Partitioning Inverted Lists for Efficient Evaluation of Set-Containment Joins in Main Memory

© Dmitry Shaporenkov

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

Abstract

We present an algorithm for efficient processing of set-containment joins in main memory. Our algorithm uses an index structure based on inverted files. We focus on improving performance of the algorithm in a main-memory environment by utilizing the L2 CPU cache more efficiently. To achieve this, we employ some optimizations including partitioning the inverted lists and compressing the intermediate results.

1 Introduction

Set-valued attributes have become more important in recent years with growing distribution of object-relational database systems (ORDBMS) and rapid development of such application areas as information retrieval and data mining. In practice it is often required to evaluate join queries on set-valued attributes. In such cases the join predicate is a set predicate, such as set containment or intersection. Many real-world queries can be naturally expressed as set containment and intersection joins. For example, a query that finds appropriate candidates among job seekers includes a condition that the set of candidate's skills contains the set of skills required for the job as a subset. If we are interested in retrieving all documents containing the specified set of terms from the collection, this again can be considered a set containment query. A relation *People* that includes a set-valued attribute *Hobbies* leads us to the problem of finding all pairs of people sharing common hobbies that can be formulated as a set intersection self-join.

Set-valued attributes are not directly supported in a traditional relational DBMS, since already the first normal form explicitly requires an attribute to be atomic, i.e. forbids the value of an attribute to be a set. However, set-valued attributes in a relational DBMS can be simulated using unnested external representation [3] that creates an auxiliary relation connected to the original relation by a foreign key, thus representing one-to-many relationship between a record of the original relation and the elements of the value of its set-valued attribute. It can be easily noticed [7] that many complex joins on atomic attributes that arise in relational DBMS in fact hide set predicates behind sophisticated expressions involving aggregation. However, as study [3] shows, relational query optimizers are generally unable to deal with such queries in an efficient manner, since set predicates are unknown for them.

Given the growing practical importance of joins with set predicates, efficient algorithms for performing such joins are required. Nested-loops algorithm is the most common way to handle joins with complex predicates. However, in case of joins with set predicates nested loops algorithm falls short, because its poor performance makes it inappropriate [2]. The reason is that testing predicates on sets is a very expensive operation. Its cost in large extent depends on how the sets in question are represented, but in general this cost is much higher than the cost of simple join predicates on atomic attributes traditionally used in relational database systems.

Main-memory DBMS (MMDBMS) have attracted much attention during recent decade. A MMDBMS stores all the data and support structures (such as indexes) in RAM of the database server. Constantly growing amount of memory in modern database servers already enables to store small and medium-size databases directly in main memory. It has been shown that MMDBMS provide huge performance gain over traditional, disk-based DBMS, since retrieving the necessary data in MMDBMS usually does not involve disk access at all. Slow disk device is used only for logging and recovery. Many researchers [11, 1] have recognized that the crucial factor for performance of a MMDBMS is CPU cache utilization, that is, how many cache misses database operations incur. If the number of cache misses is high, CPU will spend most of time waiting the data to be fetched from RAM (so-called CPU *stall*).

In this paper we present an efficient algorithm based on inverted files for set-containment joins in main memory. Inverted files are well-known and widely used tool for indexing text documents. Our algorithm takes two relations R and S sharing a set-valued attribute A as parameters, scans the common inverted file IF_{RS}^{A} built on the set-valued attribute A for R and S and processes the inverted lists in such a way that the resulting structure appears to be the answer to the set-containment query. The idea of the algorithm is fairly simple, and we focus our study on tailoring the algorithm for MMDBMS by improving CPU cache utilization. We achieve this by partitioning the inverted lists being processed. This enables to fit the working set of the algorithm into the L2 CPU cache and reduce the number of L2 cache misses. We also study the effect of compressing the intermediate re-

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2005

sults which provides significant memory savings and allows to join larger relations at the cost of more intensive CPU usage and some loss in response time. We present experimental results showing that our optimizations give significant effect as compared with a straightforward implementation. We also demonstrate that the algorithm is superior to some other algorithms for set-containment joins.

The structure of the paper is as follows. The section 2 presents a survey of related work in the area of algorithms for joins with set predicates. In the section 3 we first describe the basic algorithm for set-containment joins using inverted files, and then discuss various optimizations aiming to improve its performance. The section 4 summarizes the results of experimental evaluation of the algorithm. The section 5 concludes the paper and outlines directions for future work.

2 Related work

Helmer and Moerkotte [2] seem to be the first researchers who addressed specifically set containment joins. They evaluated several algorithms for set containment join in main memory. The first group of algorithms includes variations of nested-loops join which differ in how the set comparison is implemented. Three implementations of set comparison were considered: the naive algorithm, an implementation based on sorting the sets and an approach that uses signatures. The latter turned out to be the best among nested-loops joins. The second algorithm employs signature-based approach by hashing all the signatures of the relation R (assuming that the join condition is $t_R.A \subseteq t_S.A$, t_R, t_S are tuples of R and S, respectively, $t_R.A$ is a set that is the value of A in t_R), enumerating subsets of each set of the relation S, and matching each subset with hashed signatures of R.

Melnik and Garcia-Molina [7] describe two algorithms for set containment joins. Both algorithms exploit essentially the same idea, namely, partitioning the relations being joined in such a way that the join result can be computed by joining sets from each pair of corresponding partitions and then merging intermediate results. The main problem with this approach is that in the case of set containment joins the partitions inevitably intersect. The first method, Adaptive Pick-and-Sweep Join, extends the Pick-and-Sweep algorithm proposed in [8]. The algorithm is parameterized by a set of $\{h_1, ..., h_k\}$ boolean hash functions that take sets as input. For each tuple $t_R \in R$ the function h_i is randomly chosen such that $h_i(t_R.A) = true$, and t_R is assigned to the partition R_i. For each tuple $t_S \in S$ all the functions h_j such that $h_j(t_S.A) = true$ are chosen, and t_S is assigned to each of the partitions S_j. The second method, Adaptive Divide-and-Conquer Join, progressively refines the partition assignment. It starts with the single partition pair, and on each step doubles the number of partitions by applying a hash function to either set $t_R.A$ or $t_S.A$. On each step the partition assignment is modified to make the condition that each set of R_i can be contained only in sets of the corresponding S_i true (details can be found in [7]).

Mamoulis [4] considers several algorithms for set containment, intersection and overlap join (two sets $s1$ and $s2$ are said to *k-overlap* if they have at least k elements in common). He proposes Block Nested-Loops algorithm (BNL) that uses inverted file S_{IF} built on the relation S. The S_{IF} is partitioned into blocks each of which can fit into the main memory. The BNL algorithm proceeds by reading each block of S_{IF} and scanning the relation R to find qualifing tuples. Different strategies for handling the intermediate results are evaluated. Mamoulis also discusses the algorithm IFJ (IFJ - *Inverted File Join*) that joins two inverted files R_{IF} and S_{IF}, but rejects this algorithm as inefficient. While IFJ and our algorithm are based on essentially the same idea, our algorithm targets specifically MMDBMS and exploits some important optimizations improving cache performance like partitioning the processing to fit the intermediate results into the L2 CPU cache. Our partitioning method differs from that of Mamoulis, as he simply reads the inverted files block by block, while our method rather operates on the individual inverted lists and processes only the part of each list relevant to the current partition. The reason is that Mamoulis designed and evaluated his algorithms in the context of a disk-based DBMS where efficiency criteria differ significantly from those in a MMDBMS.

During the last decade, many algorithms commonly used in DBMS in the course of many years were reconsidered from the viewpoint of their suitability and optimality for main-memory DBMS. [11] was one of the first works concerning this problem. It suggests cache-conscious versions of several well-known database algorithms such as hash-join, and also demonstrates some fundamental techniques that can be used for improving performance of main-memory algorithms. One of the recent works in the field, [6], discusses cache-conscious hash-join algorithms including projections on different storage models. These and other works have made great contribution by increasing researchers' and developers' awareness of cache performance issues. However, to the best of our knowledge, more complex database algorithms like one discussed in this paper have not been reviewed from the viewpoint of their optimality on modern hardware yet.

3 Set-containment join algorithm using inverted files

Inverted files are a well-known technique for indexing text documents. Essentially, an inverted file provides a mapping of a term T into the list of documents $D_1, ..., D_k$ where this term occurs (D_j is an ID of the document in the collection). [13] discusses in depth various methods for constructing inverted files. To reduce storage cost, inverted files can be efficiently compressed at almost no loss in search speed: the list of documents $D_1, ..., D_k$ is ordered, and instead of storing document IDs, the differences $D_i - D_{i-1}$ are kept in an encoded form. Given that the access to inverted list is mostly sequential, document IDs can be easily decoded 'on the fly'.

We apply inverted files in a different context but the idea remains the same. Instead of collection of documents, we consider relations R and S and the common set-valued attribute A. We denote $Domain(A)$ the do-

main from which elements of values of A in tuples of R and S are drawn. Tuples of relations are identified by record ID (RID). Inverted file R_{IF} then maps an element of the $Domain(A)$ into the list of RIDs of tuples whose value of A contains this element.

We design our algorithm under assumption that all the data and indexes are kept in main memory. This assumption greatly affects the algorithm design, since main focus now shifts from minimizing the number of disk accesses (which we assume do not happen at all) to minimizing the number of cache misses. While both these aims require good reference locality, the methods for achieving them are somewhat different [6]. The first important difference between the CPU cache and file cache in RAM is that the former is normally much smaller. The second difference is that generally speaking CPU cache cannot be directly controlled by the program.

Our algorithm employs the idea of *join indexes* [12, 9] to speed-up the join. Instead of building two separate inverted files R_{IF} and S_{IF}, we build one inverted file (we denote it RS_{IF}) that maps an element v of the $Domain(A)$ into two lists of the RIDs, l_R^v and l_S^v, where l_R^v and l_S^v consist of all RIDs of tuples from R and S, respectively, each of which contains the element v among the values of the set-valued attribute. In an actual implementation, l_R^v and l_S^v can be merged into one list, but there should be an efficient way to separate them. This enables us to find all tuples from R and S containing the given element using only single lookup in the inverted file. As noticed in [9], this property comes at the cost of some loss in efficiency in case if RS_{IF} is used in role of either R_{IF} or S_{IF} (that is, if RS_{IF} is used for finding all tuples from either R or S containing an element). The exact value of the decrease in search efficiency depends on the implementation of the inverted file. In our implementation it is quite affordable. On the positive side, combining two inverted files into one gives us a very efficient way for traversing all elements of the $Domain(A)$ and their corresponding inverted lists in both relations without using index lookup.

3.1 Basic set-containment join algorithm

We present the basic version of the algorithm for set-containment joins based on inverted files. We assume that the join predicate is $r \subseteq s$ where r, s are values of the set-valued attribute A in relations R and S, respectively. We denote t_R, t_S tuples of relations R and S, $t_R.rid$, $t_S.rid$ RIDs of tuples, and $t_R.A$, $t_S.A$ their values of the attribute A. Let also $|Result|$ be the cardinality of the result relation that consists of pairs $(t_R.rid, t_S.rid)$ qualifying the containment predicate $t_R.A \subseteq t_S.A$. The algorithm traverses the combined inverted file RS_{IF}, and on each step processes inverted lists l_R^v, l_S^v, where v is an element of the $Domain(A)$ (both lists may be empty). It maintains a mapping $Workmap$ that maps $t_R.rid$ into a reference to the list of $L_r = t_{S_1}.rid, ..., t_{S_n}.rid$ such that each of $t_{S_k}.A$ contains all the elements of $t_R.A$ encountered so far. These lists shrink as the algorithm proceeds. After the final iteration $Workmap$ is exactly a mapping of $t_R.rid$ into the list of $t_{S_1}.rid, ..., t_{S_m}.rid$ where $t_R.A \subseteq t_{S_k}.A, k = 1, ..., m$. The average length of the L_r in the final result can be estimated as

```
Workmap :  map :  RID -> ref to list of RID;
foreach (Value v in RS_IF)
{
  l_R^v, l_S^v :  ref to list of RID;
  Initialize l_R^v and l_S^v;
  foreach (t_R.rid in ↑l_R^v)
  {
     L_r, result :  ref to list of RID;
     L_r = Workmap.Get(t_R.rid);     (!)
     if (L_r = NULL)
       result = l_S^v;
     else
       result = Intersect(↑L_r, ↑l_S^v);  (!)
     Workmap.Put(t_R.rid, result);
  }
}
```

Figure 1: Basic algorithm for computing set-containment joins

$L(R,S) = \frac{|Result|}{|R|}$. The basic algorithm is presented in the Figure 1.

For clarity, our notation uses $\uparrow$ to mark dereferences which cause cache misses. This facilitates identifying sources of the cache misses in the algorithm.

The function $Intersect$ computes intersection of two inverted lists. Given that lists are kept in ascending order of RIDs, this function can be efficiently implemented by synchronously traversing both inverted lists. For the sake of simplicity, in our analysis we will ignore the fact that lists shrink during processing. Then the cost of computing the intersection in terms of elementary operations (such that index increment and integer comparison) does not exceed $C(|L_r| + |l_S|)$, where $|L_r|$ is the average length of the L_r and $|l_S|$ is the average length of inverted list for S, and C is a constant. Let |V| be the number of different values in the inverted file RS_{IF}, |R| and |S| - cardinality of relations R and S, and |r| and |s| - average cardinalities of values of the set-valued attribute A in relations R and S, respectively. Assuming that the values are uniformly distributed across tuples, the probability for a tuple t_S to have a value v among the elements of the set $t_S.A$ is equal to $P_{v \in s} = 1 - (1 - \frac{1}{|V|})^{|s|}$. Hence we have $|l_S| = |S| P_{v \in s}$, and the cost of the algorithm can be estimated as $|r||R| \left(C \left(\frac{|Result|}{|R|} + |S| P_{v \in s} \right) + C_{Workmap} \right)$ where $C_{Workmap}$ is the cost of lookup into the $Workmap$.

Our algorithm and cost estimation, however, do not take into account that the memory access is not uniform on modern architectures. Random memory access such that access to the $Workmap$ becomes very expensive in presence of several memory hierarchies like L1 and L2 caches, unless the $Workmap$ entirely fits into one of the caches. Let us then take a closer look on the memory reference behavior of the algorithm. Let C_{miss} be the cost of a L2 cache miss. The main sources of cache misses in the basic algorithm are (in the Figure 1 the corresponding lines are marked (!)):

- The $Workmap$ is accessed in a random fashion, and if the size of the $Workmap$ exceeds the size of the L2 cache, we may consider that every access to the $Workmap$ incurs at least one cache miss (The exact number of cache misses depends on the implementation of the $Workmap$. In case if $Workmap$

is implemented as a hash table, we expect the access to this hash table to be as efficient as access to an array, since the number of collisions in our situation is small). The total cost of these cache misses is therefore $|r||R|C_{miss}$.

- Dereferencing a reference to L_r that happens in the line $result = Intersect(\uparrow L_r, \uparrow l_S^v)$ incurs another cache miss. The cost of these cache misses is $(|r| - 1)|R|C_{miss}$, since the first access to the $Workmap$ for the given $t_R.rid$ does not cause a cache miss.

- Computing the intersection of two lists of lengths l_1 and l_2 sorted in ascending order incurs $\frac{(|l_1|+|l_2|)}{|CacheLine|}$ cache misses, where $|CacheLine|$ is the size of the L2 cache line.

Of course, there are other sources of cache misses in the algorithm, for instance, dereferencing references to the lists l_R^v and l_S^v may also incur a cache miss. However, provided that the |V| is small as compared with |R| and |S| (we expect this to be true in practical cases), the cost of those cache misses is negligible.

3.2 Partitioned version of the algorithm

Our method reduces amount of cache misses of all the aforementioned kinds. In order to eliminate misses of the first kind, we partition the processing to fit $Workmap$ into the L2 cache. This can be done in an obvious way - we modify the algorithm so that only a part of the tuples of R are considered at a moment, and other tuples of R are ignored. This also reduces the number of cache misses of the second kind, since the fewer inverted lists are processed at a moment, the more likely they reside in the L2 cache. The cache misses of the third kind are dealt with by compressing the L_r using one of many compression techniques for ascending sequences of positive integer numbers [13]. The modified version of the algorithm is presented in the Figure 2. To make presentation clearer, we omit compression. In reality, if inverted lists are already kept compressed in the inverted files, the algorithm can be unaware of compression at all. The reason is that the computation of intersection of inverted lists is done by the same code regardless of whether the arguments are compressed are not, since this computation accesses elements of lists in a sequential fashion.

The function $Select$ selects the range of RIDs relevant to the current partition p from the (possibly compressed) inverted list l_R^v. This function can be implemented in various ways: using sequential scan of the l_R^v, the binary search in l_R^v or even the specialized index on l_R^v that would enable to determine the first RID residing within the interval $[p.FirstID, p.LastID]$.

Let us now estimate the cost of the partitioned version of the algorithm. We can proceed by analogy with the case of the basic algorithm. For the partitioned version of the algorithm we also need to take into account the cost of the $Select$ function. Generalizing possible implementations of $Select$, we denote this cost C_{select}. Thus we have an additional term $N_p|V|C_{select}$ (where N_p is the number of partitions), and the overall cost in terms of elementary operations can be estimated as:

```
  PartBounds : array of
(FirstID:RID,LastID:RID);
  Fill PartBounds by partition R into
|PartBounds| partitions;
  foreach (PartBound p in PartBounds)
  {
    Workmap_p = map : RID -> ref to list of
RID;
    foreach (Value v in RS_IF)
    {
      l_R^v, l_S^v, l_Rp^v : ref to list of RID;
      Initialize l_R^v and l_S^v;
      l_Rp^v = Select(↑l_R^v, p);
      foreach (t_R.rid in ↑l_Rp^v)
      {
        L_r, result : ref to list of RID
        L_r = Workmap_p.Get(t_R.rid);
        if (L_r == NULL)
          result = l_S^v;
        else
          result = Intersect(↑L_r, ↑l_S^v);
        Workmap_p.Put(t_R.rid, result);
      }
    }
  }
```

Figure 2: Partitioned algorithm for computing set-containment joins

$$|r||R|\Big(C'\Big(\frac{|Result|}{|R|} + |S|P_{v\in s}\Big) + C_{Workmap_p}\Big) + N_p|V|C_{select} \quad (1)$$

where C' is a constant accounting for synchronous traversal of (possibly compressed) lists, and $C_{Workmap_p}$ is the cost of lookup into the $Workmap_p$. Comparison with the basic algorithm shows that the $C_{Workmap_p} \leq C_{Workmap}$, and, in presence of a compression, $C' > C$. So in terms of elementary operations we do not get clear benefits. Let us now turn to the cost of memory references. Since $Workmap_p$ now fits into the L2 cache, we do not get misses of the first kind at all. Since the number of inverted lists processed in a partition is fewer than that in the basic algorithm, we expect the cache misses of the second kind to happen more rarely. The number of misses of the third kind is also significantly reduced, since compressed lists occupy much less space and utilize cache lines more efficiently.

The question is therefore whether the better locality of memory references of the partitioned algorithm outweighs the cost of some additional CPU-intensive processing. As our experimental study reveals, under the condition that the number of partitions is properly chosen according to characteristics of the system, the partitioned version provides a significant performance improvement over the basic one.

The optimal number of partitions can be estimated as follows. Since it is desirable that all L_r fit into the cache, the number of tuples to be processed in each partition should not exceed $\frac{|CacheSize|}{size(L_r)}$, and hence the number of partitions should be the minimal number that is greater than $\frac{|R|size(L_r)}{CacheSize}$. $|L_r|$ decreases as the algorithm proceeds, but its average value remains in the bounds

$[L(R,S), |l_S|]$, where $L(R,S)$ and $|l_S|$, as before, are the average number of qualifying tuples of S for a tuple of R, and average length of inverted lists l_S^v in RS_{IF}, respectively. This observation enables to get an upper and a lower bounds for the number of partitions, but the interval between them can be quite large. So in practice, the number of partitions needs to be carefully tuned.

4 Experimental study

We have implemented both basic and partitioned version of the algorithm in our research prototype MMDBMS kernel Memphis. A short description of Memphis can be found in [10]. The implementation is written in the C# programming language [14] and runs on the .NET framework. Inverted files are implemented using the standard $Hashtable$ class. We considered two variants of the algorithm: the first does not use compression, and the second compresses inverted lists by encoding differences between subsequent elements in γ-code [13]. Unless explicitly mentioned, we assume that the version without compression is used. Our implementation of the $Select$ function uses sequential scan of l_R^v, since in the case of compressed inverted lists binary search would be impossible. Trying implementation of the $Select$ function in the form of index on long inverted lists is left for future research.

All experiments were conducted on a laptop with Intel P4 2.8 GHz CPU and 1 Gb RAM running under Windows XP. This machine features Intel Pentium Mobile processor with 2-level on-chip cache. The size of the L2 cache is 512 Kb, and the size of the L2 cache line is 128 bytes (these parameters were measured using Stefan Manegold's Calibrator tool [5]). For measuring the number of L2 cache misses we used the Intel VTune Performance Analyser [15] that provides a graphical user interface to various CPU counters. All reported times were estimated using *QueryPerformanceCounter / QueryPerformanceFrequency* Windows API that gives a programmatic access to the high-resolution hardware performance counter. To reduce noise, we present average times based on results of several runs. We do not include in the results the time necessary for constructing the inverted file (join index), since it is assumed to exist in a practical situation.

In the most of experiments, synthetic datasets are used. These datasets were generated by a program that takes desired characteristics of the dataset (relations cardinalities, average set cardinality, distributions of set cardinalities and set elements, size of the element domain etc.) as parameters and produces the resulting dataset in the form of a text file. Each experiment starts with bulk-loading the relevant relations from text files into memory. We used VTune's *ResumeSampling / SuspendSampling* API to profile only cache misses that happen during the join execution and exclude other stages of the process.

4.1 Case Study 1: Tuning number of partitions

In this case study, we use a dataset containing two relations, each of which consists of two attributes - the first is used as a primary key, and the second is a set-valued attribute. The characteristics of relations are given in the Table 1, the join selectivity is $\frac{5007}{|R||S|} = 1.2 * 10^{-7}$.

Table 1: Relations characteristics for Case Study 1

Rel.	Rel. Card.	Avg. Set Card.	$\lvert Domain(A)\rvert$
R	150000	5	5000
S	300000	5	5000

Partitions	Time, sec	L2 Cache Misses, * 10^6
1	3.9	42
3	3.66	36
5	3.52	33
8	3.72	33
11	3.84	33
16	4.04	32

Table 2: Tuning the number of partitions

Running times corresponding to different values of the N_p, i.e. to different numbers of partitions, are presented in the Table 2. Note that the basic version of the algorithm is essentially the case $N_p = 1$. Looking at the Table 2, it becomes obvious that 5 partitions (which correspond to $150000/5 = 30000$ RIDs in a partition) give the best results, providing the balance between the number of L2 cache misses and the amount of extra processing caused by partitioning. As the number of partitions grows further, the number of L2 cache misses reduces only marginally. The observed dependency between running time and number of partitions is depicted in the Figure 3.

4.2 Case Study 2: Varying element domain's cardinality

This experiment shows the impact of $|V|$ on the running time of the algorithm. In this case study we only consider the partitioned version of the algorithm, the number of partition is chosen so that $Workmap_p$ fits into the L2 cache, and the cardinalities of relations R and S are selected to keep all the processing in the main memory and avoid trashing as long as possible. We keep $|R| = 100000$, $|S| = 250000$ and $|r| = |s| = 5$ fixed, and vary $|V|$. The less $|V|$, the greater the expected length of inverted lists $|S|P_{v\in s}$. On the other hand, the weight of the term $N_p|V|C_{select}$ decreases with the decrease of $|V|$. The results of the experiment given in the

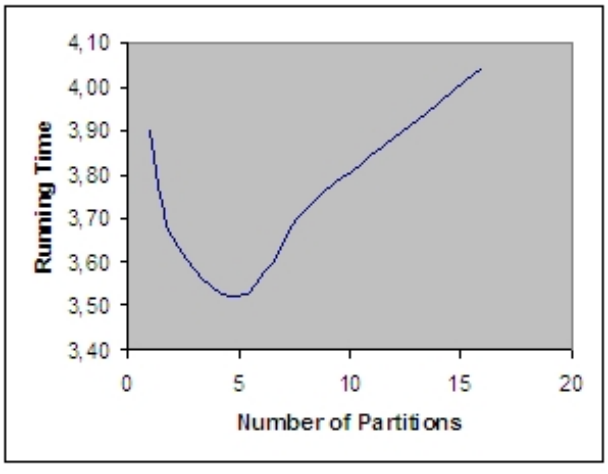

Figure 3: Dependency of running time on number of partitions

$\|V\|$	Time, sec
500	15
250	29
200	37
100	213

Table 3: Impact of element domain cardinality

Table 3 show that the cost of merging the inverted lists dominates in the cost formula (1), and long inverted lists dramatically reduce the performance of the algorithm. This is where compression comes into play and enables to process much larger relations by effectively reducing memory space consumed by inverted lists.

4.3 Case Study 3: Effect of compression

The compression of inverted lists in the intermediate results ($Workmap_p$) helps to reduce number of L2 cache misses while intersecting inverted lists, at the cost of increasing the constant C' in the formula (1). We vary length of the inverted lists by changing average set cardinalities and the size of the domain $|V|$. The cardinalities of relations are kept fixed: $|R| = 100000, |S| = 250000$. The dependency among running time, total memory used by the program (to retrieve this value, the code was instrumented by a call to the $GC.GetTotalMemory$ function from the .NET standard library. This function performs garbage collection before calculating the size of the heap), the average lengths of inverted lists and the cardinality of the result $|Result|$ is illustrated in the Table 4. The table shows results for both versions of the algorithm, with and without compression. The one cell '-' is missed because the algorithm was not able to finish due to exhaustion of all the available memory.

From these results it becomes clear that more compact inverted lists in the 'compressed' version, though reducing the number of L2 cache misses, do not pay off the increase in the cost of intersection operation. So the most important goal for using compression is smaller memory footprint. As the third and the fourth rows of the table demonstrate this effect is achieved only when all l_R, l_S and $|Result|$ are large enough. This fact quite matches the intuitive expectations, since the less the cardinality of the result (in the extreme case of the third row the result is empty), the shorter inverted lists in the intermediate results, and the less benefits we give compressing them. A more sophisticated algorithm could use compression adaptively, predicting the cardinality of the join and keeping track of the size of intermediate results. More detailed development of such an algorithm is left for future work.

4.4 Case Study 4: Comparison with other algorithms

To demonstrate the efficiency of the proposed algorithmm, we have compared its performance with some other algorithms for set-containment joins. Those algorithms include signature nested-loops join (SNL), partitioned set join (PSJ) and the algorithm Index-SCJ based on computing the *intersection index* of the relations which we discussed in [10]. To keep running times reasonable we considered small relations: $|R|$ = 15000, $|S|$ = 25000, $|V|$ = 1000, $|r| = |s|$ = 5, $|Result|$ = 1147. The

Algorithm	Time, sec
SNL	545
PSJ	146
Index-SCJ	12
IFJ	0.3

Table 5: Comparison with other algorithms

results are depicted in the Table 5, where the proposed algorithm is denoted IFJ (Inverted File Join). Though this measurement is not quite representative, it nevertheless shows that the IFJ outperforms competitors on the orders of magnitude. For fair comparison we should note, however, that such a performance is achieved at the cost of significant extra space needed for intermediate results. Among other algorithms, SNL and PSJ have very moderate space requirements, while Index-SCJ also needs much memory for the intersection index.

5 Conclusion

We have proposed an efficient algorithm for set-containment joins in main memory. Our algorithm exploits inverted files which are combined into single join index allowing very efficient traversal of the set elements and the corresponding RIDs. The algorithm maintains a mapping of RIDs of the first relation to the list of RIDs of the second relation each of which contains all the elements of the set of the first relation encountered so far. After the final iteration, this mapping is exactly the join result. We have focused on the cache efficiency and discussed a partitioning method that improves memory references locality of the algorithm. As our experimental evaluation shows, the partitioning reduces the number of L2 cache misses by $10-15\%$. We have also applied compression to inverted lists in the intermediate results. The compression may decrease space requirements of the algorithm 4 times at the cost of performance deterioration. Its effectiveness in large extend depends on such parameters of the input relations as average lengths of the inverted lists in the index and join selectivity.

Future research on this algorithm should focus on applying compression adaptively. An adaptive version may keep inverted lists compressed or not depending on the length of the list, predicted join selectivity and current memory usage. Another possible direction for future work is a parallel algorithm for set-containment joins. The partitioned version of the algorithm can be easily parallelized, since there is no any data dependency between processing of different partitions. Running the algorithm in a multi-processor environment opens new possibilities for improving its response time. A more accurate cost model should be developed, since the current one does not take into account the fact that the inverted lists shrink during processing. We are now working on these problems and hope to present detailed results in the next paper.

References

[1] 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.

			Uncompressed		Compressed	
$\|l_R\|$	$\|l_S\|$	$\|Result\|$	Time, sec	Mem. Usage, Mb	Time, sec	Mem. Usage, Mb
20	50	137	1.5	110	3.2	113
75	188	1759	1.8	113	3.9	90
3000	7500	0	73	144	115	146
500	2250	$50 * 10^6$	-	300	151	91

Table 4: Effect of compression

[2] Sven Helmer and Guido Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the 23rd VLDB Conference*, pages 386–395, 1997.

[3] Sven Helmer and Guido Moerkotte. Compiling away set containment and intersection joins (technical report), 2002.

[4] Nikos Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the SIGMOD 2003 Conference*, pages 157–168, 2003.

[5] Stefan Manegold. The Calibrator, a Cache-Memory and TLB Calibration Tool. *http://homepages.cwi.nl/˜manegold/Calibrator/*.

[6] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. Cache-conscious radix-decluster projections. In *Proceeding of the SIGMOD 2004 Conference*, 2004.

[7] Sergey Melnik and Hector Garcia-Molina. Adaptive Algorithms for Set Containment Joins. *ACM Transactions on Database Systems*, 28:56–99, 2003.

[8] Karthikeyan Ramasamy et al. Set containment joins: The good, the bad and the ugly. In *Proceedings of the 26th VLDB Conference*, pages 351–362, 2000.

[9] Dmitry Shaporenkov. Multi-indices - a tool for optimizing join processing in main memory. In *Proceedings of the Baltic DBIS 2004 Conference*, pages 105–114, 2004.

[10] Dmitry Shaporenkov. Performance comparison of main-memory algorithms for set containment joins. In *Proceedings of the SYRCoDIS'04*, pages 17–21, 2004.

[11] 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.

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

[13] Ian Witten, Alistair Moffat, and Timothy Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann publishers, second edition, 1999.

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

[15] Intel VTune Performance Analyzer. *http://www.intel.com/software/products/vtune/*.