Set containment joins using two prefix trees (Exposé)

Anja Kunkel

Supervised by: Prof. Dr. Ulf Leser, Dr. Panagiotis Bouros

September 7, 2013

1 Introduction

Set containment joins are a straightforward way of joining complex data: Given two relations both including a field with set data, those tuples shall be joined where each entry of the set of the first relation's set data field is completely contained in the set of the second relation's set data field.

A common example (see Mamoulis, 2003) of a set containment problem is the matching of people, people's skills, jobs and job skills. The way to answer "Which people are matching which job?" depends on the used data structures. In relational databases set attributes are usually modeled by normalized mapping using a separate relation for every single set attribute. People and their skills are then modeled with one relation containing some information about them and a separate relation containing one skill and one people id per line. The same applies to the job relation. To answer the question based on these relations, you have to join all four resulting relations together.

A more intuitive option is to keep the skill sets as data fields within the two containing relations[1]. A set containment join using both skill set fields will then answer the question above.

Set containment joins can be formally defined as follows:
Definition 1 (Set containment join)**.** *Given two relations R,S such that R.A and S.B are set values. Set containment join is then defined as*

$$R \bowtie_{A \subseteq B} S := \{(r, s) \mid r \in R, s \in S, r.A \subseteq s.B\}.$$

Apart from this, set containment join algorithms are most times discussed on a simplified data model: R and S are then tuples containing only an identifier and the data set field which consists only of integers.

[1]Ramasamy et al. (2000a) introduced this as "nested representation" in contrast to the unnested representation where the set is represented as a separate relation.

2 Related work

Set containment joins are shown to be a hard problem – in fact finding an optimal join strategy is NP-complete (Cai et al., 2001). They are well-studied and there are many algorithmic approaches. Most authors implement data structures for set containment joins as well as set equality joins and superset joins.

Starting in 1997, Helmer and Moerkotte (1997) introduced the first set containment join algorithms. They implemented some basic join strategies: nested-loop joins, sort-merge join, hash-based joins and a first version of a tree join. Afterwards different authors developed better algorithms at each of these techniques. Today set containment joins are often evaluated either doing nested-loop-evaluation or using an index structure on the inner relation and a loop over the outer relation.

When we evaluate a set containment join in nested loop manner, set containment joins are strongly related with set containment querying. Set containment querying means comparing a single set against a relation containing multiple sets. This is what is done in the inner loop of a nested loop , too. That is why some querying algorithms can also be used to compute joins and therefore are taken into account for the join problem.

2.1 Signature files

Signature-based join algorithms don't use the whole data itself but a smaller essence of them, their signature. This allows the algorithms to work more efficiently, but comes with the price of less precision and therefore false-positives and the need to dismiss them.

Signature functions infer – at least at small b which is the normal use case – that multiple sets are encoded into the same signature. Comparing signatures can tell you if an element is *not* included in a set. Comparing signatures can also tell you if it *may* be included. Comparing signatures cannot tell you that one element is *surely* included in a set. This leads to false-positives when comparing signatures of sets.

NL, SSF, ST and SNL (Helmer and Moerkotte, 1997, 2003; Mamoulis, 2003) create signature files of the inner join relation in a first step. In a second step they create signatures of the outer relation on the fly and match them against the inner relation. The third step does the verification which dismisses false positives. SSF and ST are two variants of NL which change the storage representation of the signature file of the inner relation: SSF stores them one after another in a sequential file while ST builds up a signature tree over them.

SHJ, ESH and LSJ (Helmer and Moerkotte, 1997, 2003; Melnik and Garcia-Molina, 2001) perform a hash-join over the signatures. The inner relation is therefore hashed with the signature function. In SHJ, the resulting buckets are kept in memory while LSJ extends them to get copied to secondary storage. To perform a join the entries of the outer relation are hashed one after another and merged with the signatures of the sets in the resulting bucket. ESH extends this approach by using a extensible hash function.

2.2 Hashing

Another approach to set containment join is directly hash based. PSJ (Ramasamy et al., 2000b), APSJ, DCJ and ADCJ (Melnik and Garcia-Molina, 2002, 2003) use hashing on the single elements of the sets to find matching candidates. This avoids signing the sets but uses more space to save the whole item sets.

PSJ and as well its extension APSJ do an adaption of a grace hash join: both relations are hashed using the same hash function and their buckets are written to disk. Afterwards the matching buckets are read two at a time and the contents get merged using additionally saved signatures. The elements of one of the relations that shall be joined are put into the bucket resulting from the hashing of only one chosen set element. The elements of the other relation must be put into all of the resulting buckets (leading to a lot of replication).

DCJ extends PSJ by avoiding the replication of one relation into that many buckets. Instead, in intermediate steps the relation which gets replicated is switched multiple times. This way the replication factor can be kept low at the price of a more difficult hashing procedure. Melnik and Garcia-Molina (2003) evaluated this to be competitive to the signature based variants depending on the element domain size and the data corellation rate.

2.3 Inverted files

Inverted files represent the sets of the relations differently: for each item in these sets a list of all containing sets is created. The inverted files are stored on disk and most times accessible using a B-tree.

Joining with inverted files is done mainly by merging different inverted file lists. For each item contained in at least one set in at least one of the relations, the set lists from the inverted files are compared and matching candidates are extracted. Helmer and Moerkotte (1997) evaluated the basic inverted file algorithm IF as much faster and using less storage space than signature based methods.

IF (Helmer and Moerkotte, 2003) , BNL and IFJ (Mamoulis, 2003) differ by the strategy how to handle the overflow of main memory. BNL, which Mamoulis (2003) evaluated as the fastest one, uses the techniques from the well known block nested loop join: The item lists are read in blocks small enough for main memory to avoid reading all the items many times from secondary storage (this implies that some tuples must still be carried over block limits).

OIF (Terrovitis et al., 2011) as a set containment querying algorithm extends the inverted files. They use ordered lists and a b-tree is laid over them to access certain parts of the file directly. The algorithm works by finding candidates on the least frequent items and dismiss candidates on other items. It outperformed other inverted file solutions for querying.

2.4 Trees

In the early beginning Helmer and Moerkotte (1997) introduced a first tree technique called Tree Join. This tree represents all data of the inner relation in a binary tree. Each non-leaf node has two children with the same node label – an item of the sets. One of them is labeled “match” which contains the subtree of all sets containing the element represented in this node label. The other one is labeled “nomatch” and is used for all other sets. For joining the outer relation is looped and every single item set is matched against the tree. Helmer and Moerkotte (1997) stated this technique as not competitive even against simple sort-merge join or nested-loop join (e.g. NL).

As other authors adopted the idea of trees as a data structure for the set data, especially prefix trees showed as a helpful data structure. Prefix trees are trees where for the content of each node in the tree is fully represented by the path to its position. Because the set elements represented have to have an order to get a unique path, an arbitrary order has to be chosen first. Each node of the prefix tree holds a list of tuple identifiers that contain exactly this set (the list can be empty). Each edge is labeled with an element of the alphabet over the set elements.

When extending a prefix tree to a prefix trie (Baeza-Yates and Ribeiro-Neto, 1999) the edge labels can instead be lists of elements. A list of e.g. two labels means the same as if there were a node between with just one anchestor and the two edges would be labeled each with one item. This extension can save a lot of time and storage space especially when compromising long chains of nodes into a single node.

Trees have been used by two algorithms that do set containment query answering instead of set containment join evaluation: MixIIT (Hossain and Jamil, 2010) and HTI (Terrovitis et al., 2006). Both mix a tree with inverted files by using the tree for the most frequent items and the inverted files for all other items. HTI uses a prefix trie to hold the top-k frequent items while infrequent items are only stored in an inverted file. At MixIIT, a concept closure tree holds the items with the top entries while all other are in an inverted file as well.

Jampani and Pudi (2005) introduced another tree based set containment join, called *PRETTI*. PRETTI uses a prefix trie for the inner relation and inverted files for the outer relation. The prefix trie is then traversed down while maintaining lists of match candidates destillated from the inverted files.

2.5 State of the art algorithm: PRETTI

Pretti uses a prefix trie to represent the inner relation. The outer relation is stored as inverted files. To evaluate a set containment join, each child of the root of the prefix trie is treated separately: A list of all match candidates for this node (at the beginning this means all sets containing this node) is held. For each child of the node – which represents a item –, the containing list is taken from the inverted files and merged with the current candidate list. This merge step is repeated recursively for each node deeper down the . Whenever a node of the trie represents an item set the current candidate list is outputted.

The partitioned signature join methods outperform all other signature based methods and are competitive with hashing methods (see Ramasamy et al., 2000b; Mamoulis, 2003). While basic inverted file techniques are slower, the fastest inverted file algorithm BNL beats even PSJ (see Mamoulis (2003)) by combining the advantages of inverted files and partitioning. Jampani and Pudi (2005) evaluated their algorithm PRETTI as even faster and more efficient than BNL. Therefore PRETTI is the most efficient set containment join algorithm so far.

3 Two prefix tries

As seen in section 2, there are some well-working solutions to the set containment join problem. Nevertheless one approach is left open for evaluation: joining using two prefix tries, one for each of the joined relations.

The prefix tries should work similar to the ones used for PRETTI. To join two of them, the tries have to get some kind of intersected. Although this makes us traversing two tries, the containment evaluation is still promising: In contrast to the PRETTI algorithm, we can avoid handling large inverted lists and instead handle small tree nodes.

The intersection of two prefix tries for set containment evaluation works in the following manner:

- Each node represents the set that is expressed by the path to this node. The node contains a list of tuples which contain this set.
- Moving foreward is asymmetric: At the tree representing the inner relation, we cannot walk "over" nodes. At the outer relation tree, we can and we will. At the tree representing the inner relation, go deeper iff we find a matching node at the outer tree. At the tree representing the outer relation, go deeper as long as there is a chance to find a matching node.

The tree shown in figure 1 represent two relations. Relation "A" contains tuples with the sets $\{B\}$, $\{B, E\}$ and $\{C\}$. Relation "B" contains tuples with the sets $\{A\}$, $\{B\}$, $\{B, C\}$

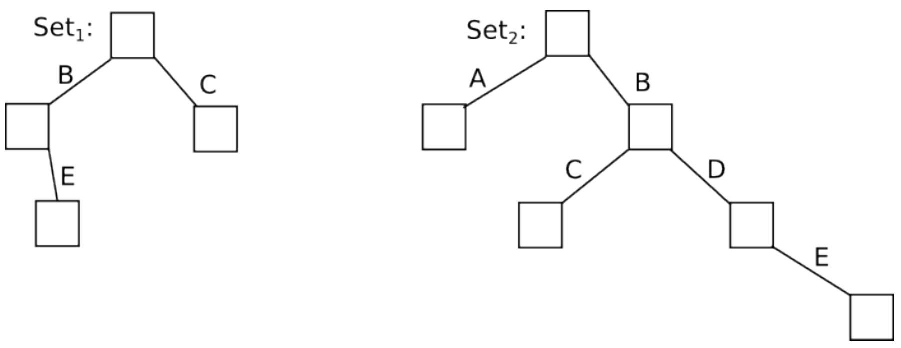

Figure 1: Example of two trees representing sets of sets.

and $\{B, D, E\}$. The resulting join pairs for $A \bowtie_{\subseteq} B$ shall be $(\{B\}, \{B\})$, $(\{B\}, \{B, C\})$, $(\{B\}, \{B, D, E\})$, $(\{B, E\}, \{B, D, E\})$ and $(\{C\}, \{B, C\})$.

3.1 Basic algorithm

As a starting point we propose the basic algorithm shown in figure 2.

Jampani and Pudi (2005) already did set equality joins using two prefix tries. They did not explain why they did not use the same approach for set containment joins. This confirms the suggestion that such a basic solution will be too slow while traversal to be competitive with PRETTI. Extensions to improve its performance will therefore be discussed in section 3.2.

```
search( $n_a$, $n_b$):
if containsTuples($n_a$) and $n_b$ = $n_a$
    output ($n_a$, $n_b$)

for $c_b \in$ child($n_b$)
  if $n_a$ = $n_b$
    for $c_a \in$ child($n_a$)
      if $c_a \geq c_b$
        search($c_a$, $c_b$)
  else
    if $n_a \geq c_b$
      search($n_a$, $c_b$)
```

```
output ($n_a$, $n_b$):
print tuples($n_a$) $\times$ tuples($n_b$)
for $c_b \in$ child($n_b$)
  output($n_a$, $c_b$)
```

Figure 2: Basic algorithm for finding set containments in two existing prefix trees

The algorithm contains one main recursive walk over both prefix trees at the same time. The evaluation of the join $a \bowtie_{\subseteq} b$ is started by calling search($root_a$, $root_b$). The algorithm then works in two parts: The main function search traverses two given trees to find pairs of subtrees that may contain matches. The helper function output traverses the found pairs of subtrees, but just outputs pairs of nodes that are surely matching.

search tries to match two subtrees at each call. In the first call these are the whole trees. As a first part (starting from line 2), the algorithm tests "Can we output the given nodes as matching partners?". If n_a matches n_b *and* n_a represents sets, we can surely output n_b and its descendants as matching partners of n_a.

As second part the search function is traversing deeper down the tree - "Where are potentially matching children?". If it's currently looking at two matching nodes (line 6), then the function is called recursively for all possibly matching pairs of child nodes. As a simple pruning strategy, pairs that cannot match are ignored: If c_b is already larger than c_a, c_b will never be found in c_a or its descendants because of the tree construction which is based on a fixed order. If we are currently looking at two nodes which don't match (line 10), we traverse only the left using the same simple pruning strategy. In both cases search is called recursively for all potentially matching combinations of subtrees.

The output function receives two subtrees that are surely matching. It prints the cartesian product of the tuple lists contained by the root nodes . If two sets are set contained, this implies that the first set also is contained in every superset of the second one – represented in the tree as descendants. Therefore output also calls itself on every child of the second root node recursively.

3.2 Extensions

The basic algorithm cannot efficiently handle cases when a matching partner in the outer relation is searched that could be a leaf or deep down node of the tree. Especially it cannot efficiently prune subtrees in which a matching partner could have been a node down at the bottom but actually isn't contained. Instead this algorithm traverses the whole tree searching for a possible matching partner. An example is shown in figure 3: Given $Set_2 \bowtie_{\subseteq} Set_1$ shall be evaluated, the value "H" could be a child of every single node of Set_1. Therefore every path in Set_1 has to be traversed to look for matching partners of "H".

This problem cannot be avoided by choosing a different order over the set items in the tree. Regardless of the chosen order there is always an item which is down in the tree. Even if we choose a frequency ordering of the set items at the price of counting all the frequencies, searching for the last item is less frequent, but not impossible.

We suggest two ways to cope with this problem. The first one is storing some extra information in each node that allows us to prune unprofitable subtrees early. The second

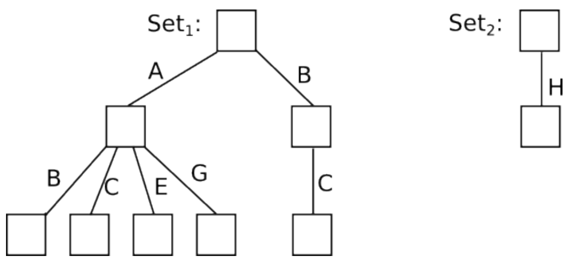

Figure 3: Bad case of basic algorithm tree algorithm: When $Set_2 \bowtie_\subseteq Set_1$ is evaluated, each node in Set_1 has to be traversed although no matching partner will be found.

one is to encode the prefix trie in a different way such that the encoding of the trie itself already contains information about pruning chances.

3.2.1 Using signatures

Storing extra information is expensive as the nodes of the tree get bigger and use more storage space so the tree traversal gets slower. That's why it is important to balance the weight of the extra information against the benefit of using it. What kind of small but helpful information can we store in the nodes?

We can use the well-known signature encoding (introduced in section 2.1) to represent the contents of all children of a node. A signature "ABCEG" at the root node of Set_1 in figure 3 for example would let us prune the whole tree traversal: Already at visiting the first node we know that "H" won't be found in any child of it. These signatures could be saved for every node so that pruning at deeper nodes gets possible.

The benefit of these signatures is obvious, but they need a lot of extra storage . There are at least two options to reduce the weight of these signatures:

- The signatures can be shortened. This implies the occurrence of false positives – leading to traversals of the tree without finding join pairs.
- Not every node of the tree has to contain a signature. This again leads to traversals of the tree without finding join pairs – at least down to the next signature.

3.2.2 Encoding of reachability information

Instead of storing extra information, the additional information about the tree can be saved in the encoding of the tree itself. Interval encoding allows us to decide whether a certain element is contained in a subtree by simple arithmetics.

For each node in a tree the interval encoding is defined as follows:

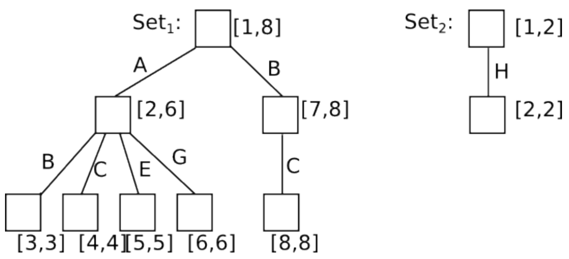

Figure 4: Trees from figure 3 with additional interval encoding.

Set_1:	A	[2,6]	
	B	[3,3]	[7,8]
	C	[4,4]	[8,8]
	E	[5,5]	
	G	[6,6]	
Set_2:	H	[2,2]	

Figure 5: Interval encoding of the trees in figure 4.

Definition 2 (Preorder-interval encoding)**.** *The interval I of each node N in a tree is $I(N) := (i_1, i_2)$ with*

$$i_1 := preorder(N)$$

and

$$i_2 := max(\{preorder(S) \mid S \in subtree(N)\})$$

The tree is saved as a list of lists. For each letter of the alphabet a list of intervals is saved.

This encoding allows us to prune subtrees not containing a certain element very early. The question "Is '1' contained in a subtree of a node B?" can easily be answered:

$$contains(subtree(B), \text{“1''}) = true \iff \exists A : val(A) = \text{“1”} \wedge I_1(B) \leq I_1(A) \wedge I_2(A) \leq I_2(B).$$

In figure 4 the trees from figure 3 are annotated using interval encoding. This encoding results in a list representation like in figure 5.

Interval encoding allows us to prune bad cases like from figure 3 very early. This comes at the cost of a totally different encoding scheme:

- All information that was saved within the nodes before must now be stored elsewhere – namely the lists of relation tuples.
- The interval lists are accessed very often. To make the encoded join performing well they have to be accessible at low cost.

Especially the second one seem to be relevant to the effectiveness of the whole join algorithm implementation. There seems to be no possibility to do these lookups in constant time. We plan to solve this by searching ordered lists accessible through a hash table.

4 Evaluation

To evaluate the proposed set containment join algorithms, four different algorithms shall be benchmarked:

- The state-of-the-art algorithm PRETTI.
- The basic algorithm proposed in section 3.1.
- The signature extension from section 3.2.1.
- The reachability information extension from section 3.2.2.

To evaluate the algorithms, different datasets with different specifics (e.g. set cardinalities) will be used. The benchmark will be done as a self join.

- BMS (Zheng et al., 2001), a collection of software click-stream data. This is the data set Jampani and Pudi (2005) already used to benchmark PRETTI.
- FLICKR, a collection of photographs and its title and tags.
- KOSARAK (Goethals, 2013), a collection of a new portal click-stream data.
- NETFLIX, a collection of movies and user ratings.

The benchmark should answer the following questions: How do the solutions perform? Which of them is better in which case? Do the extensions really solve the problem or instead add too much overhead?

Bibliography

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (1999). *Modern Information Retrieval.* ACM press New York, 1 edition.

Cai, J.-Y., Chakaravarthy, V. T., Kaushik, R., and Naughton, J. F. (2001). On the complexity of join predicates. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 207–214, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/375551.375592.

Goethals, B. (2013). Frequent itemset mining dataset repository. Available from: http://fimi.ua.ac.be/data/.

Helmer, S. and Moerkotte, G. (1997). Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. of the 23rd Conf. on Very Large Data Bases (VLDB), pages 386–395.* Morgan Kaufmann Publishers Inc.

Helmer, S. and Moerkotte, G. (2003). A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261. Available from: http://link.springer.com/content/pdf/10.1007

Hossain, S. and Jamil, H. (2010). A hybrid index structure for set-valued attributes using itemset tree and inverted list. In *Database and Expert Systems Applications*, pages 349–357. Springer.

Jampani, R. and Pudi, V. (2005). Using prefix-trees for efficiently computing set joins. In *Proceedings of the 10th international conference on Database Systems for Advanced Applications*, pages 761–772. Springer-Verlag.

Mamoulis, N. (2003). Efficient processing of joins on set-valued attributes. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM.

Melnik, S. and Garcia-Molina, H. (2001). Divide-and-conquer algorithm for computing set containment joins (extended technical report). Technical Report 2001-32, Stanford InfoLab. Available from: http://ilpubs.stanford.edu:8090/502/.

Melnik, S. and Garcia-Molina, H. (2002). Divide-and-conquer algorithm for computing set containment joins. In *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology, pages 427–444.* Springer-Verlag.

Melnik, S. and Garcia-Molina, H. (2003). Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems*, 28(1):56–99.

Ramasamy, K., Naughton, J., and Maier, D. (2000a). High performance implementation techniques for set valued attributes. Technical report, Computer Sciences Department, University of Wisconsin, Madison.

Ramasamy, K., Patel, J. M., Naughton, J. F., and Kaushik, R. (2000b). Set containment joins: The good, the bad and the ugly. In *International Conference on Very Large Data Bases, pages 351–362.* Morgan Kaufmann Publishers Inc.

Terrovitis, M., Bouros, P., Vassiliadis, P., Sellis, T., and Mamoulis, N. (2011). Efficient answering of set containment queries for skewed item distributions. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 225–236, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1951365.1951394.

Terrovitis, M., Vassiliadis, P., Passas, S., and Sellis, T. (2006). A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *Proceedings of* *the 15th ACM international conference on Information and knowledge management*, pages 728–737. ACM.

Zheng, Z., Kohavi, R., and Mason, L. (2001). Real world performance of association rule algorithms. In *Knowledge Discovery and Data Mining*, pages 401–406.