Efficient Processing of Joins on Set-valued Attributes

Nikos Mamoulis
Department of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road
Hong Kong
nikos@csis.hku.hk

Abstract

Object-oriented and object-relational DBMS support set-valued attributes, which are a natural and concise way to model complex information. However, there has been limited research to-date on the evaluation of query operators that apply on sets. In this paper we study the join of two relations on their set-valued attributes. Various join types are considered, namely the set containment, set equality, and set overlap joins. We show that the inverted file, a powerful index for selection queries, can also facilitate the efficient evaluation of most join predicates. We propose join algorithms that utilize inverted files and compare them with signature-based methods for several set-comparison predicates.

1. Introduction

Commercial object-oriented and object relational DBMS [14] support set-valued attributes in relations, which are a natural and concise way to model complex information. Although sets are ubiquitous in many applications (document retrieval, semi-structured data management, data mining, etc.), there has been limited research on the evaluation of database operators that apply on sets.

On the other hand, there has been significant research in the IR community on the management of set-valued data, triggered by the need for fast content-based retrieval in document databases. Research in this area has focused on processing of keyword-based selection queries. A range of indexing methods has been proposed, among which signature-based techniques [2, 3, 9] and inverted files [16, 17] dominate. These indexes have been extensively revised and evaluated for various selection queries on set-valued attributes [15, 8].

An important operator which has received limited attention is the join between two relations on their set-valued attributes. Formally, given two relations R and S, with set-valued attributes $R.r$ and $S.s$, $R \bowtie_{r\theta s} S$ returns the subset of their Cartesian product $R \times S$, in which the set-valued attributes qualify the join predicate θ. Two join operators that have been studied to date are the *set containment* join, where θ is $\subseteq$ [7, 13], and the *all nearest neighbors* operator [11], which retrieves for each set in R its k nearest neighbors in S based on a similarity function. Other operators include the *set equality* join and the *set overlap* join.

As an example of a set containment join, consider the join of a job-offers relation R with a persons relation S such that $R.required\_skills \subseteq S.skills$, where $R.required\_skills$ stores the required skills for each job and $S.skills$ captures the skills of persons. This query will return qualified person-job pairs. As an example of a set overlap join, consider two bookstore databases with similar book catalogs, which store information about their clients. We may want to find pairs of clients which have bought a large number of common books in order to make recommendations or for classification purposes.

In this paper we study the efficient processing of set join operators. We assume that the DBMS stores the set-valued attribute in a *nested* representation (i.e., the set elements are stored together with the other attributes in a single tuple). This representation, as shown in [13], facilitates efficient query evaluation. We propose join algorithms that utilize inverted files and compare them with signature-based methods for several set-comparison predicates. Our contribution is two-fold:

- We propose a new algorithm for the set containment join, which is significantly faster than previous approaches [7, 13]. The algorithm builds an inverted file for the "container" relation S and uses it to process the join. We demonstrate the efficiency and robustness of this method under various experimental settings.
- We discuss and validate the efficient processing of other join predicates, namely set equality and set overlap joins. For each problem, we consider alternative methods based on both signatures and inverted files. Experiments show that using an inverted file to perform the join is also the most appropriate choice for overlap joins, whereas signature-based methods are competitive only for equality joins.

The rest of the paper is organized as follows. Section 2 provides background about set indexing methods. Section 3 describes existing signature-based algorithms for the set containment join and presents new techniques that employ inverted files. Section 4 discusses how signature-based and inverted file join algorithms can be adapted to process set equality and set overlap joins. The performance of the algorithms is evaluated experimentally in Section 5. Finally, Section 6 concludes the paper with a discussion about future work.

2. Background

In this section we provide a description for various methods used to index set-valued attributes. Most of them originate from Information Retrieval applications and they are used for text databases, yet they can be seamlessly employed for generic set-valued attributes in object-oriented databases.

2.1 Signature-based Indexes

The *signature* is a bitmap used to represent sets, exactly or approximately. Let D be the (arbitrarily ordered) domain of the potential elements which can be included in a set, and $|D|$ be its cardinality. Then a set x can be represented exactly by a $|D|$-length signature $sig(x)$. For each i, $1 \leq i \leq |D|$, the i-th bit of $sig(x)$ is 1 iff the i-th item of D is present in x. Exact signatures are expensive to store if the sets are sparse, therefore approximations are typically used; given a fixed *signature length* b, a mapping function assigns each element from D to a bit position (or a set of bit positions) in $[0, b)$. Then the signature of a set is derived by setting the bits that correspond to its elements.

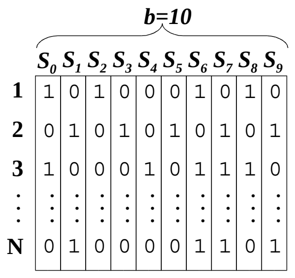
Queries on signature approximations are processed in two steps (in analogy to using rectangle approximations of objects in spatial query processing [5]). During the *filter step*, the signatures of the objects are tested against the query predicate. A signature that qualifies the query is called a *drop*. During the *refinement step*, for each drop the actual set is retrieved and validated. If a drop does not pass the refinement step, it is called a *false drop*.

Assume, for example, that the domain D comprises the first 100 integers and let $b = 10$. Sets $x = \{38, 67, 83, 90, 97\}$ and $y = \{18, 67, 70\}$ can then be represented by signatures $sig(x) = 1001000110$ and $sig(y) = 1000000110$ if modulo 10 is used as a mapping function. Checking whether $x = y$ is performed in two steps. First we check if $sig(x) = sig(y)$, and only if this holds we compare the actual sets. The same holds for the subset predicate ($x \subseteq y \Rightarrow sig(x) \subseteq sig(y)$) and the simple overlap predicate (e.g., whether x and y overlap or not). The signatures are usually much smaller than the set instances and binary operations on them are very cheap, thus the two-step query processing can save many computations. In the above example, since $sig(x) \neq sig(y)$, we need not validate the equality on the actual set instances. On the other hand, the object pair $\langle x, y \rangle$ passes the filter step for $x \subseteq y$, but it is a false drop. Finally, for $x \cap y \neq \varnothing$ $\langle x, y \rangle$ passes both the filter and refinement steps. Naturally, the probability of a false drop decreases as b increases. On the other hand, the storage and comparison costs for the signatures increases, therefore a good trade-off should be found.

For most query predicates the signatures can serve as a fast preprocessing step, however, they can be inefficient for others. Consider for instance the query $|x \cap y| \geq \epsilon$, called ϵ-*overlap* query and denoted by $x \cap_\epsilon y$ in the following, asking whether x and y share at least ϵ common items. Notice that the signatures can be used to prune only sets for which $sig(x) \wedge sig(y) = 0$ (the wedge here denotes logical AND). Thus, if $\epsilon > 1$ the signatures are not prune-effective. Consider the running example instances x and y and assume that $\epsilon = 2$. The fact that $sig(x)$ and $sig(y)$ have three common bits does not provide a lower bound for the actual overlap, since different elements can map to the same bit. As another example, assume that $x = \{18, 38, 68\}$ and $y = \{18, 68, 70\}$. The signatures now share just one bit, but they qualify the query. This shows why we cannot prune signatures with overlap smaller than ϵ.

Signatures have been used to process selection queries on set-valued attributes. The most typical query is the *set containment query*, which given a set q and a relation R, asks for all sets $R.r \in R$ such that $q \subseteq r$. As an example application, assume that we are looking for all documents which contain a set of index terms. Usually, signatures are organized into indexes to further accelerate search. The simplest form of a signature-based index is the signature file [3]. In this representation the signatures of all sets in the relation are stored sequentially in a file. The file is scanned and each of them is compared with the signature of the query. This is quite efficient if the query selectivity is low (i.e., if the percentage of the qualifying signatures is large), but it is not an attractive method for typical applications with sparse sets.

An improved representation of the signature file employs *bit-slicing* [9]. In this representation, there is a separate bit-slice vector stored individually for each bit of the signatures. When applying query q the bit slices where $sig(q)$ has 1 are ANDed to derive a bit slice, representing the record-ids (rids) that pass the filter step (for containment and equality queries). Figure 1 shows an example of a bit-sliced signature file. If the query signature is 1001000100, we need to join slices S_0, S_3, and S_7 in order to get the candidate rids. If the partial join of some slices sets only a few rids, it might be beneficial not to retrieve and join the remaining slices, but to validate q directly on the qualifying rids. Overlap queries can be processed by taking the logical OR of the bit-vectors.



	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
1	1	0	1	0	0	0	1	0	1	0
2	0	1	0	1	0	1	0	1	0	1
3	1	0	0	0	1	0	1	1	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
N	0	1	0	0	0	0	1	1	0	1

Figure 1: A bit-sliced signature file

The signature-tree [2, 10] is a hierarchical index, which clusters signatures based on their similarity. It is a dynamic, balanced tree, similar to the R–tree [6], where each node entry is a tuple of the form $\langle sig, ptr \rangle$. In a leaf node entry *sig* is the signature and *ptr* is the record-id of the indexed tuple. The signature of a directory node entry is the logical OR of the signatures in the node pointed by it and *ptr* is a pointer to that node. In order to process a containment/equality query we start from the root and follow the entry pointers *e.ptr*, if $sig(q) \subseteq e.sig$, until the leaf nodes where qualifying signatures are found. For overlap queries all entries which intersect with $sig(q)$ have to be followed. This index is efficient when there are high correlations between the data signatures. In this case, the signatures at the directory nodes are sparse enough to facilitate search. On the other hand, if the signatures are random and uncorrelated the number of 1's in the directory node entries is high and search performance degrades. A strength of the index is that it handles updates very efficiently, thus it is especially suitable for highly dynamic environments.

Another method for indexing signatures is based on hashing [8]. Given a small k, the first k bits are used as hash-keys and the signatures are split into 2^k partitions. The first k bits of the query are then used to select the buckets which may contain qualifying results. To facilitate locating the qualifying buckets, a directory structure which contains hash values and pointers is stored separately (even in memory if the number of buckets is small enough). In order to control the size of the directory and partitions dynamically, extendible hashing can be used. This index is very efficient for equality selection queries, since only one bucket has to be read for each query. The effect is the same as if we sorted the signatures and imposed a B$^+$–tree over them, using a signature prefix as key. However, the performance of this index is not good for set containment queries [8].

2.2 The Inverted File

The inverted file is an alternative index for set-valued attributes. For each set element el in the domain D, an *inverted list* is created with the record ids of the sets that contain this element in sorted order. Then these lists are stored sequentially on disk and a directory is created on top of the inverted file, pointing for each element to the offset of its inverted list in the file. If the domain cardinality $|D|$ is large, the directory may not fit in memory, so its entries $\langle el, offset\rangle$ are organized in a B$^+$–tree having el as key.

Given a query set q, assume that we want to find all sets that contain it. Searching on the inverted file is performed by fetching the inverted lists for all elements in q and merge-joining them. The final list contains the record-ids of the results. For example, if the set containment query $q = \{el_2, el_3\}$ is applied on the inverted file of Figure 2, only tuple with rid=132 qualifies. Since the record-ids are sorted in the lists, the join is performed fast. If the search predicate is 'overlap', then the lists are just merged, eliminating duplicates. Notice that the inverted file can also process ϵ-overlap queries efficiently since we just need to count the number of occurrences of each record-id in the merged list.

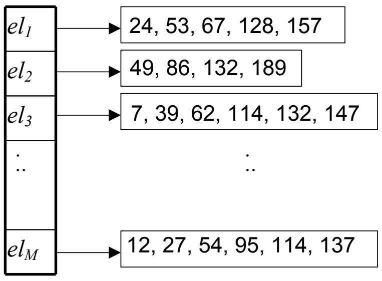


Figure 2: An inverted file

Fetching the inverted lists can incur many random accesses. In order to reduce the access cost, the elements of q are first sorted and the lists are retrieved in this order. Prefetching can be used to avoid random accesses if two lists are close to each other.

Updates on the inverted file are expensive. When a new set is inserted, a number of lists equal to its cardinality have to be updated. An update-friendly version of the index allows some space at the end of each list for potential insertions and/or distributes the lists to buckets, so that new disk pages can be allocated at the end of each bucket. The inverted file shares some features with the bit-sliced signature file. Both structures merge a set of rid-lists (represented in a different way) and are mostly suitable for static data with batch updates.

In recent comparison studies [15, 8], the inverted file was proven significantly faster than signature-based indexes, for set containment queries. There are several reasons for this. First, it is especially suitable for real-life applications where the sets are sparse and the domain cardinality large. In this case, signature-based methods generate many false drops. Second, it applies exact indexing, as opposed to signature-based methods, which retrieve a superset of the qualifying rids and require a refinement step. Finally, as discussed in the next subsection, it achieves higher compression rates, than signature-based approaches and its performance is less sensitive to the decompression overhead.

2.3 Compression

Both signature-based indexes and inverted files can use compression to reduce the storage and retrieval costs. Dense signatures are already compressed representations of sets, but sparse ones could require more storage than the sets (e.g., the exact signature representations of small sets). Thus, a way to compress the signatures is to reduce their length b at the cost of increasing the false drop probability. Another method is to store the element-ids instead of the signature for small sets or to encode the gaps between the 1's instead of storing the actual bitmap. However, this technique increases the cost of query processing; the signature has to be constructed on-the-fly if we want to still use fast bitwise operations. The bit-sliced index can also be compressed by encoding the gaps between the 1's, but again this comes at the expense of extra query processing cost [15].

The compression of the inverted file is much more effective. An inverted list is already sorted and can be easily compressed by encoding the gaps (i.e., differences) between consecutive rids. For example, assume that an inverted list contains the following rids: $\{11, 24, 57, 102, 145, 173, ...\}$. It can be converted to the *run-length encoding* of the rids: $\{11, 13, 33, 45, 43, 28, ...\}$, i.e., the series of differences between successive rids [16]. Observe that this comes at no cost for query processing, since the algorithm for merging the lists can be easily adapted. Although the above list contains small numbers there is no upper bound for them; a fixed-length encoding would yield no compression. Thus a variable-length encoding should be used.

An example of such an encoding is the Golomb coding [4]. Assume that the integers we want to encode follow a geometric law with a probability parameter p. The Golomb code takes a parameter g such that $(1-p)^g + (1-p)^{g+1} \leq 1 < (1-p)^{g-1} + (1-p)^g$. Now assume that we want to encode an integer x, where $x > 1$. In order to save code space, we encode $x - 1$ which is potentially 0. We represent $(x-1)$ using two integers: $l = \lfloor (x-1)/g \rfloor$ and $r = (x-1)$ modulo g. Finally, x is encoded using a unary encoding for $l + 1$ followed by a binary encoding of r. Table 1 shows the run-length integers in our running example and their Golomb codes for $g = 16$. For example, to compute the code of 45 we derive $l = \lfloor(44/16)\rfloor = 2$ and $r = 44$ modulo $16 = 12$, and concatenate the unary $3 = 110$ with the binary $12 = 1100$. Additional coding techniques are also applicable. See [16, 15] for more details on compressing inverted files.

number	code
11	01011
13	01101
33	1100000
45	1101100
43	1101010
28	101011

Table 1: Example of Golomb codes for integers

Decoding the rids is quite fast and affects little the performance of the inverted file. Moreover, the compression rate increases with the density of the list, as the gaps between the rids decrease. To exemplify this, consider a relation R of 100,000 tuples, where each set $R.r$ has 20 elements on the average, and the set domain D has 10,000 elements. The number of rids per inverted list is $\frac{|R|\times|R.r|}{|D|} = 200$ on the average. This means that the expected difference between two rids in an inverted list is $|R|/200 = 500$. If we use Golomb coding with $g = 128$ we can encode each rid with 10-11 bits on the average. Now assume that each set has 200 elements on the average. In this case, the expected difference between two rids is just 50 and we need fewer bits to encode them. Thus the size (and retrieval cost) of the inverted file increases sub-linearly with the average set cardinality. Moreover, the Golomb coding can be applied independently for each inverted list adjusting the compression rate according to the data skew. The two extreme cases are lists containing few uncompressed integers, and lists which reduce to bitmaps.

3. Evaluation of Set Containment Joins

Several join operators can be applied on set-valued attributes, including the set containment, the set equality and the set overlap join. In this section we restrict our attention to the set containment join, which is one of the most interesting operators and also has been proven rather hard in past research [7, 13]. We first describe existing signature-based join algorithms. Then we propose and discuss the optimization of new algorithms that utilize inverted files. Throughout this section we will deal with the join $R \bowtie_{r \subseteq s} S$, where r and s are set-valued attributes of relations R and S, respectively. Unless otherwise specified, we will assume that none of the relations is indexed.

3.1 The Signature Nested Loops Join

In [7] two main memory join algorithms were proposed for the set containment join. One of them is also applicable for large relations that do not fit in memory. The *Signature Nested Loops* (SNL) Join consists of three phases. During the *signature construction* phase, for each tuple $t_S \in S$ a triplet $\langle |t_S.s|, sig(t_S.s), t_S.rid\rangle$ is computed, where $|t_S.s|$ is the cardinality of $t_S.s$, $sig(t_S.s)$ is its signature, and $t_S.rid$ is the record-id of t_S. These triplets are stored in a signature file S_{sig}. During the *probe* phase, each tuple $t_R \in R$ is read, its signature $sig(t_R.r)$ is computed and the signature file S_{sig} is scanned to find all pairs $\langle t_R.rid, t_S.rid\rangle$, such that $sig(t_R.r) \subseteq sig(t_S.s)$ and $|t_R.r| \leq |t_S.s|$. These rid-pairs are written to a temporary relation. During the final *verification* phase the $\langle t_R.rid, t_S.rid\rangle$ pairs generated from the probe phase are read, the corresponding tuples are fetched, and $t_R.r \subseteq t_S.s$ is verified.

This method requires a quadratic number (to the relation size) of signature comparisons and is not attractive for large problems. Notice that the verification phase could be combined with the match phase if we fetch $t_S.s$ immediately after the signature comparison succeeds, however, this would incur many random accesses. In [7] a hash-based method is also proposed, however, it would result in a large number of random I/Os if applied to secondary memory problems.

3.2 The Partitioned Set Join

Ramasamy et al. [13] introduced a hash-based algorithm that aims at reducing the quadratic cost of SNL. The *Partitioned Set Join* (PSJ) hashes the tuples from R and S to a number of P buckets $R_1, R_2, \ldots, R_P$ and $S_1, S_2, \ldots, S_P$ and joins R_x with S_x for all $1 \leq x \leq P$.

The algorithm also works in three phases. During the first (*partitioning*) phase, the tuples from the two relations are partitioned. Each tuple $t_R \in R$ is read, and the triplet $\langle |t_R.r|, sig(t_R.r), t_R.rid\rangle$ is computed. A *random* element e_r from $t_R.r$ is picked and the triplet is written to the bucket determined by a hash function $h(e_r)$. Then relation S is partitioned as follows. Each tuple $t_S \in S$ is read, and the triplet $\langle |t_S.s|, sig(t_S.s), t_S.rid\rangle$ is computed. *For each* element e_s in $t_S.s$, the triplet is written to the partition determined by the same hash function $h(e_s)$. If two elements $e_s \in t_S.s$ have the same hash value the triplet is sent to the corresponding bucket only once. Thus, each triplet from R is hashed to exactly one partition, but a triplet from S is in general hashed to many partitions. The algorithm does not miss join results, because if $t_R.r \subseteq t_S.s$ then the random element $e_r \in t_R.r$ will equal some $e_s \in t_S.s$, and the triplets will co-exist in the partition determined by $h(e_r)$.

During the *join* phase, the pairs of partitions $\langle R_x, S_x\rangle$ corresponding to the same hash value x are loaded and joined. R_x should be small enough to fit in memory; its contents are loaded and hashed using a secondary hash function. More specifically, a random set bit in each $sig(t_R.r) \in R_x$ is picked and the triplet is assigned to the corresponding memory bucket. Then each triplet from S_x is compared with all triplets in the memory buckets determined by the set bits in $sig(t_S.s)$. Qualifying rid-pairs are written to a temporary file. Finally, the *verification* phase of PSJ is the same as that of SNL.

Although PSJ is faster than SNL, if suffers from certain drawbacks. First, the replication of the signatures from S can be high if the set cardinality is in the same order as the number of partitions P. Thus if c is the cardinality of a $t_S.s \in S$, the tuple will be hashed into $c - c(1 - 1/c)^P$ partitions on the average [7, 13]. For example if $P = 100$ and $c = 50$, the quantity above shows that each tuple is replicated to 43.3 partitions on the average. Thus the size of each S_x is comparable to S_{sig} and the algorithm has similar cost to SNL (actually to a version of SNL that employs in-memory hashing; called PSJ-1 in [13]). On the other hand, P should be large enough in order to partition R into buckets that fit in memory. Second, PSJ carries the inherent drawback of signature-based evaluation methods; it introduces false drops, which have to be verified. The verification phase incurs additional I/O and computational overhead which constitutes a significant portion of the overall join cost, as shown in [13] and validated by our experiments. In the rest of this section we introduce join algorithms based on inverted files that alleviate these problems.

3.3 Block Nested Loops Using an Inverted File

As shown in [15], inverted files are more efficient than signature-based indexes for evaluating selection queries. Among their other advantages once compression is taken into account, it is mentioned that they are also relatively cheap to create.

This motivated us to study alternative methods for the set containment join, based on inverted files. Even if the relations are not indexed, it could be more beneficial to build inverted files and perform the join using them, rather than applying signature-based join techniques. Hopefully, this would yield two advantages. First, the join will be processed fast. Second, we would have built a powerful index that can be used afterwards not only for other join queries, but also for selections on the set-valued attribute, as described in Section 2. We will now discuss how to build inverted files fast and use them for evaluating set containment joins.

3.3.1 *Constructing an inverted file*

The inverted file for a relation R can be constructed fast using sorting and hashing techniques [12]. A simple method is to read each tuple $t_R \in R$ and decompose it to a number of binary $\langle el, t_R.rid\rangle$ tuples, one for each element-id el that appears in the set $t_R.r$. Then these tuples are sorted on el and the index is built. For small relations this is fast enough, however, for larger relations we have to consider techniques that save I/O and computational cost. A better method is to partition the binary tuples using a hash function on el and build a part of the index for each partition individually. Then the indexes are concatenated and the same hash function is used to define an order for the elements.

In order to minimize the size of the partitions, before we write the partial lists of the partitions to disk, we can sort them on $t_R.rid$ and compress them. This comes at the additional cost of decompressing them and sort the tuples on el to create the inverted lists for the partition. An alternative method is to sort the partial lists on el in order to save computations when sorting the whole partition in memory.

The cost of building the inverted file comprises the cost of reading the relations, the cost of writing and reading the partitions and the cost of writing the final inverted file. As we demonstrate in Section 5, if compression is used, the overall cost is not much larger than the cost of reading R. This is due to the fact that the compressed partial information is several times smaller than the actual data.

3.3.2 *A simple indexed nested-loops join method*

The simplest and most intuitive method to perform the set containment join using inverted files is to build an inverted file S_{IF} for S and apply a set containment query for each tuple in R. It is not hard to see that if S_{IF} fits in memory, this method has optimal I/O cost.

However, for large problems S_{IF} may not fit in memory and in the worst case it has to be read once for each tuple in R. Therefore, we should consider alternative techniques that apply in the general case.

3.3.3 *Block Nested Loops Join*

Since it is not efficient to scan the inverted file for each tuple from R we came up with the idea of splitting the inverted file in large blocks that fit in memory and scan R for each block. Our *Block Nested Loops* (BNL) algorithm reads S_{IF} sequentially in blocks $B_1, B_2, \dots, B_N$, such that each block fits in memory and there is enough space for loading a page from R. Assume that currently block B_i is loaded in memory, and let L_i be the set of elements whose inverted list is in B_i. For each tuple $t_R \in R$, three cases may apply:

1. $t_R.r \subseteq L_i$; the lists of all elements in $t_R.r$ are in B_i. In this case the lists are joined and the qualifying results are output.
2. $t_R.r \cap L_i = \varnothing$; the list of no element in $t_R.r$ is in B_i. In this case t_R is ignored and we go to the next tuple.
3. $t_R.r \cap L_i \neq \varnothing \wedge t_R.r \nsubseteq L_i$; the lists of some (but not all) elements in $t_R.r$ are in B_i. This is the hardest case to handle, since we may need information from other blocks in order to verify whether t_R has any superset in S. We call t_R a *dangling* tuple.

The first two cases are trivial; thus we concentrate on handling dangling tuples efficiently. A simple way to manage a dangling tuple t_R is to merge the lists in $t_R.r \cap L_i$ (these are currently in memory), compute a *partial rid list*, and, if non-empty, write it to a temporary file T_i. The partial lists are of the form $\langle t_R.rid, n, t_{S1}.rid, t_{S2}.rid, \dots t_{Sn}.rid\rangle$, where n is the number of tuples from S that currently match with $t_R.rid$. If n is large, the list is compressed to save space. After we have scanned S_{IF}, a second phase begins, which loads the temporary files and merge-joins them. The contents of each T_i are already sorted on $t_R.rid$, thus merging them in memory is very efficient.

Figure 3 shows the pseudocode of BNL. l_{el} denotes the inverted list of element el. In the implementation, L_i is not computed explicitly, but the number m of elements $el \in t_R.r \wedge l_{el} \in B_i$ is used to validate whether $t_R.r \subseteq L_i$; if $|t_R.r|$ equals m we know that all elements of $t_R.r$ are in L_i. As a final comment, we assume that the directory for each B_i can fit in memory, since it should be much smaller than the block itself. We further compress it, by storing the offsets of each inverted list, with respect to the absolute offset of the first list in B_i.

3.3.4 *Optimizing BNL*

The partial lists are many if the memory blocks B_i are not large, and so is the overhead of writing (and reading) temporary results to disk. Therefore we have to minimize this temporary information at no additional cost in the join process.

3.3.4.1 *Keeping $\pi_{S.|s|}$ in memory.*

A fact that we have not exploited yet, is that a tuple $t_R \in R$ could join with a tuple $t_S \in S$ only if the cardinalities of the sets qualify $|t_R.r| \leq |t_S.s|$. If we use this information, we could shrink or prune many partial lists. The basic version of the inverted file does not include any information about $|t_S.s|$. We can obtain this information in two ways. The first is to store $|t_S.s|$ with every appearance of $t_S.rid$ in the inverted lists. This method is also used in [8]. However, the size of the inverted file may increase dramatically (it may

```
Algorithm $BNL(R,\ S_{IF})$ {
  $i := 0$;
  while there are more lists in $S_{IF}$ {
    $i := i + 1$;
    $B_i$ := read next block of $S_{IF}$ that fits in memory;
    $L_i$ := elements whose inverted list is in $B_i$;
    Initialize temporary file $T_i$;
    for each tuple $t_R \in R$ do {
      $list(t_R) := \bigcap_{l_{el}},\ el \in t_R.r \wedge el \in L_i$;
      if $list(t_R) \neq \varnothing$ then
        if $t_R.r \subseteq L_i$ then output results;
        else write $list(t_R)$ to $T_i$;
    }
  }
  $n := i$;
  merge-join all $T_i$, $1 \leq i \leq n$ and output results;
}
```

Figure 3: Block-Nested Loops join

even double). In a compressed inverted file, we only need few bits (e.g., 10) to encode each $t_S.rid$, and storing $|t_S.s|$ could require that many extra bits. On the other hand, we observe that in typical applications, where sets are small, $|t_S.s|$ does not require more than a byte to store. Thus we can keep the cardinalities of all $t_S.s \in S$ in memory (we denote this table by $\pi_{S.|s|}$) without significant overhead. For example, if $|S|$=1M tuples, we need just a megabyte of memory to accommodate $\pi_{S.|s|}$. This information can be computed and stored in memory while reading S at the construction phase of S_{IF}. It is then used to (i) accelerate the merging of inverted lists, (ii) reduce the number and length of the partial lists that have to be written to temporary files.

3.3.4.2 *Pipelining the partial lists.*

Even with the availability of $\pi_{S.|s|}$, the partial lists could be many and long. If an object's elements are spanned to multiple blocks, multiple lists may have to be written to the temporary files. To make things worse, an object which is pruned while joining block B_i with R, because its lists there have empty intersection, may qualify at a later block B_j, $i < j$, although it should never be considered again. For example, assume that a $t_R \in R$ has two elements in B_1 and their inverted lists have empty intersection. This means that there is no set in S containing both elements, and as a result no set in S joins with t_R. When B_2 is loaded, we can find some elements in t_R with lists in B_2, which should not normally be processed, since we already know that t does not join with any tuple from S.

Another observation is that while processing B_2, we can merge the partial lists generated by it with the lists in T_1, i.e., the temporary file created by the previous pass. Thus during the iteration where we join B_i with R, we keep in memory (i) block B_i, (ii) one (the current page) from R, and (iii) one (the current) page from T_{i-1}. Since we scan R sequentially and the lists in T_{i-1} are sorted on $t_R.rid$, we can merge efficiently the lists produced by the previous passes with the ones produced by the current one. This makes it easier also to discard all t_R which disqualified at the previous steps; if there is any element $el \in t_R.r$ whose inverted list l_{el} was at a previous B_j, $j < i$ and there is no list in T_{i-1} for $t_R.rid$, then we know that t_R has been pruned before and we can ignore it at the current and next blocks. Another way to avoid processing tuples which disqualified during a previous block-join is to maintain a memory-resident bitmap indicating the pruned tuples from R.

The pseudocode of this version of BNL that employs pipelining is shown in Figure 4. At each step, the qualifying lists from the current block $B \in S_{IF}$ are merged with the temporary file T_{prev} from the previous step. To perform merging efficiently we run a pointer on T_{prev} showing the current list l_p. This serves multiple purposes. First, each $t_R \in R$ which disqualified in some previous step is easily spotted (condition at line 1). Indeed, if the inverted list of some element from t_R has been loaded before (this can be easily checked using the order of the elements in $t_R.r$) and $t_R.rid$ does not match the current $l_p.t_R.rid$ in T_{prev}, then we know that $t_R.rid$ has been pruned before, because otherwise there would be an entry for it in T_{prev}. The second use of l_p is to join its contents with the lists in B. If the current t_R has elements in the current block and matches with the current l_p, the lists in B are joined with l_p. If the resulting list is not empty, the results are either output, or written to the next temporary file T_{next}, depending on whether there are elements in t_R yet to be seen at a next step (cf. line 4 in Figure 4). l_p is updated after the corresponding tuple $l_p.t_R$ is read from R (conditions at lines 2 and 3).

```
    Algorithm $BNL(R,\ S_{IF})$ {
      $T_{prev}$ := NULL;
      while there are more lists in $S_{IF}$ {
        $B$ := read next block of $S_{IF}$ that fits in memory;
        $L$ := elements whose inverted list is in $B$;
        Initialize temporary file $T_{next}$;
        $l_p$ := get next list from $T_{prev}$; /*if applicable*/
        for each tuple $t_R \in R$ do {
(1)       if some $el \in t_R.r$ was in some previous $B$
               and $l_p.t_R.rid \neq t_R.rid$ then
            continue; /*prune this tuple*/
          if $t_R.r \cap L = \varnothing$ then {
(2)         if $l_p.t_R.rid = t_R.rid$ then
              $l_p$ := get next list from $T_{prev}$; /*if applicable*/
            continue; /*goto next tuple*/
          }
          $list(t_R) := \bigcap_{l_{el}},\ el \in t_R.r \wedge el \in L_i$;
(3)       if $l_p.t_R.rid = t_R.rid$ then {
            $list(t_R) := list(t_R) \cap l_p$;
            $l_p$ := get next list from $T_{prev}$; /*if applicable*/
          }
          if $list(t) \neq \varnothing$ then
(4)         if all inv. lists for $t_R.r$ have been seen so far then
              output results;
            else write $list(t_R)$ to $T_{next}$;
        }
        $T_{prev} := T_{next}$;
      }
    }
```

Figure 4: Block-Nested Loops with pipelining

3.3.4.3 *Optimizing the computational cost.*

Another factor which requires optimization is the join algorithm that merges the inverted lists in memory. We initially implemented a multiway merge join algorithm with a heap (i.e., priority queue). The algorithm scans all lists synchronously increasing all pointers if all rids match, or the smallest one if they do not. The current positions of all pointers were organized in the heap to facilitate fast update of the minimum pointer. However, we found that this version performed rather slow, whereas most of the joined lists had high selectivity or gave no results.

Therefore, we implemented another version that performs a binary join at a time and joins its results with the next list. We experienced great performance improvement with this version, since since many sets from R with large cardinality were pruned much faster.

3.3.5 *Analysis of BNL*

The cost of BNL heavily depends on the size of S_{IF}. Clearly, given a memory buffer M, the smaller S_{IF} is, the fewer the number of passes on R required by the algorithm. Also the more the inverted lists in a block, the fewer the temporary partial lists generated by each pass. The number of bits to store a compressed inverted file I that uses Golomb coding is in the worst case (assuming each inverted list has the same number of rids) given by the following formula [15]:

$$bits(I) = cN(1.5 + \log_2 \frac{|D|}{c}), \tag{1}$$

where N is the number of indexed tuples, c is the average set cardinality and $|D|$ is the domain size of the set elements. The factor in the parentheses is the average number of bits required to encode a "gap" between two rids in an inverted list. We can use this formula to estimate the size of S_{IF}. The I/O cost of BNL can then be estimated as follows:

$$IO_{BNL} = size(S_{IF}) + size(R)\lceil \frac{size(S_{IF})}{M} \rceil + 2\sum size(T_i) \tag{2}$$

The last component in Equation 2 is the cost of reading and writing the temporary results T_i for each block B_i. Assuming that the simple version of BNL is applied (without pipelining), all T_i will be equal, each containing the partial lists for the dangling tuples in B_i. Assuming that rids are uniformly distributed in the inverted list, the probability $Prob_{dng}(t_R)$ of a $t_R \in R$ to become a dangling tuple can be derived by the probability that all elements of t_R to be contained in L_i and the probability that no element in t_R is part of L_i, or else:

$$Prob_{dng}(t_R) = 1 - \frac{\binom{|L_i|}{c} + \binom{|D|-|L_i|}{c}}{\binom{|D|}{c}} \tag{3}$$

L_i can be estimated by $\frac{size(S_{IF})}{M}$ since we assume that all lists have the same size. The number of dangling tuples can then be found by $|R|Prob_{dng}(t_R)$. Not all dangling tuples will be written to T_i, since some of them are pruned. The expected number of qualifying dangling tuples, as well as the size of their partial lists can be derived by applying probabilistic counting, considering the expected lengths of the lists and other parameters. The analysis is rather long and complex, therefore we skip it from the paper.

As shown in Section 5, the size of the temporary results is small, especially for large c, where the chances to prune a tuple are maximized. Moreover, the pipelining heuristic decreases fast the temporary results from one iteration to the next. For our experimental instances, where M is not much smaller than S_{IF} (e.g., $M \geq S_{IF}/10$), T_i is typically a small fraction of S_{IF}, thus the I/O cost of BNL reduces roughly to the cost of scanning R times the number of passes.

3.4 An algorithm that joins two inverted files

The set containment join can also be evaluated by joining the two inverted files R_{IF} and S_{IF}. Although this may require preprocessing both relations, at a non-negligible cost, the resulting inverted files may accelerate search, and at the same time compress the information from the original relations.

The join algorithm traverses synchronously both inverted files. At each step, the inverted lists are merged to partial lists $\langle t_R.rid, n, t_{S1}.rid, t_{S2}.rid, \dots t_{Sn}.rid\rangle$, where n is the number of tuples from S that match so far with $t_R.rid.$ Each inverted list pair generates as many partial lists as the number of tuples in the list of R_{IF}. After scanning many inverted list pairs, the number of generated partial lists exceeds the memory limits and the lists are merged. The merging process eliminates lists with empty intersection and outputs results if the number of merged lists for a tuple t_R is equal to $|t_R.r|$. The rest of the merged lists are compressed and written to a temporary file. In the final phase, the temporary files are loaded and joined. We call this method *Inverted File Join* (IFJ).

IFJ can be optimized in several ways. First, the projection $\pi_{R.|r|}$ can be held in memory, to avoid fetching $|t_R.r|$ for verification. Alternatively, this information could be embedded in the inverted file but, as discussed above, at the non-trivial space cost. Another, more critical optimization is to minimize the size of the temporary files. For this, we use similar techniques as those for the optimization of BNL. We keep $\pi_{S.|s|}$ in memory, and use it to prune fast rid pairs, where $|t_R.r| > |t_S.s|$. Another optimization is to use the pipelining technique. Before we output the partial lists, the previous temporary file T_{prev} is loaded and merged with them. This may prune many lists. In addition, if we know that $t_R.r$ has no elements in inverted lists not seen so far, we can immediately output the $\langle t_R.rid, t_{Si}.rid\rangle$ pairs, since they qualify the join.

However, this comes at the cost of maintaining an additional table R_{count}, which counts the number of times each $t_R.rid$ has been seen in the inverted lists we have read so far from R_{IF}. Since the sets are relatively sparse, we will typically need one character per tuple to encode this information. In other words, the size of this array is at most the size of $\pi_{R.|r|}$ and can be typically maintained in memory. R_{count} is initialized with 0s. Whenever we read an inverted list l in R_{IF} we add 1 for each $t_R.rid$ present in l. After a partial list has been finalized, we output the results if $R_{count}[t_R.rid] = |t_R.r|$, otherwise we write the list to the temporary file. An additional optimization uses the equivalent table S_{count} for S, which can be used to prune some $t_{Si}.rid$ from the partial lists, as follows. If $|t_R.r| - R_{count}[t_R.rid] > |t_{Si}.s| - S_{count}[t_{Si}.rid]$ we can prune $t_{Si}.rid$ because in future lists the occurrences of $t_R.rid$ are more than the occurrences of $t_{Si}.rid.$ In other words, the elements of $t_R.r$ we have not seen yet are more than the elements of $t_{Si}.s$ we have not seen yet, thus $t_R.r \nsubseteq t_{Si}.s$.

4. Evaluation of Other Join Predicates

In this section we discuss the evaluation of other join operators on set-valued attributes, namely the set equality join and the set overlap join.

4.1 The Set Equality Join

Adaptations of the signature-based and inverted-file based algorithms discussed in Section 3 can also be applied for the set equality join, since it is not hard to see that it is a special case of the set containment join.

From signature-based techniques, we do not consider SNL, since this algorithm compares a quadratic number of signatures and we can do much better for joins with high selectivity. PSJ on the other hand can be very useful for set equality queries. We propose an adapted version of PSJ for set equality joins. The main difference from the algorithm described in Section 3.2 is that we use a single method to partition the data for both relations. While computing the signature of each tuple in $t_R \in R$ we also find the *smallest* element $e_r \in t_R.r$ (according to a predefined order in the domain D) and send the triplet $\langle |t_R.r|, sig(t_R.r), t_R.rid \rangle$ to the partition determined by a hash function $h(e_r)$. Tuples from S are also partitioned in the same way. The rationale is that if two sets are equal, then their smallest elements should also be the same.

The join phase is also similar; each pair of partitions is loaded in memory and the contents are re-hashed in memory using a prefix of their signature (e.g., the first byte). Then all pairs of memory buckets (one pair for each value of the prefix) are joined by applying a signature equality test. The set cardinality is also used before the signature comparison to prune early disqualifying signature pairs. We expect that this version of PSJ will do much better than the one for set containment joins, since replication is avoided in both partitioning and join phases.

The inverted file algorithms BNL and IFJ, can also be adapted to process set equality joins. The only change in BNL is that $\pi_{S.|s|}$ is now used to prune fast inverted list rids of different cardinality than the current $t_R \in R$. In other words, the condition $|t_R.r| \leq |t_S.s|$ is now replaced by $|t_R.r| = |t_S.s|$ for each $|t_S.rid|$ found in the lists of B_i. IFJ is also changed accordingly (using both $\pi_{S.|s|}$ and $\pi_{R.|s|}$).

4.2 The Set Overlap Join

The set overlap join retrieves the pairs of tuples which have at least one common element. For the more general ϵ-overlap join, the qualifying pairs should share at least ϵ elements. In this paragraph we describe how these operators can be processed by signature-based methods and ones that use inverted files.

The relaxed nature of the overlap predicate makes inappropriate the application of hashing techniques, like PSJ, on signature representations. Indeed it is hard to define a partitioning scheme which will divide the problem into joining P pairs of buckets much smaller than R and S. An idea we quickly abandoned is to partition both R and S by replicating their signatures to many hash buckets, after applying a hash function on each of their elements. This approach does not miss any solution, but the size of partitions becomes so large that the overall cost becomes higher than applying nested loops on two signature files. Another problem with this partitioning approach is that it introduces duplicate drops. Thus, for this operator we use the simple SNL algorithm. Finally, notice that whatever signature-based method we use, the filtering step will be the same for all ϵ-overlap queries independent to the value of ϵ, as discussed in Section 2.

On the other hand, algorithms that use inverted files could process overlap joins more efficiently. We propose an adaptation of BNL as follows. At each pass BNL merges the inverted lists found in B_i for each t_R. During merging, the number of occurrences for each $t_S.rid$ is counted. If this number is larger or equal to the threshold ϵ (i.e., we already found a matching pair of rids), the pair $\langle t_R.rid, t_S.rid \rangle$ is written to a temporary file Q_i. The remainder of the list is written to a file T_i, as before. Notice that if $\epsilon = 1$, there will be no temporary files T_i. In this special case we also compress the temporary Q_i's, since each t_R joins with numerous t_S. After processing all blocks, BNL merges all T_i's to produce a last file Q_{i+1} of qualifying results. The final phase scans all Q_i's (including the last one) and merges them to eliminate duplicates. Notice that each multiway merge is performed with a single scan over the temporary lists or results, since these are produced sorted. This version of BNL is described in Figure 5. We comment only on the marked line (1), which avoids writing a partial list to T_i if all elements of t_R are in the current block. If $t_R.r \subseteq L_i$, t_R cannot produce more qualifying pairs, since no more lists for t_R will be (or have been) produced.

```
Algorithm $BNL(R, S_{IF}, \epsilon)$ {
  $i := 0$;
  while there are more lists in $S_{IF}$ {
    $i := i + 1$;
    $B_i :=$ read next block of $S_{IF}$ that fits in memory;
    $L_i :=$ elements whose inverted list is in $B_i$;
    Initialize temporary file $T_i$;
    Initialize temporary file $Q_i$;
    for each tuple $t_R \in R$ do {
      $list(t_R) := \bigcup_{l_{el}}$, $el \in t_R.r \wedge el \in L_i$;
      for each $t_S.rid$ in $list(t_R)$ do
        if $t_S.rid$ appears at least $\epsilon$ times then {
          remove all $t_S.rid$ from $list(t_R)$;
          append $\langle t_R.rid, t_S.rid \rangle$ to $Q_i$; }
(1)   if $list(t_R) \neq \varnothing$ and $t_R.r \nsubseteq L_i$ then
        write $list(t_R)$ to $T_i$;
    }
  }
  $n := i$;
  merge all $T_i$, $1 \leq i \leq n$ to produce $Q_{i+1}$;
  merge all $Q_i$, $1 \leq i \leq n + 1$ to eliminate duplicates;
}
```

Figure 5: Block-Nested Loops (set overlap)

IFJ can also be adapted the same way as BNL for overlap joins. A set of temporary files and result files T_i and Q_i are produced as the algorithm merges the inverted lists. A special feature of IFJ is that, for $\epsilon = 1$ and if we ignore duplicate elimination, it can output results immediately and at the minimal cost of reading the inverted files.

5. Experiments

In this section we evaluate the performance of join algorithms on set-valued attributes. We first compare the proposed algorithms for set containment joins with PSJ, the previously proposed signature-based method. Afterwards, we evaluate the performance of signature-based and inverted file methods for other join predicates. The experiments were performed on a PC with a Pentium III 800MHz processor and 256MB of memory, running Linux 2.4.7-10. In the next subsection we describe the generator we used for our test data.

5.1 Data Generation

We generated synthetic relations with set-valued data using the same generator as in [13]. The parameters of the generator are the relation cardinality, the average set cardinality in a tuple, the domain size of set elements, and a correlation percentage $corr$. The domain of size $|D|$ is represented by the set of first $|D|$ integers for simplicity, and without loss of generality (such a mapping is typically used in real applications to reduce the space requirements of the tuples and the comparison costs). The domain is split into 50 equal-sized subdomains. Elements which fall in the same subdomain, model elements falling in the same real-world class (i.e., correlated elements).

The correlation percentage is used to tune the number of elements in a set which are correlated (i.e., fall in the same class). Thus, the relation is generated as follows. For each tuple, we pick a class according to a distribution (unless otherwise stated, we consider a uniform distribution). Then $corr\%$ of the set elements are picked from this sub-domain and the rest of them are randomly chosen from the remaining 49 subdomains. For set containment joins the data were generated, such that the size of the join result was controlled to be in the order of $|R|$. Finally, in all experiments, unless otherwise stated, R and S were generated to have the same cardinality, the correlation is set to 10%, and $|D|$=10,000. The same settings are used in the experiments of [13].

5.2 Set Containment Joins

In this subsection, we compare the performance of three algorithms for set containment joins. The first is the state-of-the-art PSJ algorithm proposed in [13]. The other two are BNL (described in Section 3.3) and IFJ (described in Section 3.4). We do not consider SNL, since it is inferior to PSJ, as shown in [13]. Unless otherwise specified, we used all optimization techniques for our proposed algorithms, (i.e., keeping the set cardinality projections in memory, pipelining, pairwise join implementation). For fairness to PSJ, in the overall cost we included the construction of the inverted files, wherever applicable.

In the first experiment, we compare the performance of the three methods for the following setting: $|R| = |S|$, the average set cardinality c is set to 20, the relation cardinality ranges from 20K to 180K and the memory buffer is set to 15% of the size of S on disk. Figure 6 shows the cost of the three methods for various relation cardinalities.

BNL is clearly the winner. The other algorithms suffer from the drawbacks discussed in Section 3. In terms of scalability, BNL is also the best algorithm; its cost is sub-quadratic to the problem size. This is due to the fact that even with only a small memory buffer, S_{IF} is split to a few blocks and R is scanned only a few times. In our experimental settings and for all instances of Figure 6, the number of blocks was just 2. On the other hand, both PSJ and IFJ do not scale well with the problem size. IFJ, in specific, produces a huge number of intermediate results, once the number of rids in the inverted lists increase. This is somehow expected, because IFJ generates a quadratic number of candidate rid pairs for each pair of inverted lists it joins.

Figure 7 shows the number of pages accessed by each algorithm. The page size is set to 4K in all experiments. The figure indicates that the I/O cost is the dominant factor for all algorithms, especially for IFJ, which generates a large number of temporary lists. The majority of the page accesses are sequential, thus the I/O cost translates mainly to disk transfer costs.

Figure 12 shows "where the time goes" in the experimental instance $|R| = |S| = 100$K. Notice that the bar for IFJ has been truncated for better visualization. Starting from the bottom, each bar accumulates the I/O and computational costs for partitioning (PSJ) or building the inverted files (BNL and IFJ), for joining (join phase of PSJ and overall join costs of BNL and IFJ), and for verifying the candidate rid-pairs (this applies only to PSJ). The burden of each algorithm can be easily spotted.

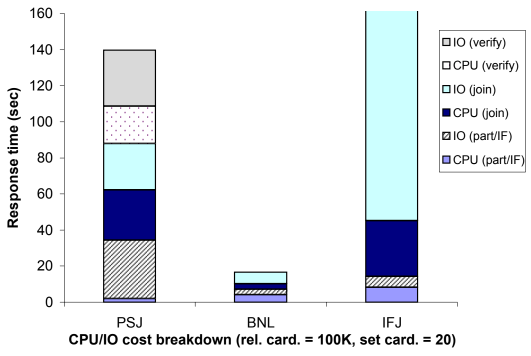


Figure 12: Cost breakdown (set containment)

The cost of PSJ is split evenly in all three phases. The algorithm generates a large number of replicated triplets for each $t_S \in S$. The triplets have to be written to the temporary files, re-read and joined with the partitions of R. Thus, PSJ spends a lot of time in partitioning (much more than the time needed to construct an inverted file). The join phase of PSJ is also slow, requiring many computations to derive the candidate pairs. Finally, the verification cost is also high due to the large number of false drops. Notice that the parameters of PSJ have been tuned according to the analysis in [13] and they are optimal for each experimental instance. For example, if we increase the signature length the I/O cost of partitioning and joining the data increases at the same rate, but the number of candidates and the verification cost drop. On the other hand, decreasing the signature length leads to smaller partitions but explodes the number of candidate rid-pairs to be verified.

The burden of IFS is the I/O cost of writing and reading back the temporary lists. The algorithm is less systematic than BNL, since it generates at a huge number of lists at a time, which cannot be managed efficiently. Apart from this, its computational cost is also high (mainly due to sorting a

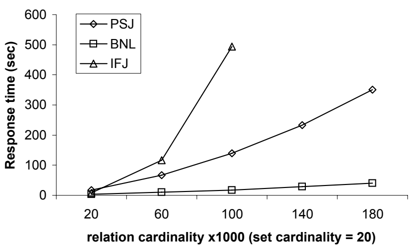


Figure 6: Scalability to the relation size

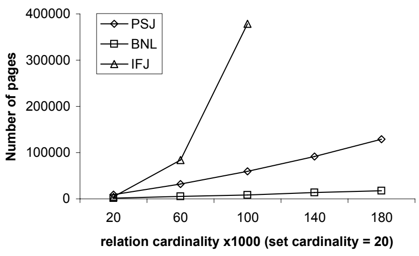


Figure 7: Scalability to the relation size (I/O)

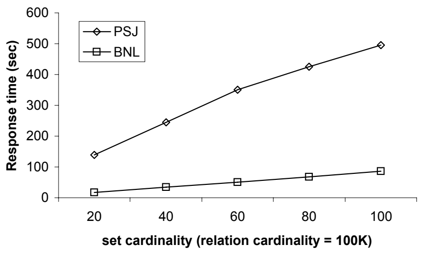


Figure 8: Scalability to set cardinality

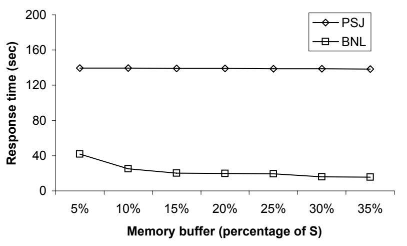


Figure 9: Scalability to memory buffer

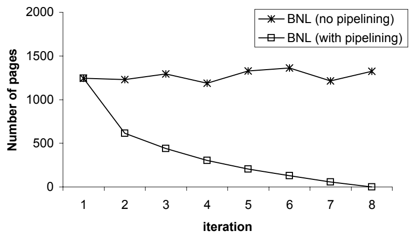


Figure 10: Effect of pipelining in BNL

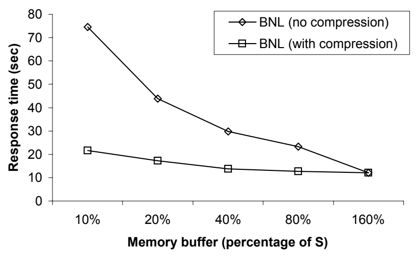


Figure 11: Effect of compression

large number of lists prior to merging them), indicating that this method is clearly inappropriate for the set containment join. Thus, we omit it from the remainder of the evaluation.

On the other hand, BNL processes set containment joins very fast. Notice that S_{IF} is constructed quite fast, due to the employment of the compression techniques. The join phase is slightly more expensive that the inverted file construction. In this experimental instance, R is scanned twice, dominating the overall cost of the algorithm, since the generated temporary results and S_{IF} are small (S_{IF} is just 28% of S), compared to the size of the relation.

The next experiment (Figure 8) compares PSJ with BNL for $|R| = |S| = 100$K and various values of the average set cardinality c. The performance gap between the two methods is maintained with the increase of c; their response time increases almost linearly with c. This is expected for BNL, since the lengths of the inverted lists are proportional to c. The I/O cost of the algorithm (not shown) increases with a small sublinear factor, due to the highest compression achieved. PSJ also hides a small sublinear factor, since the ratio of false drops decreases slightly with the increase of c. Finally, we have to mention that the cost of IFJ (not shown in the diagram) explodes with the increase of c, because of the quadratic increase of the produced rid-lists; the higher $|t_R.r|$ is, the larger the number of inverted lists that contain $t_R.rid$, and the longer the expected time to visit all these lists in order to output the matching pairs for this tuple.

We also compared the algorithms under different conditions, (e.g., $|R| \neq |S|$, different correlation values, etc.), obtaining similar results. BNL is typically an order of magnitude faster than PSJ. Notice, however, that the efficiency of BNL depends on the available memory. In the next experiment, we validate the robustness of the algorithm for a range of memory buffer values. The settings for this and the remaining experiments in this section are $|R| = |S| =$ 100K and $c = 20$. Figure 9 plots the performance of PSJ and BNL as a function of the available memory buffer. Notice that even with a small amount of memory (case 5% is around 350Kb), BNL is significantly faster than PSJ. The algorithm converges fast to its optimal performance with the increase of memory. On the other hand, PSJ joins a large amount of independent partitioned information and exploits little the memory buffer.

The next experiment demonstrates the effectiveness of the pipelining heuristic in BNL. We ran again the previous experimental instance for the case where the memory buffer is 5% the size of S. The number of blocks S_{IF} is divided to is 8 in this case. Figure 10 shows the number of intermediate results generated by BNL at each pass, compared to the results generated by the basic version of BNL that does not use pipelining. The results are lists of variable size (some compressed, some not) and we measure them by the number of pages they occupy on disk. Observe that pipelining reduces significantly the size of intermediate results as we proceed to subsequent iterations. Many lists are pruned in the latter passes, because (i) they are unsuccessfully merged with the ones from the previous pass, (ii) the corresponding tuple has already been pruned at a previous pass, or (iii) no more elements of the tuple are found in future lists and the current results are output. On the other hand, the basic version of BNL generates a significant amount of intermediate results, which are only processed at the final stage.

Finally, we demonstrate the effectiveness of compression in BNL. Figure 11 shows the performance of BNL and a version of the algorithm that does not use compression, as a function of the available memory. The effects of using compression are two. First, building the inverted file is now less expensive. Second, the number of inverted lists from S_{IF} that can be loaded in memory becomes significantly smaller. As a result, more passes are required, more temporary results are generated, and the algorithm becomes much slower. The performance of the two versions converges at the point where the uncompressed S_{IF} fits in memory. A point also worth mentioning is that the version of BNL that does not use compression is faster than PSJ (compare Figures 10 and 11) even for small memory buffers (e.g., 10% of the size of S).

To conclude, BNL is a fast and robust algorithm for the set containment join. First, it utilizes compression for better memory management. Second, it avoids the extensive data replication of PSJ. Third, it exploits greedily the available memory. Fourth, it employs a pipelining technique to shrink the number of intermediate results. Finally, it avoids the expensive verification of drops required by signature-based algorithms.

5.3 Other Join Operators

In this section, we compare signature-based methods with inverted file methods for other join predicates. In the first experiment, we compare the performance of the three methods evaluated in the previous section for set equality joins under the following setting: $|R| = |S|$, the average set cardinality c is set to 20, the relation cardinality varies from 20K to 180K and the memory buffer is set to 15% of the size of S on disk. Figure 13 shows the cost of the three methods for various relation cardinalities.

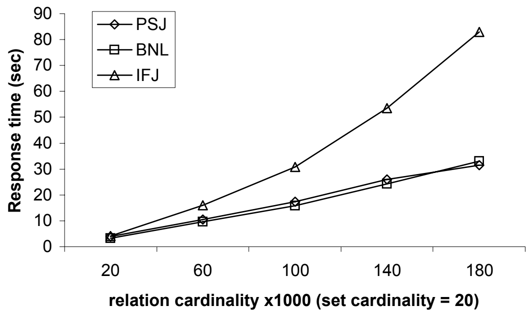


Figure 13: Varying $|R|$ (set equality join)

Observe that PSJ and BNL perform similarly in all experimental instances. They both manage to process the join fast for different reasons. PSJ avoids the extensive replication (unlike in the set containment join). It also manages to join the signatures fast in memory, using the prefix hashing heuristic. On the other hand, it still has to verify a significant number of candidate object pairs. The verification cost of the algorithm sometimes exceeds the cost of partitioning and joining the signatures.

BNL is also fast. Its performance improves from the set containment join case, although not dramatically. Many partial lists and rids are pruned due to the cardinality check and the temporary results affect little the cost of the algorithm. The decrease of the join cost makes the index construction an important factor of the overall cost. Finally, IFJ performs bad for set equality joins, as well. The arbitrary order of the generated lists and the ad-hoc nature of the algorithm make it less suitable for this join operation.

Figure 14 shows the performance of the algorithms, when the relation cardinality is fixed to 100K and the set cardinality varies. The conclusion is the same: PSJ and BNL have similar performance, whereas the cost of IFJ explodes with the set cardinality, because of the huge number of inverted lists that need to be merged.

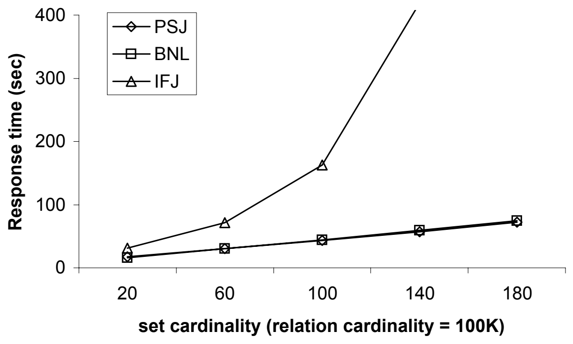


Figure 14: Varying c (set equality join)

In the final experiment we compare SNL, BNL and IFJ for set overlap joins. Figure 15 shows the performance of the algorithms for $|R| = |S| = 20\text{K}$ and $c = 20$. The extreme cost of the signature-based method makes it clearly inappropriate for set overlap joins. The selectivity of the signatures is low, even if a large signature length is picked, and the ratio of false drops is huge. In this setting, 86% of the pairs qualified the signature comparison, whereas only 4% are actual results for ϵ=1. Moreover, the increase of ϵ (and the selectivity of the join) does not affect the signature selectivity (as discussed in Section 2). BNL is faster than IFJ, although it generates many temporary results. There are two reasons for this. First, some results are output immediately because we know that they cannot match with lists in other blocks. Second, IFJ spends a lot of time in sorting, due to the less systematic production of the temporary lists. Notice that for $\epsilon = 1$ the cost of BNL is higher than for other values of ϵ due to the higher overhead of the result size. On the other hand, IFJ is less sensitive than BNL to the value of ϵ.

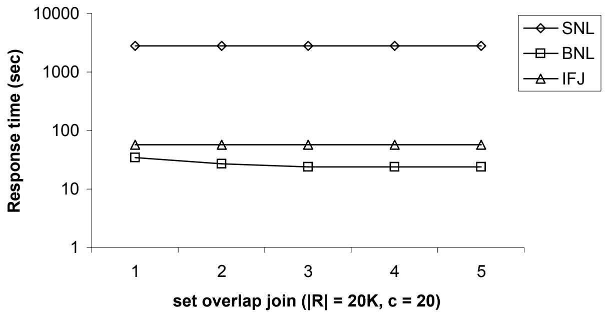


Figure 15: Varying ϵ (set overlap join)

As an overall conclusion, BNL the most appropriate algorithm for set overlap joins, too. In the future, we plan to optimize further this algorithm for this join operator, and also study its adaptation to other related query types, like the set similarity join. Arguably, the present implementation is not output sensitive; although the query result size reduces significantly with ϵ, this is not reflected to the evaluation cost, due to the a large number of temporary lists. An optimization we have not studied yet is to consider which of R and S should be placed as the "outer" relation in equality and overlap joins, since these operators are symmetric (as opposed to the set containment join).

6. Conclusions

In this paper, we studied the efficient processing of various join predicates on set-valued attributes. Focusing on the set containment join, we introduced a new algorithm which creates an inverted file for the "container" relation S and uses it to find in a systematic way for each object in the "contained" relation R its supersets in S. This algorithm is a variation of block nested loops (BNL) that gracefully exploits the available memory and compression to produce fast the join results. BNL consistently outperforms a previously proposed signature-based approach, typically by an order of magnitude. If the relations are already indexed by inverted files[1], join processing can be even faster.

We also devised adaptations of signature-based and inverted file methods for other two join operators; the set equality join and the set overlap join. The conclusion is that signature-based methods are only appropriate for set equality joins. For the other join types, a version of BNL is always the most suitable algorithm. On the other hand, a method that joins two inverted files was found inappropriate for all join types.

In the future, we plan to study additional, interesting set join operators. The *set similarity join*, retrieves object pairs which exceed a given similarity threshold. This join type is very similar to the ϵ-overlap join, however, the similarity function is usually more complex, depending on both overlap and set cardinality. Another variation is the *closest pairs* query [1], which retrieves from the Cartesian product $R \times S$ the k pairs with the highest similarity (or overlap). A similar operation is the *all nearest neighbor* query, which finds for each set in R its nearest neighbor in S. This query has been studied in [11], where inverted files were used, but the proposed algorithms were not optimized and only the I/O cost was considered for evaluating them.

Acknowledgements

This work was supported by grant HKU 7380/02E from Hong Kong RGC.

References

[1] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proc. of ACM SIGMOD Int'l Conference*, 2000.

[2] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *Proc. of ACM SIGIR Int'l Conference*, 1986.

[3] C. Faloutsos. Signature files. In *Information Retrieval: Data Structures and Algorithms*, pages 44–65, 1993.

[4] S. W. Golomb. Run length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[1]We expect this to be a typical case, in applications that index sets for containment queries [15, 8].

[5] R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.

[6] A. Guttman. R-trees: a dynamical index structure for spatial searching. In *Proc. of ACM SIGMOD Int'l Conference*, 1984.

[7] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. of VLDB Conference*, 1997.

[8] S. Helmer and G. Moerkotte. A study of four index structures for set-valued attributes of low cardinality. In *Technical Report, University of Mannheim*, number 2/99. University of Mannheim, 1999.

[9] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in oodbs. In *Proc. of ACM SIGMOD Int'l Conference*, 1993.

[10] N. Mamoulis, D. W. Cheung, and W. Lian. Similarity search in sets and categorical data using the signature tree. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2003.

[11] W. Meng, C. T. Yu, W. Wang, and N. Rishe. Performance analysis of three text-join algorithms. *TKDE*, 10(3):477–492, 1998.

[12] A. Moffat and T. Bell. In-situ generation of compressed inverted files. *Journal of the American Society of Information Science*, 46(7):537–550, 1995.

[13] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *Proc. of VLDB Conference*, 2000.

[14] M. Stonebraker. *Object-relational DBMS: The Next Great Wave*. Morgan Kaufmann, 1996.

[15] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *TODS*, 23(4):453–490, 1998.

[16] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full text databases. In *Proc. of VLDB Conference*, 1992.

[17] J. Zobel, A. Moffat, and R. Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proc. of VLDB Conference*, 1993.