IJCSE International Journal of Computer Sciences and Engineering Open Access
Survey Paper Volume-6, Issue-9 Sep 2018 E-ISSN: 2347-2693

An Extensive Survey on Various Set Containment Joins Techniques

G. Sakthivel[1*], P.Madhubala[2]

[1]Dept. of Computer Science, Periyar University, Salem, India
[2]Dept. of Computer Science, Arignar Anna College (Arts & Science), Krishnagiri, India
[2]Dept. of Computer Science, Don Bosco College, Dharmapuri, India

*Corresponding Author: sakthishc@gmail.com

Available online at: www.ijcseonline.org

Accepted: 22/Sept./2018, Revised:25/Sept./2018

Abstract—A set containment join is a join between set-valued attributes of two relations, whose join condition is specified using the subset (⊆) operator. Set containment joins are deployed in many database applications, even those that do not support set-valued attributes. In this paper, we study the problem of set containment join. Given two collections R and S of records, the set containment join R⋈⊆S retrieves all record pairs {(r,s)} ∈ R × S such that r ⊆ s . This problem has been extensively studied in the literature and has many important applications in commercial and scientific fields. Recent research focuses on the in set containment join algorithms, In this paper, we propose three novel partitioning algorithms, called the Adaptive Pick-and-Sweep Join (APSJ), the Adaptive Divide-and-Conquer Join (ADCJ), and Divide-and-Conquer Join (DCJ) which allow computing set containment joins efficiently. We present a detailed analysis of the algorithms and study their performance on real and synthetic data using an implemented of this algorithm.

Keywords -Algorithms, Experimentation, Performance and Implementations

I. INTRODUCTION

Set containment queries are utilized in many database applications, especially when the underlying database systems support set-valued attributes. Set containment joins are used in a variety of other scenarios. If, for instance, our first relation contained sets of parts used in construction projects, and the second one contained sets of parts offered by each equipment vendor, we could determine which construction projects can be supplied by a single vendor using a set containment join. Or, consider a database application that recommends to students a list of courses that they are eligible to take. The two best known algorithms for computing set containment joins efficiently are the Partitioning Set Join (PSJ) proposed in Ramasamy et al. 2000 [2] and the Divide and- Conquer Join (DCJ) that we suggested in Melnik and Garcia-Molina 2002 [5]. PSJ and DCJ introduce crucial performance gains compared with straightforward approaches. A major limitation of PSJ is that it quickly becomes ineffective as set cardinalities grow. In contrast, DCJ depends only on the ratio of set cardinalities in both relations, and, therefore, wins over PSJ when the sets are large. Often, the sets involved in the join computation are indeed quite large. For instance, Biochemical databases contain sets with many thousands elements each. In fact, the fruit fly (drosophila) has around 14000 genes, 70-80% of which are active at any time. A snapshot of active genes can thus be represented as a set of around 10000 elements. PSJ is ineffective for such data sets. The contribution of this paper are two novel algorithms called the Adaptive Pick and-Sweep Join (APSJ) and the Adaptive Divide-and-Conquer Join (ADCJ), which extend and improve on the best known algorithms PSJ and DCJ. We show that ADCJ always outperforms DCJ, especially when the relations to be joined have different sizes. APSJ overcomes the main limitation of PSJ, namely, it's in ability to deal with large sets effectively. The set containment join is the component of a database management system to optimize the joins. It is used to select an efficient execution strategy for processing a joins. Set containment joins is a function of many relational database management systems in which multiple joins plans for satisfying the joins are examined and a good join plan is identified to minimize the use of certain resources like I/O by selecting the best join access plan. In present scenario data warehouses & mining turned out to be the common basis for the integration and analysis of data in modern enterprises. The goal of a data warehouse is to provide analysts and managers with strategic information about the key figures of the underlying business. Since micro data are of no interest at this level, almost all joins on data warehouses involve aggregates. In large data warehouse systems, it is critical to set containment join workloads to maximize system utilization and minimize processing times. The database administrator and the Data Base Management System joins became free to choose among many different storage formats and execution plans to answer a declarative joins. The challenge, since then, has been how to deliver on these promises regardless of where or how the data is laid out, how complex the join is, and how unpredictable the operating environment is. This survey is organized as follows, Section I contains the introduction of set containment join and its importance, Section II Set Containment Related works Section III Various Approaches and Section IV concludes the paper with future directions.

II. Set Containment Related works

The set containment join and other join operators for sets enjoyed significant attention in the area of data modelling. However, relatively little work deals with efficient implementations of these operators. Helmer and Moerkotte 1997 [1] were the first to directly address the implementation of set containment joins. They investigated several main memory algorithms including different flavours of nested-loop joins, and suggested the Signature-Hash Join (SHJ) as a best alternative. Later,

Ramasamy et al. 2000 [2] developed the Partitioning Set Join (PSJ), which does not require all data to fit into main memory. They showed that PSJ performs significantly better than the SQL-based approaches for computing the containment joins using unnested representation. Prior to Helmer and Moerkotte 1997 [1] and Ramasamy et al. 2000 [2], the related work focused on signature files, which had been suggested for efficient text retrieval two decades ago. A detailed study of signature files is provided by Faloutsos and Christodoulakis 1984 [3]. Ishikawa et al. 1993 [4] applied the signature file technique for finding subsets or supersets that match a fixed given query set in object-oriented databases.

In Melnik and Garcia-Molina 2002 [5], we presented the Divide-and-Conquer Set Join (DCJ) and the Lattice Set Join (LSJ) algorithms. LSJ is a partitioning algorithm which extends the main-memory algorithm SHJ Helmer and Moerkotte 1997 [1]. We demonstrated that DCJ always outperforms LSJ in terms of the replication factor. In Melnik and Garcia-Molina 2002 [5] we developed a comprehensive model for analyzing different partitioning algorithms that takes into account different set cardinalities and relation sizes, and measures the efficiency of the algorithms using the comparison and replication factors. In this paper, we used this analytical model for studying our novel algorithms APSJ and ADCJ. The adaptive algorithms presented in this paper introduce significant improvement over PSJ and DCJ. In particular, ADCJ always outperforms DCJ due to smaller replication factor, just like DCJ outperforms LSJ. In Melnik and Garcia- Molina 2002 [5] we suggested for DCJ a fixed pattern for applying operators *α* and *β*, which works reasonably well when the input relations *R* and *S* have approximately equal sizes and the set cardinalities are approximately the same (i.e., $\rho \approx 1, \lambda \approx 1$). In this paper, we compute the *α*, *β*-pattern adaptively based on the characteristics of the input relations to minimize replication. For brevity we do not discuss several aspects relevant for computing set containment joins. Examples are trading CPU time for I/O time by selecting the algorithm and partition number appropriately, choosing the signature size optimally, or using multi-stage partitioning (some of these aspects are examined in Ramasamy et al.2000 [1]. For generating synthetic databases used in our experiments, we deployed the methods described in Gray et al. 1994 [6]. The inherent theoretical complexity of computing set containment joins was addressed in Cai et al. 2001 [7] and Heller stein et al. 1997 [8]. Partitioning has been utilized for computing joins over other types of non-atomic data, e.g., for spatial joins Patel and DeWitt 1996 [9]. A possible alternative to partitioning joins are index joins. Index-based approaches for accessing multi-dimensional data were studied e.g. in B¨ohm and Krieger 2000 [10].

III. Various Approaches

A. Partitioning Set Join (PSJ)

Partitioning has been suggested to further improve performance by decomposing the join task R $\bowtie$ S into k smaller subtasks R1 $\bowtie$ S1, . . .,Rk $\bowtie$ Sk such that R $\bowtie$ S = $\bigcup$ ^{k} i=1 $R_i \bowtie S_j$. The so called partitioning function $\prod$ assigns each tuple of R to one r multiple partitions R_1, . . . , R_k, and each tuple of S to one or multiple partitions S_1, . . . , S_k. Consider our sample relations R and S from Two Relation with Set – Valued attributes. Let $\prod$ (a) =$\prod$ (b) = $\prod$ (A) = $\prod$ (B) = {1}, $\prod$ (c) = (C) = {2}, and $\prod$ (D) = {1, 2}. That is, R is partitioned into R_1 = {a, b}, R_2 = {c}, and S is partitioned into S_1 = {A, B, D}, S_2 = {C,D}. Note that we have constructed π so that tuples in R_1 can only join S_1 tuples, and R_2-tuples can only join S2-tuples. Thus, finding R $\bowtie_{\subseteq}$ S amounts to computing ($R_1 \bowtie_{\subseteq} S_1$) $\bigcup$ (R_2 $\bowtie_{\subseteq}$ S_2). Notice that computing $R_1 \bowtie_{\subseteq}$ S1 = {a, b} $\bowtie_{\subseteq}$ {A,B,D} and R2 R1 $\bowtie_{\subseteq}$ S_2 = {c} $R_1 \bowtie_{\subseteq}$ {C,D} requires only 2 · 3 + 1 · 2 = 8 signature comparisons. Hence, by using partitioning we reduced the total number of signature comparisons from 12 to 8. We refer to the fraction 8 12 as a comparison factor. The comparison factor ranges between 0 and 1. Besides reducing the number of required signature comparisons, partitioning helps to deal with large relations R and S that do not fit into main memory by storing the partitions R_1, . . . , R_k and S_1, . . . , S_k on disk. To minimize the I/O costs of writing out the partitions to disk and reading them back into memory, the partitions typically contain only the set signatures and the corresponding tuple identifiers. In our example, |{a, b}|+|{c}| = 3 signatures from $R_{1,2}$ and |{A,B,D}|+|{C,D}| = 5 signatures from $S_{1,2}$ are stored on disk temporarily. We refer to the ratio between the total number of signatures that are written out to disk and the total number of tuples in *R* and *S* as the *replication factor*. In our example, the replication factor is 3+5/3+4 = 8/7. Assuming that no partition is permanently kept in main memory, he optimal replication factor that can be achieved in a partition-based join is 1. A major challenge of effective partitioning is to construct a partitioning function that minimizes the comparison and replication factors. Obviously, $\prod$ needs to be correct, i.e., it has to ensure that all joining tuples are found.

Partitioning Set Join is algorithm reduce join execution time by partitioning the problem into smaller sub problems. A partitioning function is used to partition the problem. An ideal partitioning function requires Tuple *r* of *R* falls in one of the partitions R_i Tuple S of *S* falls in one of S_i Join is accomplished by joining only R_i with S_i Imagine that we want to partition *R* and *S* from Two Relation with Set – Valued attributes into $k = 8$ partitions. The partition number of each set of *R* is determined using a single, randomly selected element of the set. Consider the set $a = \{2, 9\} \in R$. Let 9 be a randomly chosen element of *a*. We assign *a* to one of the partitions 0, 1, . . . , 7 by taking the element value modulo $k = 8$. Thus, *a* is assigned to partition with index (9 mod 8) = 1, i.e., to partition R_1. Element 18 chosen from $b = \{8, 18\}$ yields partition number 2 = (18 mod 8). Finally, set *c* falls into partition R_3 based on randomly chosen element 3 $\in c$. Now we repeat the same procedure for *S*, but consider *all* elements of each set for determining the partition numbers. Taking all elements into account ensures that all joining tuples will be found. Thus, $A = \{2, 4, 9\}$ is assigned to partitions S_2, S_4, and S_1, $B = \{3, 8, 18\}$ goes into partitions S_3, S_0, and S_2, etc. The partition number of each set of *R* is determined using a single, randomly selected element of the set. We assign *a* to one of the partitions 0, 1, . . . , 7 by taking the element value modulo $k = 8$.

R1 = a{2, 9}	R_1 = a	9 mod 8 = 1
R2 = b{8, 18}	R_1 = 2	18 mod 8 = 2
R2 = b{1, 3}	R_1 = 2	3 mod 8 = 3

The same procedure for R, But all elements into account ensures that all elements into account ensures that all joining tuples will be found.

A {2, 4, 9} 2 mod 8 = 2 4 mod 8 = 4 9 mod 8 = 1

S_7 = A
S_7 = A
S_2 = A

B {3, 8, 18} 3 mod 8 = 3 8 mod 8 = 0 8 mod 8 = 2

S_7 = B
S_2 = B
S_2 = B

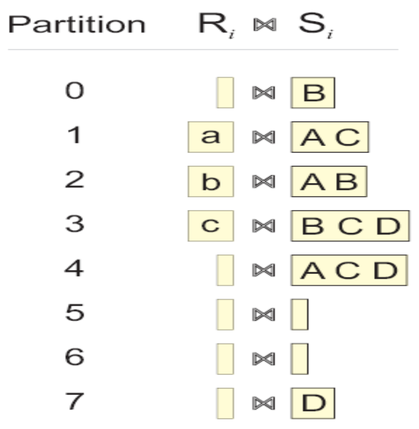
C {1, 3, 4} 1 mod 8 = 13 mod 8 = 3 4 mod 8 = 4

S_7 = C
S_2 = C
S_2 = C

D {3, 4, 7} 3 mod 8 = 3 4 mod 8 = 4 7 mod 8 = 7

S_7 = D
S_2 = D
S_2 = D

The complete partition assignment for *R* and *S*. PSJ requires that no *R* set be empty (a set with no elements cannot be assigned to any of the partitions without losing joining tuples). Once both relations are partitioned, i.e., the set signatures and tuple identifiers have been written out to



Partition	R_i	⋈	S_i
0		⋈	B
1	a	⋈	A C
2	b	⋈	A B
3	c	⋈	B C D
4		⋈	A C D
5		⋈	
6		⋈	
7		⋈	D

Figure 1. Partitioning with PSJ: 7 Comparison, 15 Replicated

disk, each pair of partitions is read from disk and joined independently. For example, when *R*3 and *S*3 are joined, the signature of set *c* is read from R_3, and is compared with the signatures of sets *B*, *C*, and *D* stored in S_3. Hence, computing $R3 \bowtie_{\subseteq} S3$ results in $1 \cdot 3 = 3$ signature comparisons. The total number of signature comparisons required in our example amounts to 0+2+2+3+0+0+0+0 = 7, whereas a total of 15 signatures need to be written out to disk. Thus, in this example, we obtain the comparison factor 7/12 ≈ 0.58, and Replication factor 15/3+4 ≈ 2.14

B. Adaptive Pick and Sweep Join (APSJ)

The Adaptive Pick-and-Sweep Join (APSJ) generalizes and extends the PSJ algorithm. We illustrate APSJ using our running example of Two Relation with Set – Valued attributes of Table I and k = 8 partitions. Assume that there exist $k - 1 = 7$ *Boolean hash functions* $h1, \ldots, h7$ that take a set of integers as input and return 0 or 1 as output. For example, consider the functions defined as

$$\boldsymbol{h_i(x) = 1 \Leftrightarrow \exists e \in x : (e \bmod 9) = i \text{ for } i = 1, \ldots, 7.}$$

R-Relation of (a) sets

a{2, 9} 2 mod 9 = 2 9 mod 9 = 0

h1	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**
0	**0**	**0**	**0**	**0**	**0**	**0**

R-Relation of (b) sets

b{8, 18} 8 mod 9 = 8 9 mod 9 = 0

h1	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**
1	**0**	**1**	**0**	**0**	**0**	**0**

R-Relation of (c) sets

c{1, 3} 1 mod 9 = 1 3 mod 9 = 3

h1	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**
0	**1**	**0**	**1**	**0**	**0**	**0**

S-Relation of (A) sets

A {2, 4, 9} 2 mod 9 = 2 4 mod 9 = 4 9 mod 9 = 9

h1	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**
0	**1**	**0**	**0**	**0**	**0**	**0**

S-Relation of (B) sets

B{3, 8, 18} 3 mod 9 = 3 8 mod 9 = 8 18 mod 9 = 0

h1	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**
0	**0**	**1**	**1**	**0**	**0**	**1**

S-Relation of (C) sets

C{1, 3, 4} 1 mod 9 = 1 3 mod 9 = 3 4 mod 9 = 4

h1	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**
0	**1**	**0**	**1**	**0**	**0**	**0**

S-Relation of (D) sets

D{ 3, 4, 7} 3 mod 9 = 3 4 mod 9 = 4 7 mod 9 = 7

h1	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**
1	0	1	1	0	0	0

Using these $k - 1 = 7$ functions, we partition our sample relations into $k = 8$ partitions as follows. For each set $r \in R$, we consider the indexes of the hash functions that fired, i.e., $\{j \mid hj(r) = 1\}$. We randomly *pick* an index i from this set, and assign r to partition R_i. If the set is empty, we assign r to the 'default' partition $R0$. For example, for set c we can choose between index 1 and 3, so say we select 1 and place c in R_1. (The selected indexes are underlined in Table III.) Set b is placed in $R0$. Every set $s \in S$ is inserted into all partitions S_j with $h_j(s) = 1$, i.e., we *sweep* the indexes of all firing functions. Additionally, each s is assigned to the 'default' partition S_0. Thus, for example, set A is assigned to partitions S_2, S_4, and, additionally, to partition S_0.

Table 1. Boolean Hash Functions used in APSJ

x ∈ **R**	**h1 h2 h3 h4 h5 h6 h7**	**y** ∈ **R**	**h1 h2 h3 h4 h5 h6 h7**
a	0 $\underline{1}$ 0 0 0 0 0	A	0 1 0 1 0 0 0
b	0 0 0 0 0 0 0	B	0 0 1 0 0 0 0
c	$\underline{1}$ 0 1 0 0 0 0	C	1 0 1 1 0 0 0
		D	0 0 1 1 0 0 1

The complete partition assignment produced by APSJ for our sample relations IS depicted in figure 2. we use Notice that because we use the default partitions R_0 and S_0, $k-1$ hash functions produce k partitions. The default partitions allow us to partition the relations correctly even if R contains empty sets, or, in general, set for which none of h_i fires (recall that PSJ cannot deal with empty sets). In our example, the joining tuples *b* and *B* are found when the partitions *R*0 and *S*0 are read from disk. Overall, 4+1+1+0+0+0+0+0 = 6 signature comparisons are needed, while the total of 16 signatures need to be stored on disk. Hence, we obtain the comparison factor 6/12 = 0.5 and replication factor 16/3+4 ≈ 2.14. In our tiny running example, APSJ wins over PSJ since it is lucky: had *c* been randomly assigned to bucket 4, APSJ would use more signature comparisons than PSJ. However, in real data, when the set cardinalities are large, PSJ tends to assign n almost all sets of *S* to each of the *Si* partitions, yielding many signature comparisons

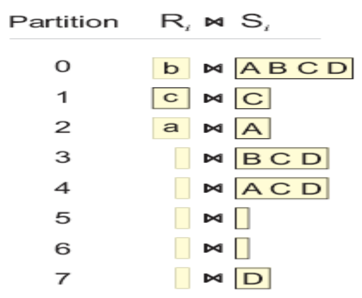


Figure 2 Partitioning with APSJ: 6 comparison, 16 Replicated

and high replication. APSJ offers an extra 'tuning knob' that is not available in PSJ, namely the boolean hash functions. Because of this flexibility, APSJ can often be tuned to achieve better performance than PSJ. Notice that if all hash functions fire with very *high* probabilities, then each S_i will include most of *S*, S_o joins will be expensive. In contrast, if the functions fire with very *low* probabilities, then R_0 will contain most of *R*, and we will have to join $R_0 \bowtie S_0 = R \bowtie S$. Clearly, to minimize the work, we need to select a firing probability somewhere in the middle. In Section 3.1 we show how to construct the APSJ hash functions *adaptively* depending on the characteristics of the input relations. Both algorithms PSJ and APSJ can be tuned by varying the number of partitions. The more partitions we use, the fewer comparisons are necessary. However, a larger number of partitions also causes more replication.

Adaptive Divide-and-Conquer Set Join (ADCJ)

The Adaptive Divide-and-Conquer Set Join (ADCJ) is based on the DCJ algorithm that we present in [Melnik and Garcia-Molina 2002]. Again, we illustrate the ADCJ algorithm using our running example of Table I and k = 8 partitions. We explain the algorithm using a series of partitioning steps depicted in Figure 3. In every step, one monotone Boolean hash function is used to transform an existing partition assignment into a new assignment with twice as many partitions. This transformation, or repartitioning, is done by applying either operator α or operator β to each pair of partitions $R_i \bowtie S_i$, as indicated by the labels 'α' and 'β' placed on the forks in Figure 3. Although we illustrate ADCJ conceptually as a branching tree, the final partition assignment is computed without using any intermediate partitions (see Appendix G). First, we explain the main idea of DCJ and then present the contribution of ADCJ, the adaptive design of the α, β-pattern. The monotone boolean hash functions that we use in Figure 3 are defined as

$$\boxed{\boldsymbol{h_i(x) = 1 \Leftrightarrow \exists\, e \in x : (e \bmod 4) = i \text{ for } i = 1, 2, 3.}}$$

R-Relation of (a) sets

a{2, 9} 2 mod 4 = 2 9 mod 4 = 1

h1	h2	h3
1	1	0

R-Relation of (b) sets

b{8, 18} 8 mod 4 = 0 18 mod 4 = 2

h1	h2	h3
0	1	0

R-Relation of (c) sets

c{1, 3} 1 mod 4 = 1 3 mod 4 = 3

h1	h2	h3
1	0	1

S-Relation of (A) sets

A {2, 4, 9} 2 mod 4 = 2 4 mod 4 = 4 9 mod 4 = 1

h1	h2	h3
1	1	0

S-Relation of (B) sets

B{3, 8, 18} mod 4 = 3 8 mod 4 = 0 18 mod 4 = 2

h1	h2	h3
0	1	1

S-Relation of (C) sets

C{1, 3, 4} 1 mod 4 = 1 3 mod 4 = 3 4 mod 4 = 4

h1	h2	h3
1	0	1

S-Relation of (D) sets

D{ 3, 4, 7} 3 mod 4 = 3 4 mod 4 = 0 7 mod 4 = 3

h1	h2	h3
0	0	1

Table 2. Boolean Hash Functions used in ADCJ

$x \in$ **R**	**h1 h2 h3**	**y** $\in$ **R**	**h1 h2 h3**
A	**1 1 0**	**A**	**1 1 0**
B	**0 1 0**	**B**	**0 1 1**
C	**1 0 1**	**C**	**1 0 1**
		D	**0 0 1**

Relations $R = \{a, b, c\}$ and $S = \{A, B, C, D\}$ form the initial partition assignment $R \bowtie S = R_0{}^1 \bowtie S_0{}^1$, where the superscript 0 indicates the step number. In Step 1, we derive a new partition assignment $(R_1^1 \bowtie S_1^1) \bigcup (R_1{}^2 \bowtie S_1{}^2)$ from $R \bowtie S$ using operator β and hash function h_1. Sets B and D with $h_1(B) = h_1(D) = 0$ are assigned to partition S_1^1, while the remaining sets A and C with $h_1(A) = h_1(C) = 1$ are inserted into S_2^1. We abbreviate this procedure concisely as $S_1^1 := S/\neg h_1$, $S_2{}^1 := S/h_1$. Since h_1 is monotone, each subset x of B or D must satisfy $h_1(x) = 0$. Therefore, partition $S_1^1 = \{B, D\}$ needs to be joined only with those sets in $R = \{a, b, c\}$ that satisfy $h_1(x) = 0$, i.e. just with set b. In contrast, each set of R may possibly be a subset of A or C. Thus, we obtain $R_1^1 := R/\neg h_1$ and $R_1^2 := R$ (the values 0 and 'any' taken by $h1$ are depicted above R_1^1 and $R_1{}^2$). Notice that instead of 4·3 = 12 signature comparisons required for $R \bowtie S$, only 1·2+3·2 = 8 signature comparisons would be needed for joining the partitions of assignment 1. Given a pair of partitions $R_i \bowtie S_i$, operator β splits partition S_i and replicates partition R_i. In contrast, operator α splits R_i and replicates S_i. Figure 3 shows how operator α is used to repartition $R_1^2 \bowtie S_1^2 = \{a, b, c\} \bowtie \{A,C\}$. First, R_1^2 is split into $R_2^3 = R_1^2/h_2 = \{a, b\}$ and $R_2^4 = R_1^2/\neg h^2 = \{c\}$. Since each superset x of a or b must satisfy $h_2(x) = 1$, R_2^3 needs to be joined only with those sets of $S_1^2 = \{A,C\}$ that satisfy $h_2(x) = 1$, i.e., just with the set A. Hence, S_2^3 is obtained as $S_2^3 = S_1^2/h_2$, whereas S_2^4 must contain all of $S_1^2 = \{A,C\}$. The definitions of operators α and β are presented in Table V.

Table 3. Repartitioning of $R \bowtie S$ using operators α and β, and a monotone Boolean hash function h

operator	**Ideally, When**	**Resulting assignment**
$A(R \bowtie S, h)$	$\lvert R\rvert \geq \lvert S\rvert$	$(R/h \bowtie S/h) \bigcup (R/\neg h \bowtie S)$
$B(R \bowtie S, h)$	$\lvert R\rvert \geq \lvert S\rvert$	$(R/\neg h \bowtie S/\neg h) \bigcup (R \bowtie S/h)$

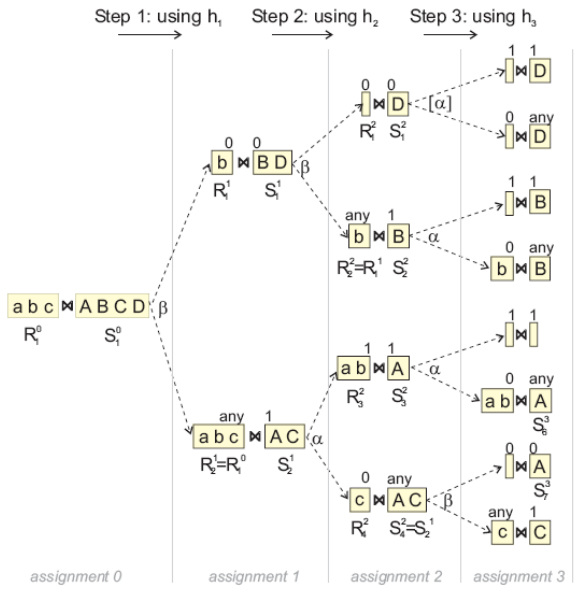


Figure 3: Partitioning with ADCJ: 4 comparisons, 11 replicated

Adaptive design of α, β-pattern. The operators α and β both perform correct repartitioning and thus can be applied interchangeably at each fork in the branching tree of Figure 3. Different patterns of applying α and β yield distinct partition sizes in the final assignment, so we can improve performance by selecting the operators judiciously. Optimal performance is achieved when the comparison and replication factors are minimal. As shown in Appendix C, the comparison factor is determined entirely by the firing probabilities of the hash functions, and is independent of the α,β-labelling of the tree. However, the choice of α, β-pattern is crucial for minimizing replication. The smallest replication factor is obtained if at each fork we always split the larger partition and replicate the smaller one. Otherwise, if a suboptimal choice is made, the replication factor of the sub tree originating at that fork increases and makes the overall replication factor grow. Hence, if $\lvert Ri\rvert \geq \lvert Si\rvert$, we should apply operator α, otherwise we should use β. For example, in Step 1, we have $\lvert R\rvert = 3 < 4 = \lvert S\rvert$. Therefore, operator β is best.

If we computed the intermediate partitions, we would know their sizes and could apply the above rule. However, we do not generate the intermediate partitions, since storing them temporarily on disk is prohibitively expensive.

Suppose for now that we know the optimal α,β-pattern, i.e., the one that minimizes replication. Then, we can compute the partition assignment of each set of R or S by 'tracing' its way through the tree, with no need for intermediate, materialized partitions. In our example, set A belongs initially to $S_0^1 = S$. Given that the β is applied at the first fork, we compute $h(A)$ to decide whether A is sent to $S11$ ('up') or S_1^2 ('down'). Since $h_1(A) = 1$, A is sent 'down'. At the next fork we send A both 'up' (S_2^2) and 'down' (S_2^4), based on $h_2(A) = 1$ and the use of operator α. Now the path of A splits, and we have to track both paths. After the final step, A is assigned to S_3^6 and S_3^7. In Appendix G we present a formal specification of the ADCJ algorithm that computes the partition assignment for each set based on the above technique.

Thus, our final challenge is to determine a 'good' α,β-pattern for the partitioning technique of the previous paragraph. We design the pattern adaptively based on the characteristics of the input relations. The key idea is to *estimate* the sizes of the intermediate partitions using the firing probabilities of the hash functions. Suppose that in our example we know that functions $h1$, $h2$, $h3$ fire with probability of 0.5 for sets in R, and with probability 0.6 for sets in S. Consider partitions $R_1^2 \bowtie S_1^2$ obtained in Step 1 using function h_1. The expected size of partition $S_1^2 = S/h_1$ can be estimated as $|S_1^2| = 0.6 \cdot |S| = 0.6 \cdot 4 = 2.4$. Given that $|R_1^2| = |R| = 3 > 2.4 = |S_1^2|$, we select operator α for repartitioning $R_1^2 \bowtie S_1^2$. Assuming that $R_1^2 \bowtie S_1^2$ are repartitioned using α, we can estimate the sizes of partitions R_2^3 and S_2^3. Since $R_2^3 = R_1^2/h_2 = R/h_2$, we get $|R_2^3| = 0.5 \cdot |R| = 0.5 \cdot 3 = 1.5$, while the expected size of S_2^3 is $0.6 \cdot |S_1^2| = 0.62 \cdot |S| = 1.44$. Because $|R_2^3| = 1.5 > 1.44 = |S_1^2|$, we choose operator α again to repartition $R_3^3 \bowtie S_2^3$. Of course, the actual partition sizes may deviate from the expected values, so we can choose a suboptimal operator. For example, the estimated size of partition R_2^1 is $|R_2^1| = |(R/\neg h1)/\neg h2| = (1-0.5)^2 \cdot |R| = 0.75$, whereas $|S_2^1| = (1-0.6)^2 \cdot |S| = 0.64$. Thus, we choose to apply operator α. However, as shown in Figure 3, in our example the actual sizes of R_2^1 and S_2^1 turn ut to be 0 and 1, i.e., β would have been a better choice. In fact, choosing β would require one less signature to be stored to disk.

To summarize, our algorithm computes the partition assignment in three stages.

(1). First, we construct the hash functions that minimize the comparison factor (just ike in DCJ).

(2). Second, we determine the α,β-tree that reduces replication using the firing probabilities of the hash functions.

(3). Finally, we compute the partition assignment by tracing each set of R and S through the α,β-tree.

In the final assignment produced in our example (Assignment 3), the total of 0 + 0 + 0 + 1 + 0 + 2 + 0 + 1 = 4 signature comparisons are required, whereas 11 signatures need to be written out to disk (one more than absolutely necessary if we had used β for $R_2^1 \bowtie S_2^1$). Thus, we obtain comparison factor $4/12 \approx 0.33$ and replication factor $11/3+4 \approx 1.57$, close to the best possible replication factor of $10/3+4 \approx 1.42$.

IV. CONCLUSION AND FUTURE SCOPE

We presented three novel partitioning algorithms, the Adaptive Pick-and-Sweep Join (APSJ), the Adaptive Divide-and-Conquer Join (ADCJ), Divide-and-Conquer Set Join DCJ which allow computing many set containment joins several times more efficiently than the previously known approaches. We provided a detailed analysis of the algorithms and studied their performance using an implemented. We found that APSJ, ADCJ, DCJ and the existing algorithm PSJ need to be used complementary for maximal performance. PSJ is the algorithm of choice when the set cardinalities are very small, e.g., below ten elements. For larger cardinalities, APSJ tends to outperform all other algorithms. In some settings, especially in those where the superset relation is much larger than the subset relation, or the element domain is small, ADCJ wins over APSJ and PSJ. It would be interesting to see whether the hash functions used in APSJ and ADCJ can be constructed optimally for small or non-uniform domains, or whether the algorithms presented in this paper reduce the theoretical complexity of containment joins below $O(|R| \cdot |S|)$. In this paper we suggested a novel algorithm called all three Set Join for computing, another challenging and mostly unexplored research direction.

Table 4. Summary of strong points of proposed techniques

No.	Techniques Proposed	Strong Point
1	Dynamically optimizing high-dimensional index structures.	We have proposed a dynamic optimization technique for multidimensional index structures. In contrast to conventional page-size optimization where the administrator determines the optimal page-size parameter before installing the database, our index is automatically adapted according to the data distribution of the objects currently stored in the database
2	On the complexity of join predicates	While the development of join algorithms is perhaps one of the best studied problems in database systems research, to date there has been very little published about the

		intrinsic difficulty of join problems. This may be because joins are too "easy." Any join can be computed in polynomial time — just compute the cross product and iterate through the result applying the join predicate to each tuple in the cross product. While this is true, this coarse-grained analysis belies the experience of researchers who have tried to find truly "good" algorithms for these join problems.
3	Quickly-generating billion-record synthetic databases	Convert a simple sequential load into a parallel load –turning a two -day task into a one-hour task. It then explored the ways to generate synthetic data. At first it focused on generating the primary keys of records and values uncorrelated to these keys: dense-unique-pseudo-random sequences. Then, attention turned to building indices on these synthetic tables –either by sorting, or by using discrete logarithms. By careful selection of generators, the discrete log problem is tractable and indices can be quickly generated within the 1-hour limit we set for the billion-record load. The paper then looked at skewed distributions. It presented the standard ways to generate uniform, exponential, normal, and Poisson distributions. It went into more detail on the new topic of self -similar and Zairian distributions. Using these techniques, one can generate billion-record databases in an hour, and a two terabyte databases per day
4	On the analysis of indexing schemes	This chapter examined techniques for implementing index structures for two-dimensional range queries. The focus was on the EPS-tree, a new access method for three-sided queries, with asymptotically optimal worst-case performance. The detail of the presentation was high, in order to demonstrate the practical decisions involved in designing such data structures. We now review some of the main conclusions from the material presented
5	evaluation of main Memory joins algorithms for joins with set comparison join predicates	For the first time, this paper investigates join algorithms for join predicates based on set comparisons. More specifically, this paper treats subset predicates. It has been shown that remarkably more efficient algorithms exist than a naive nested-loop join. Even the signature nested-loop join results in an order of magnitude improvement over the naive nested-loop join. The hash join surpasses the signature nested-loop join only by a factor of 5-10 depending on various parameters. Although this is a result' that is not to be neglected, the question arises whether even better alternatives exist. This is one issue for future research. Other problems need to be solved as well. First, join algorithms whose join predicate is based on non-empty intersection have to be developed Second, all the algorithms presented are main memory algorithms. Hence, variants for secondary storage have to be developed. Also the different tuning parameters will have to be adjusted for 393 secondary storage variants.
6	A repository of web pages	Our work raises a number of areas for further work: (i) How can we annotate and organize the communities discovered by the trawling process of Section 2.2? (ii) Bipartite cores are not necessarily the only sub graph enumeration problems that are interesting in the setting of the Web graph. The sub graphs corresponding to Web rings (which look like bidirectional stars, in which there is a central page with links to and from a number of \spoke" pages), cliques, and

		directed trees are other interesting structures for enumeration. How does one devise general paradigms for such enumeration problems? (iii) What are the properties and evolution of random graphs generated by specific versions of our models in Section 4? This would be the analog of the study of traditional random graph models such as Gn;p . (iv) How do we devise and analyze algorithms that are e - client on such graphs? Again, this study has an analog with traditional random graph models. (v) What can we infer about the distributed sociological process of creating content on the Web? (vi) What are structure can we determine for the map of the Web graph (Figure 4) in terms of domain distributions, pages that tend to be indexed in search engines, and so on?
7	Evaluation of signature files as set access facilities in oodbs	We have in this report described how signatures can be stored in the the OIDX. As the OD is accessed on every object access in any case, there is no extra signature retrieval cost. In non-versioned OODBs, maintaining signatures means that the OIDX needs to be updated every time an object is updated, but as the analysis shows, it will in most cases pay back, as less objects need to be retrieved. Storing signatures in the OIDX is even more attractive in TOODBs. In TOODBs, the OIDX will have to be updated on every object update anyway, so that in this case, the extra cost associated with signature maintenance is very low. As showed in the analysis, substantial gain can be achieved by storing the signature in the OIDX. We have done the analysis with different system parameters, access patterns, and query patterns, and in most cases, storing the object signatures in the OIDX is beneficial. The typical gain is from 20 to 40%. Interesting to note is that the optimal signature size can be quite small. In this paper we suggested a novel algorithm called the Divide-and-Conquer Set Join for computing the set containment joins. We compared the performance of DCJ with that of PSJ and LSJ. We developed a detailed analytical model that allowed us to study the join algorithms qualitatively, and to tune them for different input relations. Furthermore, we explored the behaviour of the algorithms experimentally using an implemented. We found that DCJ always outperforms LSJ in terms of the replication factor. In contrast, PSJ and DCJ provide complementary approaches for computing set containment joins. Specifically, when the set cardinalities are large, DCJ introduces a significant performance improvement as compared to PSJ. On the other hand, PSJ wins over DCJ when small sets are used.
9	Partition based spatial-merge join	In web-based environments, access to remote servers is restricted, progressive or inaccurate results can be tolerated, and existence of full spatial capabilities (e.g., spatial index structures and operations) cannot be assumed. Therefore, support for spatial joins in these environments becomes challenging. We proposed a three step simulation of spatial join on web-based environments. We put together an experimental setup with real database servers to evaluate our different plans. We demonstrated that Dynamic-MBR, which dynamically a proxy mates and merges polygons at the local site is the superior approach for the first step. We also proposed two alternative heuristics for Dynamic-MBR and we showed that the Minimum-Centroid Distance

		heuristic results in more merges while the Minimum-Wasted Area heuristic results in less number of false hits. Hence, in an environment with fast network and powerful local server that can deal efficiently with false hits, Minimum-Centroid Distance is the superior heuristic since it minimizes the remote query processing time
10	Set containment joins: The good, the bad and the ugly	This paper investigates algorithms for computing a set containment join. These algorithms cover two possible attributes: the un nested external representation and the nested internal representation. The un nested external representation is used by commercial O/R DBMSs for implementing set-valued attributes. In this case, set containment join is implemented using a standard SQL2 query. For the nested internal representation, this paper considers two algorithms. The rst is a variation of nested loops (Sig-NL) that uses signatures to speed up the evaluation of the join predicate. The second algorithm is PSJ, a new partition based algorithm that is proposed in this paper. This algorithm is based on a two level partitioning scheme by using set elements to partition relation R and replicate relation S. Within each partition, it uses an in-memory algorithm based on partitioning of signatures.

REFERENCES

[1] HELMER, S. AND MOERKOTTE, G. 1997. Evaluation of main memory joins algorithms for joins with set comparison join predicates. In VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. Morgan Kaufmann, 386–395.

[2] Ramasamy, K., Patel, J. M., Naughton, J. F., and Kaushik, R. 2000. Set containment joins: The good, the bad and the ugly. In VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan Kaufmann, 351–362.

[3] C. Faloutsos and S. Christodoulakis. Signature _les: An access method for documents and its analytical performance evaluation. ACM Trans. On office Information Systems (TOIS), 2(4):267-288, 1984.

[4] ISHIKAWA, Y., KITAGAWA, H., AND OHBO, N. 1993. Evaluation of signature files as set access facilities in oodbs. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993, P. Buneman and S. Jajodia, Eds. ACM Press, 247–256.

[5] MELNIK, S. AND GARCIA-MOLINA, H. 2002. Divide-and-conquer algorithm for computing set containment joins. In Proceedings of Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, C. S. Jensen, K. G. Jeffery, J. Pokorn´y, S. Saltenis, E. Bertino, K. Bohm, and M. Jarke,Eds. Lecture Notes in Computer Science, vol. 2287. Springer.

[6] GRAY, J., SUNDARESAN, P., ENGLERT, S., BACLAWSKI, K., AND WEINBERGER, P. J. 1994. Quickly generating billion-record synthetic databases. In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994,R. T. Snodgrass and M. Winslett, Eds. ACM Press, 243–252.

[7] CAI, J.-Y., CHAKARAVARTHY, V. T., KAUSHIK, R., AND NAUGHTON, J. 2001. On the complexity of join predicates. In PODS'01, Proceedings of the 20th ACM SIGACT-SIGMOD-SIGARTSymposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California. ACM Press.Faloutsos, C. and Christodoulakis, S. 1984. Signature files: An access method for documents and its analytical performance evaluation.

[8] HELLERSTEIN, J. M., KOUTSOUPIAS, E., AND PAPADIMITRIOU, C. H. 1997. On the analysis of indexing schemes. In PODS'97, Proceedings of the 16th ACM SIGACT-SIGMOD-SIGARTSymposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona. ACMPress, 249–256.

[9] Patel, J. M. and DeWitt, D. J. 1996. Partition based spatial-merge join. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec,Canada, June 4-6, 1996, H. V. Jagadish and I. S. Mumick, Eds. ACM Press, 259–270.

[10] BOHM, C. AND KRIEGEL, H.-P. 2000. Dynamically optimizing high-dimensional index structures. In Proceedings of Advances in Database Technology - EDBT 2000, 7th International Conferenceon Extending Database Technology, Konstanz, Germany, March 27-31, 2000, C. Zaniolo, P. C.Lockemann, M. H. Scholl, and T. Grust, Eds. Lecture Notes in Computer Science, vol. 1777.Springer

Authors Profile

Dr. P. Madhubala pursued Ph.D. in Computer Science from Mother Teresa Women's University, kodaikanal in the year 2017. She is currently working as Head & Assistant Professor in PG & Research Department of Computer Science, Don Bosco College, Periyar University, Salem since 2007. She has published more than 13 research papers in reputed international journals and participated in conferences including IEEE and it's also available online. Her main research work focuses on Cloud Security and Privacy, Cryptography Algorithms, Network Security, and Big Data Analytics. She has 17 years of teaching experience and 5 years of Research Experience.

G.Sakthivel pursued Bachelor of Science from Sacred Heart College, Madras University, Master of Computer Science from Thiruvalluvar University and M.phil of computer science in the year 2009. He is currently pursuing Ph.D. and working as Assistant Professor in PG Department of Computer Science, Arignar Anna College (Arts & Science) since 2010. He has published more than 3 research papers in reputed international journals and presented papers in National and International conferences. His main research work focuses on Set containment Joins, Cryptography Algorithms, Big Data Analytics and Data Mining. He has 10 years of teaching experience & 2 years of Research Experience.