *ClassName* attributes. It is joined with the relation S ('Members') that consists of *ClassID* and *MemberName* attributes, where *S.ClassID* is a foreign key referring to the *R.ClassID*.

$SynthI1$ and $SynthI2$ are synthetic datasets that include two relations, each of which contains one integer and one string attribute. Integer attributes are unformly distributed in the interval $[1, 100000]$. The relations are joined using integer attributes as the join attribute. Finally, $SynthS1$ contains two relations, with two string attributes in each relation. The pair of joined attributes are short strings (3-5 characters) that mimic e.g. item code represented in a string form. Note that in case of string attributes the multi-index join provides the noticeable speed-up, as it dramatically reduces the number of expensive string comparisons needed to compute the join result.

5 Conclusion

In this paper we have presented multi-indices - a generalization of conventional indices commonly used in databases. Multi-indices essentially precompute information needed to perform equi-joins by mapping an attribute's value to all the records of different relations containing this value. This enables very efficient processing of equi-joins. We have discussed trade-offs for constructing multi-index, the layout of index records optimized for better cache utilization, maintenance and usage of multi-indices. We have also proposed a further development of the idea - connecting index records with data records, which enables to avoid search in the index structure and leads to even better performance.

We believe that multi-indices can be an effective tool for main-memory database systems. Currently we are working on tuning and experimental study of multi-indices in our research database kernel Memphis. Preliminary experiments show that multi-indices perform well in practically important situations. We will present detailed experimental results in the extended version of this paper.

References

[1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB Journal*, 11:198–215, 2002.

[2] David J. DeWitt, et al. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, 1984.

[3] H.V. Jagadish, et al. Dali: A high performance main memory storage manager. In *Proceedings of the 20th VLDB Conference*, 1994.

[4] Richard A. Hankins and Jignesh M. Patel. Data Morphing : An Adaptive, Cache-Conscious Storage Technique. In *Proceedings of the 29th VLDB Conference*, 2003.

[5] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th VLDB Conference*, pages 510–521, 1994.

[6] Changho Kim. Join Processing and Domain Indices, 1991.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th VLDB Conference*, pages 54–65, 1999.

[8] Jun Rao and Kenneth A. Ross. Making B+-Trees Cache-Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000.

[9] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics. A Foundation for Computer Science.* Addison-Wesley, second edition, 1998.

[10] Patrick Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12:218–246, 1987.

[11] Jingen Zhou and Kenneth A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *Proceedings of the 29th VLDB Conference*, 2003.

[12] C# Language Specification. ECMA-334 International Standard, 2001.