Progressive Join Algorithms Considering User Preference

Mengsu Ding Shimin Chen* Nantia Makrynioti Stefan Manegold
SKL of Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences
{dingmengsu,chensm}@ict.ac.cn
CWI, Amsterdam, The Netherlands
{nantia.makrynioti,stefan.manegold}@cwi.nl

ABSTRACT

Progressive query processing is a new attractive paradigm for exploratory data analysis. This paper considers the case where users want to receive results ordered according to their preference, and specifically focuses on the design of join algorithms. We investigate the use of contour lines in progressive algorithms with user preferences, and propose ContourJoin to reduce sorting overhead of progressive preference-aware joins. Experimental results show that compared with the naïve blocking algorithm and the top-k RankJoin algorithm, ContourJoin has superior performance in both early result generation and total result computation.

1. INTRODUCTION

Exploratory data analysis is essential in a variety of applications [8], from analyzing astronomy data to detecting financial fraud and improving decision-making in healthcare. To support these applications, the research community has long realized the need for a new *progressive* query processing paradigm, where initial partial results are quickly returned to the user and then complemented gradually over time, instead of asking the user to wait for the full output at the end of query processing. The adaptation of joins, an essential operation in relational databases, under this new progressive paradigm has attracted considerable interest [13, 9, 11, 20].

However, there are cases where not all tuples are equally interesting to the user. In such cases the user would like to receive results relevant to her preferences first. If the system returns plenty of uninteresting results, this will disrupt the user's concentration and hamper interactive exploration. Hence, the progressive generation of results needs to consider user preferences as well. When a join operation is involved and assuming that a user preference is expressed as a score associated to each tuple in a table, the goal is to output join results of the highest combining preference (computed as a function on the individual table scores) first. A straightforward solution is to first join the tables and then sort the results on their combined scores. However, this is a blocking algorithm and it violates the requirements of the progressive paradigm. A more promising direction would be the use of top-k join algorithms [9, 11, 20], which can report the k most interesting join results early on. However, in a data exploration scenario, the value of k would not be defined a priori, and could end up being even equal to the total number of results, if the user decides to see the entire result set. Figure 1 demonstrates the problematic performance of the two described solutions: the blocking JoinSort (JS) algorithm depicts 0.8 seconds of "silent" processing before showing any result — during this time, top-k RankJoin (RJ) [9] produces more than 700,000 result tuples — and then yields all results "instantly", while the top-k RankJoin starts yielding results instantly, but takes more than 3 times longer than JoinSort to produce all results.

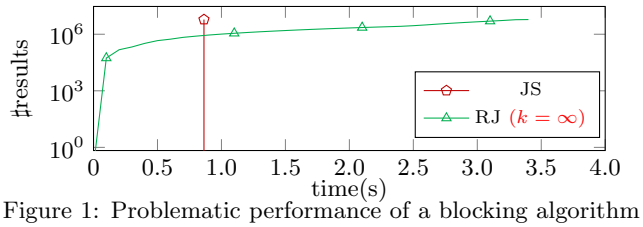


Figure 1: Problematic performance of a blocking algorithm and a representative top-k join algorithm using two TPC-H tables (cf., Table 3) and default settings (cf., Section 4).

In this paper, we investigate progressive join algorithms that take user preferences into account and need to (1) produce early results with the highest preference as quickly as possible without knowing k, and (2) progressively converge to the complete result set with low latency.

We propose *ContourJoin*, a new progressive preference-aware join algorithm. It is based on the idea of *contour lines*, which capture join results with *equal* combined preference scores. We conceptually divide the join solution space into score ranges using contour lines. Then we can process the input tables to cover the score ranges in descending order. In contrast with top-k join algorithms, there is no parameter k in ContourJoin. The algorithm sorts the results only within individual score ranges, and avoids the sorting overhead across score ranges, as they are already ordered. If the user can tolerate small predefined errors in the preference order of the results, we can further remove the inner-range sorting overhead with a relaxed ContourJoin algorithm. Hence, by following the contour lines to access the tuples of the relations, we need to buffer fewer intermediate results and reduce sorting in comparison to RankJoin.

Contributions. The key contributions of the paper are:

*Corresponding author

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

- We investigate the use of contour lines in progressive algorithms with user preferences. We propose ContourJoin that exploits contour lines to reduce sorting overhead of progressive preference-aware joins.

- We perform experiments to compare the efficiency of ContourJoin with the top-k RankJoin and the naive blocking algorithm (i.e. hash join then sort). Experimental results show that ContourJoin has superior performance in both (i) early result generation and (ii) total result computation.

Paper Organization. The remainder of the paper is organized as follows. Section 2 describes preliminaries and fundamentals. Section 3 elaborates on the algorithm design, and Section 4 presents experimental results. Section 5 discusses relevant issues, such as multi-way joins and unsorted inputs. Section 6 describes related work. Finally, Section 7 concludes the paper.

2. BACKGROUND

We define the problem of progressively joining preference-scored relations. Then, we discuss the limitations of a representative top-k join algorithm, RankJoin [9].

2.1 Problem Definition

We consider an equality join between two input tables L and R. Each table contains at least two columns, i.e. a join key and a preference value. The preference value is normalized to [0,1]. Suppose the preference value columns of tables L and R are x and y, respectively. The tuples in each table are sorted in the descending order of preference values. The preference score of a result tuple from joining tuple (key, x) from L and (key, y) from R is given by a monotonic score function $f(x, y)$. This follows the most popular way (i.e. quantitative approach) in representing preferences, where preferences are described by certain columns (e.g., price, rating, speed) with numeric values (can also be normalized to $[0, 1]$) and composed using functions [11, 2, 17, 14, 1]. A two-way preference-aware progressive join is to report matches between L and R in order of decreasing result preference. We discuss how to use the two-way preference progressive join as a building block to support multi-way preference-aware progressive joins in Section 5.

In this paper, we assume that the input tables and the join results can all fit into main memory[1]. We shall mainly focus on the weighted sum, i.e. $f(x, y) = Ax + By$, which is a popular score function [16]. Without loss of generality, $A > 0$ and $B > 0$. If $A < 0$, then let $A' = -A > 0$ and $f'(x, y) = f(x, y) + A' = A' - A'x + By = A'(1-x) + By$. We simply read L in the reverse order and compute the results using $f'(x, y)$. If $A = 0$, then the score function degenerates to $f(x, y) = By$. We discuss the support for this degenerated score function and other functions briefly in Section 3.4.

2.2 Top-k Join Algorithms Meet Any-k

Top-k join algorithms could be adapted to meet any-k scenarios. However, their performance degrades considerably as k grows.

[1]To support tables larger than memory, we can develop a solution by combining the proposed technique in this work and external sorting.

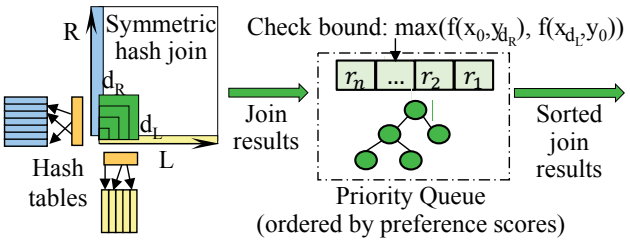


Figure 2: The RankJoin algorithm [9].

Algorithm 1: RankJoin (Any-k)

```
1  Function RankJoin(table L, table R, combining
     function f(x, y) = Ax + By)
2    x0=x_{dL}= L.first.x; y0=y_{dR}= R.first.y;
3    while L or R has next record do
4      determine next dataset to access, I;
5      record = I.getNext();
6      insert record into I's hash table;
7      probe the other hash table with record;
8      foreach valid join combination do
9        compute the join result score using f(x, y);
10       insert the join result in priority queue Q;
11     if I is L then x_{dL}= record.x ;
12     else y_{dR}= record.y ;
13     T = max(f(x0, y_{dR}), f(x_{dL}, y0));
14     while !Q.empty() and Q.top().score >= T do
15       next_result= Q.pop();
16   while !Q.empty() do
17     next_result= Q.pop()
```

We use RankJoin [9] as a representative top-k join algorithm in our analysis below. To the best of our knowledge, RankJoin is the most efficient top-k join algorithm that can be easily extended for any-k scenarios. Figure 2 illustrates the RankJoin algorithm. It performs a symmetric hash join on the two input relations L and R, as shown in the left part of Figure 2. The join result tuples are inserted into a priority queue ordered by the preference scores. However, before returning any join result to the user, RankJoin must make sure that no future join results can have larger preference scores. This is achieved using an upper bound on the preference scores of future join results.

Let us understand the bound. x_0 (y_0) denotes the preference value of the first tuple in L (R), which is the largest preference value in L (R). x_{d_L} (y_{d_R}) denotes the preference value of the last processed L (R) tuple so far by the symmetric hash join. The tuples of L are read in the descending order of preference values. Hence, for any unread tuple $l(key, x)$ in L, we know $x \leq x_{d_L}$. Tuple l may join with some R tuple $r(key, y)$. Whether r has been processed or not, $y \leq y_0$ is always true. As $f(\cdot)$ is a monotonic function, $f(x, y) \leq f(x_{d_L}, y_0)$. In other words, $f(x_{d_L}, y_0)$ is an upper bound of the preference score of joining any unread L tuple with R tuples. Similarly, $f(x_0, y_{d_R})$ is an upper bound of the preference score of joining any unread R tuple with L tuples. Therefore, $max(f(x_0, y_{d_R}), f(x_{d_L}, y_0))$ gives an upper bound of the preference score of any future join result.

L	
key	score
1	1.0
2	0.4
3	0.3
4	0.2

R	
key	score
1	1.0
2	0.4
3	0.3
4	0.2

Figure 3: An example for which RankJoin works poorly.

The original top-k RankJoin keeps a priority queue of size k. It updates the priority queue until the last tuple in the queue has a preference score greater than or equal to the bound. At this moment, RankJoin has seen all join result tuples with preference scores larger than the bound. RankJoin returns the k tuples in the queue.

We extend RankJoin to support any k (cf., Algorithm 1) as follows. First, we do not restrict the size of the priority queue. A new result tuple does not replace a tuple in the queue. Second, we output the top-most tuple t of the priority queue whenever t's score $\geq$ bound (Line 14 – Line 15).

However, the any-k RankJoin has two main sources of inefficiency: sorting overhead and score bound.

Sorting overhead. A priority queue is used to sort the join result tuples. In any-k RankJoin, a large number of low-score result tuples stay in the priority queue. For n result tuples, the queue performs $O(n \log(n))$ tuple comparison and swap operations. To make matters worse, the most efficient priority queues are implemented by heaps, which have poor CPU cache behaviors. Heap operations often incur random memory accesses, resulting in a large number of expensive CPU cache misses and TLB misses.

Score bound. The bound does not handle skewed score distributions well. Figure 3 shows an example. Suppose $f(x, y) = x + y$. Even if all tuples are processed, the bound is 1.2, which is still quite large. This is larger than the score of all but the first join result tuple. Thus, RankJoin can produce only one early result tuple. The rest will be output at the end of the join.

3. CONTOURJOIN

In this section, we introduce the idea of contour lines, a method for partitioning the result space of a join into score ranges. Based on this idea, we propose *ContourJoin*, a progressive preference-aware join algorithm, and describe four variants of it. Finally, we briefly discuss support for other score functions.

3.1 Contour Lines and Contour Bounds

Assuming that input tables are ordered on preference values, we divide the tuples into ranges based on the preference values. For example, in Figure 4, x-axis and y-axis represent the preference value space for each table. Table L and R are divided into p_L and p_R ranges, respectively. In each range L_i of table L, $x \in (1 - \frac{i+1}{p_L}, 1 - \frac{i}{p_L}]$. In each range R_i of table R, $y \in (1 - \frac{j+1}{p_R}, 1 - \frac{j}{p_R}]$. Note that p_L and p_R satisfy $\frac{A}{p_L} = \frac{B}{p_R}$, which ensures that L and R contribute to the final score equally. The diagonal lines in Figure 4, each delimiting a contoured area, are called *contour lines*. Final scores on a contour line are the same.

Features. Contour lines have the following features.

1. Input data which lie between any two successive lines generate join results with the same score range;

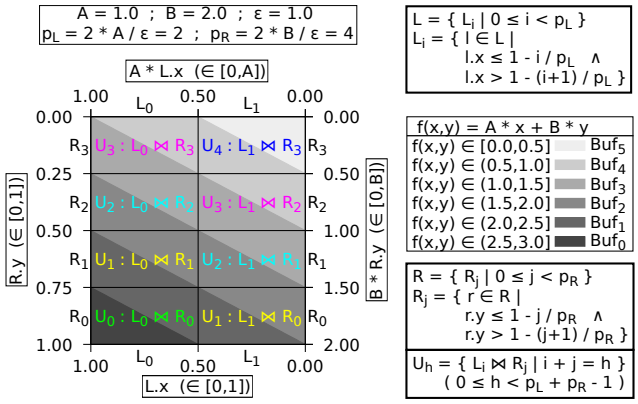


Figure 4: Contour lines.

2. As contour lines go forward (i.e. from bottom-left to top-right), the result score range that they cover decreases.

Essentially, a contour line determines the score bound (contour bound) between the two areas that it divides, which can be exploited to output join results with higher scores first. Unlike previous top-k join algorithms where the threshold values have to be updated every time after reading an input record (in order to output early results), the contour bounds are constant, and not limited by the top-first preference values of the input data. Hence, by accessing the input tuples in the order of the contour lines, our proposed ContourJoin can avoid unnecessary caching of join matches and cluster results into buffers to reduce sorting overhead.

3.2 The ContourJoin Algorithm

The join operation can be viewed as the process of spanning the space of contour lines to get valid join combinations. An important observation is that, after spanning a line, parts of results can be output early. Algorithm 2 shows the generic framework for ContourJoin algorithm. The algorithm works as follows.

1. (Line 3) Compute parameter p_L and p_R.
2. (Line 4) Retrieve and process input data in a loop, following the contour lines (i.e. from the highest to the lowest score range).
3. (Line 5 – Line 8) For each tuple that resides in the rectangle covered by the line, generate join combinations using a join strategy (cf., Sec. 3.3).
4. (Line 18) For each resulting join combination, compute the combined score using the score function f.
5. (Line 19 – Line 20) Map each join result to a buffer based on its combined score. For example, in Figure 4 Buf_0 contains results with a score between 2.5 and 3.0. In this way, the algorithm avoids the sorting cost across buffers.
6. (Line 21 – Line 25) If the lower bound of a buffer is larger than the contour bound (i.e. the bound of the current line), sort (if needed) and output the buffer. Join results over the following contour lines cannot have higher scores than the current contour bound.

Algorithm 2: ContourJoin Generic Framework

```
1  Function ContourJoin(table L, table R, combining
     function f(x,y) = Ax + By, ε)
2      initialize a set of buffers;
3      Divide2Ranges(L, R, f, ε);
4      while (cur_line_id=getNextContourLine()) and
         cur_line_id < |contour lines| do
5          next_L = L.getNextRange();
6          next_R = R.getNextRange();
7          /* depends on join strategy */
8          Join(next_L, next_R, f);
9          OutputBuffer(cur_line_id);
10 Function Divide2Ranges(table L, table R, combining
     function f(x,y) = Ax + By, ε)
11     if relaxation is allowed then
12         calculate integer p_L, p_R:
13             A/p_L = B/p_R and A/p_L ≤ ε/2
14     else
15         calculate integer p_L, p_R: A/p_L = B/p_R
16     divide L and R into p_L, p_R ranges, respectively
17 Function MapJoinResult(a join result < l_i, r_j >,
     combining function f(x,y))=Ax+By)
18     calculate score s = f(l_i, r_j);
19     buf_id = getBufID(s) ;
20     push the result to buffer[buf_id];
21 Function OutputBuffer(contour line lid)
22     buf_id = getOutputBufID(lid);
23     if relaxation is not allowed then
24         sort results in buffer[buf_id];
25     output results in buffer[buf_id] ;
```

Algorithm 3: Join Inputs Follow Contour Lines

```
1  Function Join(partition next_L, partition next_R,
     combining function f(x,y) = Ax + By)
2      for each r ∈ next_L do
3          insert r into L's hash table;
4          probe R's hash table to find join matches;
5          for each join match < l_i, r_j > do
6              MapJoinResult(< l_i, r_j >, f);
7      for each r ∈ next_R do
8          insert r into R's hash table;
9          probe L's hash table to find join matches;
10         for each join match < l_i, r_j > do
11             MapJoinResult(< l_i, r_j >, f);
```

Algorithm 4: Both Join Inputs and Join Results Follow Contour Lines

```
1  Function Join(partition next_L, partition next_R,
     combining function f(x,y))
2      if next_L(next_R) is not null then
3          cache next_L(next_R);
4      for each rectangle rect covered by the current
         contour line do
5          Partition l_i = rect.getPartitionFromL();
6          Partition r_j = rect.getPartitionFromR();
7          compute join matches on l_i and r_j using
             existing join methods (e.g., sort merge join,
             hash join, ...);
8          for each join match < l_i, r_j > do
9              MapJoinResult(< l_i, r_j >, f);
```

Buffer Maintenance. ContourJoin exploits the features of contour lines to temporarily cache results into buffers. Join results that lie between any two successive contour lines are mapped to the same buffer. In this way, inter-buffer sorting is avoided. The algorithm can output early results in a buffer when the score range of the buffer is greater than or equal to the current contour line. The number of buffers to maintain and whether intra-buffer sorting is necessary depend on the specific join strategy (cf., Section 3.3.1) and relaxation policy (cf., Section 3.3.2).

3.3 Variants of ContourJoin

ContourJoin can be viewed as a generic framework for any-k join algorithms. For example, when $p_L = p_R = 1$, it becomes a blocking algorithm, i.e. join everything first and then sort; when p_L and p_R are large enough, it is similar to RankJoin, which takes a single tuple as an input. Furthermore, we can have an approximate version of the algorithm by adding an appropriate parameter. We discuss variants of the algorithm below.

3.3.1 Join Strategy

The order in which the points in the result space are checked has a great effect on the intermediate result size. The larger the intermediate result, the more buffers are required to cache results, and the more cache misses and TLB misses. Figure 5 presents two strategies and the join combinations they produce.

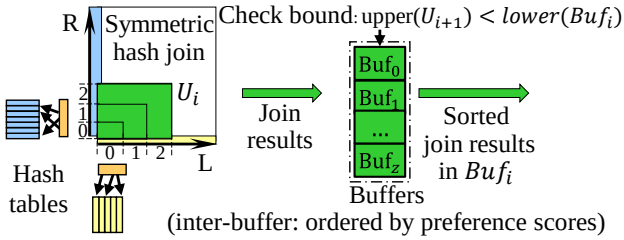
The first join strategy (Figure 5a) ensures that join inputs follow contour lines. It spans the space of the Cartesian product of the input relations by exploiting symmetric hash join, which is a popular non-blocking join algorithm [4, 7]. Algorithm 3 shows the join algorithm. At each invocation, it retrieves input data from the next range of both input relations, and computes valid join combinations on all tuples seen so far (Line 2 – Line 11). The generated join results may lie in many contour areas. A maximum of $|contour\ lines| + 1$ (where $|contour\ lines| = p_L + p_R - 1$) buffers are required for caching results. The total buffer size for the whole process is $O(total\_results)$.

On the other hand, the second strategy (Figure 5b) ensures that both join inputs and join results follow contour lines. It exploits contour lines' features to span the contour space. At each invocation, it follows the next contour line to retrieve input data and then computes join matches only on the data that lie in the rectangles covered by the line (Line 4 – Line 9 in Algorithm 4). New results lie in at most 2 adjacent contour areas, and results in the higher contour area can be output first. Therefore, at most 2 buffers are required for caching results. When join results are uniformly distributed in the result space, the size of the intermediate buffer for the whole process is $O(\sqrt{total\_results})$.

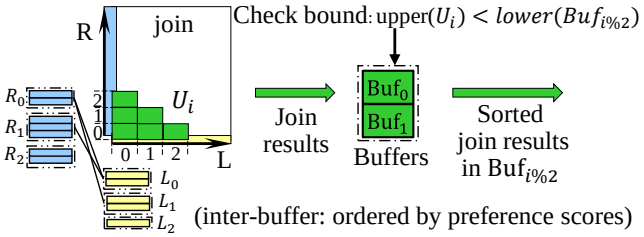
3.3.2 Relaxation of Output Order

Table 1: Notations.

Notation	Description	Notation	Description
$upper(RRec_{(i,j)})$	upper-bound value of output results for $L_i \bowtie R_j$	n_L, n_R, n_O	number of L, R, output records
$lower(RRec_{(i,j)})$	lower-bound value of output results for $L_i \bowtie R_j$	$c_{compute}$	computation time
$\beta_1, \beta_2, \beta_3$	coefficient (build, probe, sort cost per tuple)	γ	join selectivity
$RRec_{(i,j)}$	number of output results for $L_i \bowtie R_j$	$\|R\|$	number of inputs in total
α	coefficient (1/0 for precise/relaxed ContourJoin)	c_{join}	join time
$upper(U_k)$	upper-bound value of output results in U_k	c_{sort}	sort time
$lower(U_k)$	lower-bound value of output results in U_k	$\|L\|$	number of inputs in total



(a) Join inputs follow contour line



(b) Both join inputs and join results follow contour line

Figure 5: ContourJoin algorithms.

In real-world applications there are cases where the exact output order of the results based on preference might not matter so much for the user. Therefore, we can relax the sorting precision by allowing a small error in the order of tuples with very similar scores.

Definition 1 (Relaxation of Order). *For any two tuples t_i, t_j, if $|t_i.score - t_j.score| <= \epsilon$, then t_i and t_j are regarded as in order.*

If we ensure that the absolute difference between any two successive contour lines, i.e. the maximum absolute difference of any two results within a buffer, is not greater than ϵ, then there is no need to sort the buffers that cache the results (Line 23 – Line 25 in Algorithm 2).

3.3.3 Number of Partitions

The number of partitions, and consequently the interval of contour lines, is determined by p_L and p_R. We consider three goals for choosing p_L and p_R: (i) satisfy the properties of the contour lines so that partitions contribute to the final score equally; (ii) output early high-score results within t_{first} (e.g., t_{first}=0.5s [18]) to offer a more interactive user experience; (iii) take the relaxation of order into consideration to avoid sorting.

Goal (i) is achieved by satisfying $\frac{A}{p_L} = \frac{B}{p_R} = \rho$. We proceed with finding the value of ρ that satisfies the other two goals.

Goal (ii) requires computing the first contour space within t_{first}, that is, $c_{compute} \leq t_{first}$. We have to estimate the input data sizes n_L, n_R and the output result size n_O of the first contour space. For simplicity, we assume that both join keys and scores are uniformly distributed (if not, n_L, n_R, n_O can be sampled). Table 1 lists notations used in this paper.

Theorem 1. *If join keys and scores are uniformly distributed, and ρ satisfies that $\frac{\beta_1\rho|L|}{A} + \frac{\beta_2\rho|R|}{B} + \alpha\beta_3\frac{\gamma\rho^2|L||R|}{AB}log(\frac{\gamma\rho^2|L||R|}{AB}) \leq t_{first}$, then goal (ii) is achieved.*

Proof. Since scores are uniformly distributed, we have $n_L = \frac{|L|}{p_L} = \frac{\rho|L|}{A}$ and $n_R = \frac{|R|}{p_R} = \frac{\rho|R|}{B}$. Since join keys are uniformly distributed, we have $n_O = \gamma n_L n_R = \frac{\gamma\rho^2|L||R|}{AB}$.

$c_{compute} = c_{join} + \alpha c_{sort} \leq t_{first}$

$\Rightarrow \beta_1 n_L + \beta_2 n_R + \alpha\beta_3 n_O log n_O \leq t_{first}$

$\Rightarrow \frac{\beta_1\rho|L|}{A} + \frac{\beta_2\rho|R|}{B} + \alpha\beta_3\frac{\gamma\rho^2|L||R|}{AB}log(\frac{\gamma\rho^2|L||R|}{AB}) \leq t_{first}$. □

Goal (iii) is achieved by ensuring that (1) the score difference between any two results which reside in the bottom-left rectangle (i.e. L_0, R_0) is not greater than ϵ (Equation 1) and (2) the score difference between any two results which reside in the area between any two successive lines is not greater than ϵ (Equation 2).

$$upper(RRec_{(0,0)}) - lower(RRec_{(0,0)}) \leq \epsilon \quad (1)$$

$$upper(U_i) - lower(U_i) \leq \epsilon \quad (2)$$

Theorem 2. *If $\rho \leq \frac{\epsilon}{2}$, then goal (iii) is achieved.*

Proof. $upper(RRec_{(i,j)}) - lower(RRec_{(i,j)}) = 2\rho \leq \epsilon$, so Equation 1 is satisfied.

$upper(U_k) - lower(U_k)$

$= max(upper(RRec_{i,k-i})) - min(lower(RRec_{i,k-i}))$

$= (A+B-k\rho) - (A+B-(k+2)\rho) = 2\rho \leq \epsilon$, so Equation 2 is satisfied. □

3.3.4 Four Variants

Based on the aforementioned parameters and design choices, we implement four variants of ContourJoin, as shown in Table 2. We evaluate the performance of all variants in Section 4.

3.4 Supporting Other Combining Functions

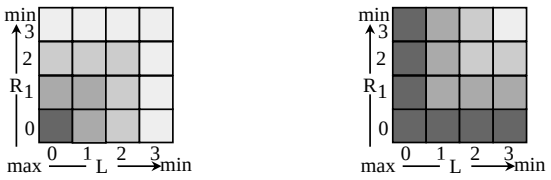
For a combining score function, we can compute the shape of its contour lines, then employ ContourJoin.

f(x,y)=By. When $A = 0$, $f(x, y) = Ax + By$ degenerates to $f(x, y) = By$. The contour lines are determined only by R tuples, and are horizontal lines. Then ContourJoin essentially performs a simple hash join. It first builds the hash table on L, then probes the hash table for R tuples

Table 2: Four Variants of ContourJoin (CJ).

Variants	Follow Contour Lines		Relaxation
	Join Inputs	Join Results	
CJpI	✓		
CJpB	✓	✓	
CJrI	✓		✓
CJrB	✓	✓	✓

p: precise, r: relaxed, I: Inputs, B: Both inputs and results



(a) Min(x,y) (b) Max(x,y)

Figure 6: Contour lines of other functions (Rectangles filled with the same color are covered by the same contour line).

with decreasing preference values, progressively generating join results.

Min(x,y). The contour lines of $\min(x, y)$ are "¬" shaped (Figure 6a) . In this case, contour lines correspond to space covered by the symmetric hash join. We can apply strategy (a) of ContourJoin to generate early results progressively.

Max(x,y). The contour lines of $\max(x, y)$ are "⌞" shaped (Figure 6b) . Unfortunately, all input tuples from both tables may contribute to join results with the maximal scores. Hence, all input tuples must be processed before generating any output results. Neither ContourJoin nor other top-k join algorithms can output progressive results for function $\max(x, y)$.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of ContourJoin. We would like to answer the following questions:

- How does ContourJoin compare to the fastest blocking algorithm and the efficient top-k rank join algorithm?
- How effective are our proposed algorithm optimization techniques (i.e. join strategy, relaxation)?

4.1 Experimental Setup

Machine Configuration. The experiments are performed on a machine equipped with one Intel Core i7-4770 CPU @ 3.40 GHz (8 MB cache) and 32 GB memory, running 64-bit Ubuntu 16.04 LTS with 4.15.0-62-generic Linux kernel.

Algorithms. We implement all algorithms in C++ using the same data structures for the same functionality and compile them using the GNU C++ compiler with optimization option `-O3`. Input data is loaded into memory and IO costs are ignored. Output results are cached in memory.

We compare ContourJoin with two baselines: (1) JoinSort (JS), a baseline for total result computation. It computes the full results using hash join[2], then sorts all results. (2) any-k RankJoin (RJ) (cf. Section 2.2), a baseline for early result generation. As for ContourJoin, CJpI and CJrI are implemented using symmetric hash join, and CJpB and CJrB are implemented using sort merge join[3] on individual partitions.

Datasets. We generate data based on two TPC-H tables, i.e. Lineitem and Partsupp. In Lineitem (Partsupp), we use l_partsupp and l_suppkey (ps_partsupp and ps_suppkey) as a composite key to generate the key column, and l_discount (ps_availqty) to generate the preference column. Each generated tuple consists of a 32-bit integer key and a normalized decimal preference value (a.k.a score) in [0, 1].

We generate three datasets based on the two base TPC-H tables. For datasets♯1, we fix the input table sizes while varying the output result size by a factor of m_1, where scale factor m_1=1, 4, 16, or 64. For datasets♯2, we fix the output result size while varying the input table sizes by a factor of m_2, where scale factor m_2=1, 4, 16, or 64. For datasets♯3, we vary both the input sizes and the output result size by a factor of m_3, where scale factor m_3=1, 4, 16, or 64. Details of data generation and statistics are presented in Appendix A.

The larger the input size, the higher the join cost. The larger the output size, the higher the sort cost. Therefore, using datasets♯1, we can observe increasing sort overhead as m_1 increases. Using datasets♯2, we can observe increasing join overhead as m_2 increases. Datasets♯3 is between datasets♯1 and datasets♯2, both join overhead and sort overhead increase as m_3 increases.

We run all experiments and report the results for scale factors 1 and 64 (the cases for scale factors 4 and 16 show similar trends).

Query. The query in the experiments is as follows. It performs an equality join on Lineitem and Partsupp. A score function is computed on the join result to obtain a combined score. The join results are ordered by the combined scores. The query is progressively computed except for the blocking JoinSort algorithm, which produces all results at the end of the query processing.

```
select L.key, f(L.score, PS.score) as score
from Lineitem as L, Partsupp as PS
where L.key = PS.key
order by score desc
any k
```

Parameters. In our experiments, the combining score function is $f(x, y) = Ax+By$. We consider two cases: $A = B = 1$ (by default) and $A = 10, B = 1$. For $A = B = 1$, we set p_L=200 and p_R=200. For $A = 10, B = 1$, we set p_L=2000 and p_R=200. We set the relaxation error $\epsilon = 0.01$ for the relaxed ContourJoin algorithms, i.e. CJrI and CJrB.

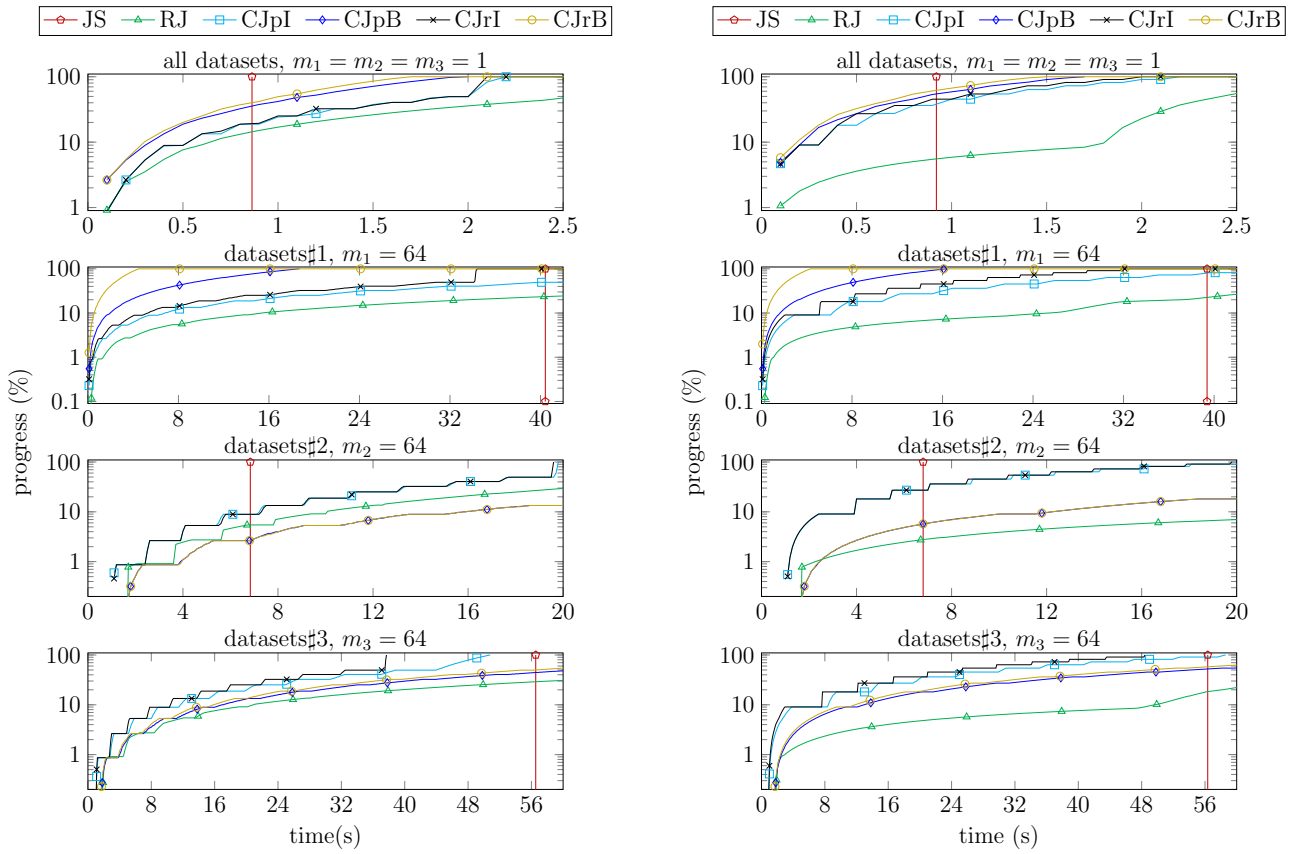
4.2 Experimental Results

4.2.1 Overall Performance

Figure 7 presents the progress of preference-aware join computation on various datasets over time, while varying scale factors. In the figure, x-axis is the elapsed time and y-axis shows the ratio between the number of results produced so far and the total number of results. We report the generated number of results every 0.1 s. Note that JoinSort is a blocking algorithm. It generates all results at the end of the computation.

[2]We build the hash table on the smaller table.

[3]We find that sort merge join is more efficient than hash join for Strategy (B) because hash join incurs large amount of TLB misses, especially when p_L or p_R grows.



(a) $f(x, y) = x + y$, $p_L = 200$, $p_R = 200$

(b) $f(x, y) = 10x + y$, $p_L = 2000$, $p_R = 200$

Figure 7: Fraction of result records produced over time for various datasets varying scale factors.

From Figure 7, we see that the best ContourJoin significantly outperforms RankJoin for computing both early results and full results. More precisely, compared to RankJoin, the best precise ContourJoin achieves 1.0x–7.0x (1.4x–7.7x) improvements for obtaining the top 1% (10%) early results, and 1.4x–10.6x improvements for computing full results. Relaxation further improves performance. The best relaxed ContourJoin achieves 1.0x–14.0x (1.4x–50.8x) improvements for obtaining the top 1% (10%) early results, and 1.4x–39.4x improvements for computing full results. Moreover, compared to JoinSort, the best ContourJoin obtains comparable or better performance for computing full results. The ratio between the execution time of JoinSort to that of the best ContourJoin is between 0.3 and 9.0.

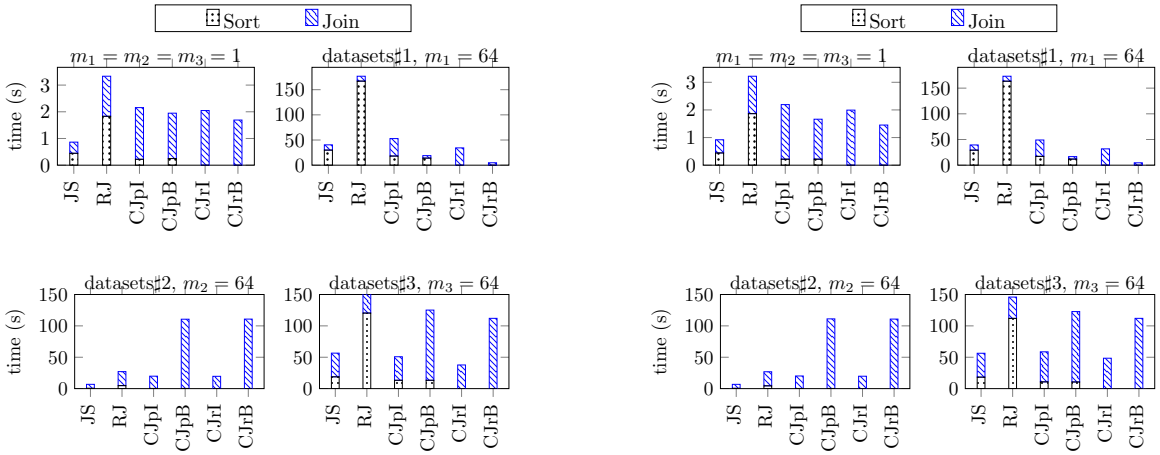
ContourJoin achieves good performance for the following main reasons. First, ContourJoin exploits contour lines to avoid inter-buffer result sorting. Figure 8 shows the execution time breakdowns for all the experiments in Figure 7. For an experiment, the height of the corresponding bar indicates the total time for computing full results. The bar is broken down into two parts. The lower part shows the time spend in sorting results and the upper part shows the time taken by the join operation. From the figure, we see that compared to RankJoin and JoinSort, the best precise ContourJoin reduces sorting time by 8.2x–19.1x and 1.4x–2.1x, respectively. Second, relaxed ContourJoins further avoid result sorting within individual buffers by using the relaxation policy, leading to zero sort times, as shown in Figure 8. Third, the join times of the best ContourJoin are comparable to that of JoinSort. Finally, the second ContourJoin strategy (i.e. CJpB and CJrB), where both join inputs and join results follow contour lines, use up to two buffers, thereby significantly reducing the intermediate result sizes, as shown in Figure 9.

In the following subsections, we study the performance impact of different scale factors, different datasets, and different combining score parameters to better understand the behavior of ContourJoin.

4.2.2 *Base Case: Scale Factor=1, f(x,y)=x+y*

When the scale factor (i.e. m_1, m_2, and m_3) is 1, all three datasets are the same. From Figures 7(a) and 8(a), we see that the best ContourJoins (i.e. CJrB and CJpB) produce both early results and full results significantly faster than RankJoin. Note that the RankJoin curve in Figures 7(a) does not show the full execution of RankJoin. The total execution time of RankJoin can be seen in Figure 8(a).

On the other hand, we see that ContourJoin is slower than JoinSort for generating full results. From Figure 8(a), we see that ContourJoin spends longer time in performing the join operation. The two input tables, i.e. Lineitem and Partsupp, have very different sizes. There are about 6 million records in the Lineitem table, but only 800 thousand in the Partsupp table. JoinSort builds a hash table on the smaller input table, Partsupp, and probes the hash table using each Lineitem record. In contrast, CJpI and CJrI are based on symmetric hash join, which builds and probes a second hash table on the much larger Lineitem table. This increases the



(a) $f(x, y) = x + y$, $p_L = 200$, $p_R = 200$

(b) $f(x, y) = 10x + y$, $p_L = 2000$, $p_R = 200$

Figure 8: Execution Time Breakdown: Total = Sort + Join.

join times significantly. CJpB and CJrB compute joins for individual rectangles. Consider a contour range L_i. It has to be joined with each of the p_R contour ranges in R. Therefore, L records are processed p_R times for finding matches in individual rectangles. Similarly, R records are processed p_L times. We employ sort merge joins to sort each range once and then use efficient merging to find matches for individual rectangles. However, even with this optimization, the join overhead is larger than that in JoinSort.

4.2.3 *Effect of Increasing Scale Factor*

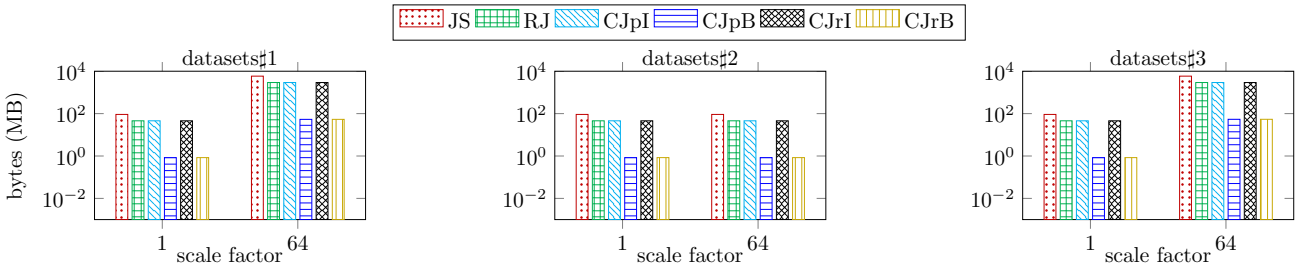
In this section, we study how algorithms perform when scale factor increases. We consider combing score function $f(x, y) = x + y$ here, and discuss effect of different combing score function in Section 4.2.4.

Datasets♯1: increasing result sorting time. As scale factor m_1 increases from 1 to 64, the input size is not changed, but the output size increases by 64x. This results in increasing sorting time. From Figures 8(a), we see that in RankJoin, the result sorting time increases by 91.2x but the joint time increases by only 6.5x. RankJoin uses heap-based structure for the priority queue, which sees significantly higher cost with increasing result records. Similarly, in JoinSort, the result sorting time increases by 67.2x as the output size increases. In ContourJoin, the result sorting (join) time of the best precise ContourJoin increases by 56.5x (2.3x), and the sort time of the best relaxed ContourJoin is 0. Precise ContourJoin removes inter-buffer result sorting, and intra-buffer result sorting is further removed by relaxation. To sum up, as the output result size increases, ContourJoin sees the smallest performance impact, and therefore ContourJoin's performance advantage for computing early results and full results becomes larger compared to RankJoin and JoinSort.

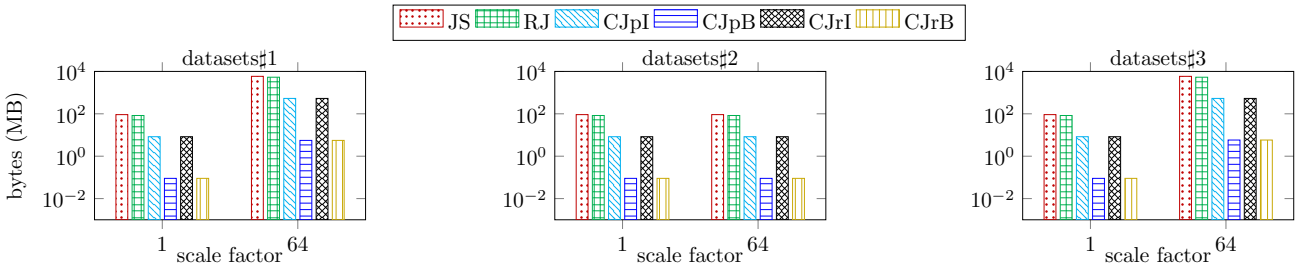
Datasets♯2: increasing join time. As m_2 increases from 1 to 64, the output size is unchanged but the input size increases by 64x. This results in increasing join time. From Figures 8(a), we see that the join times of RankJoin and JoinSort increase by 15.3x and 14.9x, respectively. The join times of CJpI and CJrI increase by 10.1x and 9.1x, respectively. Therefore, the performance comparison of CJpI/CJrI with RankJoin and JoinSort for m_2=64 is similar to that in the base case. However, the join times of CJpB and CJrB show more drastic 52.6x and 65.1x increases. This is because their join strategy computes matches for individual rectangles in the result space. Every input record is matched multiple times with different contour ranges in the other table. As a result, CJpI and CJrI become the best precise and relaxed ContourJoin algorithms when the input size increases.

Datasets♯3: increasing join and result sorting time. As m_3 increases from 1 to 64, both the input size and the output size increase by 64x. This results in increasing join and sort time. From Figures 8(a), we see that the join (result sorting) times of RankJoin and JoinSort increase by 22.7x (65.8x) and 90.4x (42.2x), respectively. Interestingly, the change in JoinSort's join time is much more drastic compared to RankJoin. JoinSort builds a hash table on the smaller Partsupp table and probes the hash table with records in Lineitem table. As m_3 increases, the number of records in Partsupp increases by m_3 times but the number of unique keys is kept the same. As a result, the number of records per join key increases by m_3 times. All m_3 records with the same join key now goes into the same hash bucket. This makes the hash table probing less efficient. On the other hand, RankJoin employs symmetric hash join, which builds and probes hash tables on both Partsupp and Lineitem. As m_3 increases, the number of duplicate records per unqiue join key does not change in Lineitem table. Therefore, the efficiency of accessing the hash table on Lineitem is roughly the same, while accessing the hash table on Partsupp becomes less efficient. The combined effect gives a more mild impact on the join time for RankJoin. As for ContourJoin, the join (result sorting) times of CJpI and CJpB increase by 19.3x (60.1x) and 53.2x (53.6x), respectively. The join times of CJrI and CJrB increase by 18.0x and 65.9x, respectively; their sorting times are 0. CJpB and CJrB have larger increased join times because of the join strategy, as analyzed in the case of datasets♯2.

Summary. For full computation performance, we see that the join algorithms see increasing join times as the input size increases, and they see increasing result sorting times as the output size increases. For early result generation performance, we see from Figure 7(a) that the input size has



(a) $f(x, y) = x + y$, $p_L = 200$, $p_R = 200$



(b) $f(x, y) = 10x + y$, $p_L = 2000$, $p_R = 200$

Figure 9: Maximum intermediate result size.

larger impact on the performance for generating top 1% results than the output size for ContourJoin. If we fix the output size and vary the input size (e.g., increasing m_2 from 1 to 64 in datasets♯2), the times to generate top 1% results increase by 12.5–40.0x for ContourJoin algorithms. In contrast, if we fix the input size and vary the output size (e.g., increasing m_1 from 1 to 64 in datasets♯1), the times to generate top 1% results increase modestly by 1–3x for ContourJoin algorithms. We can also compare the case of datasets♯2 m_2=64 and the case of datasets♯3 m_3=64. The two cases also have the same input size but different output sizes. The times to generate top 1% results differ by only 0.9–1.2x for the two cases.

4.2.4 *Effect of Different Function Parameters*

We study how algorithms perform when the parameters of the combining score function $f(x, y) = Ax + By$ change. As A increases from 1 to 10, the join time and result sorting time do not have large changes, as shown in Figure 8. For example, for JoinSort, the change in the score function does not impact the behavior of the join and the sorting in the algorithm. Therefore, its performance does not vary significantly for different function parameters. We focus our discussion on early result generation below.

For RankJoin, as A increases, the contributions of preference values from the two tables to the combined score become less balanced, resulting in more skewed result score distribution. Due to the limitation of RankJoin's score bound, more results with low preference are generated early and cached, and the maximum intermediate result size increases. This explains why RankJoin has worse early result performance when A is larger, as shown in Figure 7. For example, it can produce 23% fewer early results for datasets♯2 and 13% fewer early results for datasets♯3 within the same given time when scale factor is 64.

For ContourJoin, as A increases from 1 to 10, the number of input partitions (i.e. p_L) increases by 10x and there are more contour lines. Each contour line tends to cover less input data, and each buffer tends to hold fewer results. As the buffers become finer-grain, fewer intermediate results are cached (see Figure 9) and more results are output early. Therefore, ContourJoin has better early result performance when A increases (e.g., 5% – 14% improvements when scale factor is 64 in all datasets), as shown in Figure 7.

5. DISCUSSION

Multi-way Join. We mainly study two-way preference-aware progressive joins. Our proposed ContourJoin algorithms can be applied to support multi-way preference-aware progressive joins. For a multi-way preference-aware progressive join, we assume that the input tables A_1, A_2, ..., A_m are all sorted according to the descending order of their preference values. The preference score of a join result tuple is given by a linear score combining function $f(x_1, x_2, ..., x_m) = a_1x_1 + a_2x_2 + \cdots + a_mx_m$. A multi-way join can be computed as a tree of two-way join operations. Therefore, we would like to compute the multi-way preference-aware progressive join as a tree of two-way preference-aware progressive joins. To achieve this, we can break down the m-variable linear function $f(\cdot)$ into a set of two-variable functions that correspond to each two-way join in the tree. Then, we employ precise ContourJoins for all but the final two-way join at the root of the tree. A precise ContourJoin will generate results in the descending preference order. Hence, the join results of a child precise ContourJoin operator can be used as input to a parent ContourJoin operator. For the final join at the root, we can employ either a relaxed or precise ContourJoin depending on whether the relaxed output order is sufficient.

Unsorted Input. In this work, we follow previous top-k studies, including TA [5], J* [15], and RankJoin [9], to assume sorted inputs. This assumption is needed so that the first join result can be generated without reading all the input tables. If inputs are unsorted according to the preference

values, we have to first sort the inputs on the fly, then apply preference-aware progressive join algorithms. However, the cost of input sorting can substantially limit the capability of displaying early results fast. It is interesting to design auxiliary data structures and optimize the input sorting process. We leave a detailed study on unsorted inputs as future work.

6. RELATED WORK

Progressive Paradigms. There are several progressive paradigms. The first representative application is online aggregation [7, 13]. It reports approximate answers with quality guarantee (usually in the form of confidence intervals), accuracy is improved as more time is spent, but it can only report aggregation results. The second paradigm [6, 3] reports approximate answers, but it will return final precise results as long as it takes enough time. The third paradigm [21] always returns precise results progressively, more results are generated as time goes by. ContourJoin follows the third paradigm.

Top-k and Any-k Algorithms. Fagin's threshold algorithm (TA) [5] is one of the best-known top-k approaches. However, it solves top-k selection problem [10], which is not a general case of join. Unlike top-k selection, top-k join querie [9, 19, 10] solves more general join problem. It assumes each record in a table has a score, and reports only k (k is known apriori) join results with top-ranked scores. Related literature has three common points. First, input tables are sorted by scores. If not, this can be handled in preprocessing step. Second, they store only k results with highest score, at any time of computing, any subsequent results with lower score are pruned, and those with higher scores replace earlier results. Finally, they often derive a score bound on future join results for fast pruning results with low scores. They achieve good performance when the number of top-ranked tuples that have to be accessed is quite small. One of the best top-k join algorithm is RankJoin [9]. There are some work [22] that performs better than RankJoin. However, they treat join as a path query in a graph. They assumes that the join relationship between records in different tables are already known and can be represented as a graph (a.k.a join graph). In the paper, we focus on the general join case without knowing k apriori, results are progressively generated. We propose a novel bound-based technique and compare with RankJoin. To the best of our knowledge, RankJoin is the most efficient top-k join algorithm that can be easily extended for the problem. Previous any-k join [21] shares the same problem as our study. However, it is based on a join graph.

Exploiting Hyperplanes for Query Processing. Khan et al. [12] propose Planar index for fast pruning d-dimensional data points for scalar product queries with $f(x_1, ..., x_n) = \sum_1^n a_i x_i \leq b$, without actually computing the scalar product. Their method is based on the assumption that query fields are known apriori, so that inputs can be pre-sorted in ascending order of the query field values. Then, an index is built by creating hyperplanes on sorted data according to function parameters. In this way, data points are pruned according to the interval of the hyperplanes. If we consider 2-dimensional data, the constructed index is actually a set of lines. Different from our work, the way to create lines proposed in the paper is not targeted at maintaining contour areas. What's more, we propose ContourJoin algorithms for progressively joining two (or more) tables. This is quite different from computing scalar product queries on tables.

7. CONCLUSION

In this paper, we study progressive preference-aware join algorithms to provide good performance in terms of both early result generation and total result computation for interactive data analysis and exploration. We propose ContourJoin, which exploits contour lines for generating results in the descending order of user preference. Following contour lines, we compare different join strategies and introduce relaxation by controlling line intervals. These techniques help reduce intermediate result size and sorting overhead. Experimental results show that ContourJoin has comparable or better performance to the fast blocking algorithm (i.e. hash join and sort) and the efficient top-k rank join algorithm (i.e. RankJoin).

8. ACKNOWLEDGMENTS

The first and the second authors are partially supported by National Key R&D Program of China (2018YFB1003303) and K.C.Wong Education Foundation. Shimin Chen is the corresponding author.

9. REFERENCES

[1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *SIGMOD*, pages 297–306, Dallas, Texas, USA, May 2000. ACM.

[2] A. Arvanitis and G. Koutrika. Towards preference-aware relational databases. In *ICDE*, pages 426–437, Washington, Apr. 2012. IEEE Computer Society.

[3] B. Chandramouli, J. Goldstein, and A. Quamar. Scalable progressive analytics on big data in the cloud. *Proc. VLDB Endow.*, 6(14):1726–1737, 2013.

[4] S. Chen, P. B. Gibbons, and S. Nath. Pr-join: a non-blocking join achieving higher early result rate with statistical guarantees. In *SIGMOD*, pages 147–158, Indianapolis, IN, USA, June 2010. ACM.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[6] A. Gogolou, T. Tsandilas, K. Echihabi, A. Bezerianos, and T. Palpanas. Data series progressive similarity search with probabilistic quality guarantees. In *SIGMOD*, pages 1857–1873, Portland, OR, USA, June 2020. ACM.

[7] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, Philadelphia, PA, USA, June 1999. ACM.

[8] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, pages 277–281, Melbourne, Vic, Australia, June 2015. ACM.

[9] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, Berlin, Germany, Sept. 2003. VLDB End.

[10] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.

[11] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Prefjoin: An efficient preference-aware join operator. In *ICDE*, pages 995–1006, Hannover, Germany, Apr. 2011. IEEE.

[12] A. Khan, P. Yanki, B. Dimcheva, and D. Kossmann. Towards indexing functions: answering scalar product queries. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 241–252. ACM, 2014.

[13] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, May 2019.

[14] N. Meneghetti, D. Mindolin, P. Ciaccia, and J. Chomicki. Output-sensitive evaluation of prioritized skyline queries. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *SIGMOD*, pages 1955–1967, Melbourne, Victoria, Australia, May 2015. ACM.

[15] A. Natsev, Y. Chang, J. R. Smith, C. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 281–290. Morgan Kaufmann, 2001.

[16] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM Trans. Database Syst.*, 36(3):19:1–19:45, 2011.

[17] B. Tang, K. Mouratidis, M. L. Yiu, and Z. Chen. Creating top ranking options in the continuous option and preference space. *Proc. VLDB Endow.*, 12(10):1181–1194, 2019.

[18] W. Tao, X. Liu, Ç. Demiralp, R. Chang, and M. Stonebraker. Kyrix: Interactive visual data exploration at scale. In *CIDR*, Asilomar, CA, USA, Jan. 2019. www.cidrdb.org.

[19] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, pages 277–288, Bangalore, India, 2003. IEEE.

[20] N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proc. VLDB Endow.*, 13(9):1582–1597, 2020.

[21] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Optimal join algorithms meet top-k. In *SIGMOD*, pages 2659–2665, Portland, OR, USA, 2020. ACM.

[22] M. Wu, L. Berti-Équille, A. Marian, C. M. Procopiuc, and D. Srivastava. Processing top-k join queries. *Proc. VLDB Endow.*, 3(1):860–870, 2010.

APPENDIX

A. DATA GENERATION AND STATISTICS

Table 3 summarizes the data characteristics of the two TPC-H tables, i.e. Lineitem and Partsupp. We call the original TPC-H tables *base tables*. In Lineitem (Partsupp), we use l_partsupp and l_suppkey (ps_partsupp and ps_suppkey) as a composite key to generate the key column, and l_discount (ps_availqty) to generate the preference column. We generate the following three datasets based on the two base tables, while keeping the key distribution and preference score distribution the same as base tables. Each generated tuple consists of a 32-bit integer key and a normalized decimal preference value (a.k.a score) in [0, 1].

- **Datasets♯1 (with fixed input size and increasing output size)**: We fix the input table sizes while varying the output result size by a factor of m_1, where m_1=1, 4, 16, or 64. To achieve this, we reduce the number of unique keys in each table by rewriting every original key *orig_key* derived from the base tables as $new\_key = \lfloor \frac{orig\_key}{m_1} \rfloor$. In this way, each Lineitem record matches m_1 Partsupp records, and the number of results records is m_1 times as many as the base case.
- **Datasets♯2 (with fixed output size and increasing input size)**: We fix the output result size while varying the input table sizes by a factor of m_2, where m_2=1, 4, 16, or 64. To achieve this, we copy every record m_2 times so that the input table sizes are m_2 times larger than the base tables. Then in every group of m_2 copied records, we rewrite the key of record i ($i \in [0, m_2 - 1]$) as follows:

  For *Lineitem*:
  $$new\_key = \begin{cases} orig\_key * (2m_2 - 1), & i = 0 \\ orig\_key * (2m_2 - 1) + 2i - 1 & i > 0 \end{cases}$$

  For *Partsupp*:
  $$new\_key = \begin{cases} orig\_key * (2m_2 - 1), & i = 0 \\ orig\_key * (2m_2 - 1) + 2i & i > 0 \end{cases}$$

  In this way, only record 0 in every group has matches. The other records do not contribute to the join output. The total number of result records is kept the same as the base case.
- **Datasets♯3 (with increasing input and output size)**: We vary both the input sizes and the output result size by a factor of m_3, where m_3=1, 4, 16, or 64. To achieve this, we copy every record m_3 times so that the input table sizes are m_3 times larger than the base tables. Then in every group of m_3 copied records, we rewrite the key of record i ($i \in [0, m_3 - 1]$) as follows:

  For *Lineitem*: $new\_key = orig\_key * m_3 + i$

  For *Partsupp*: $new\_key = orig\_key * m_3$

  In this way, record 0 in every group in Lineitem matches all m_3 records in the corresponding group in Partsupp, while the other records in Lineitem do not have any matches. Hence, the total number of result records is m_3 times more than in the base case.

Statistics of the three datasets are shown in Tables 4–6.

Table 3: TPC-H tables: Lineitem and Partsupp (sf=1.0)

Table	#Records	#Unique-Keys	#Matches-Per-Record
Lineitem	6001215	799541	1
Partsupp	800000	800000	[0, 24]; avg. 7.5

Table 4: Statistics of Datasets♯1

Table	m_1	#Inputs	#Unique-Keys	#Matchs-Per-Record	#Outputs
Lineitem	1	6001215	799541	1	6001215
	4		200000	4	24004860
	16		50000	16	96019440
	64		12500	64	384077760
Partsupp	1	800000	800000	[0,24]; avg. 7.5	6001215
	4		200000	[9,57]; avg. 30	24004860
	16		50000	[70,166]; avg. 120	96019440
	64		12500	[394,569]; avg. 480	384077760

Table 5: Statistics of Datasets♯2

Table	m_2	#Inputs	#Unique-Keys	#Matchs-Per-Record	#Outputs
Lineitem	1	6001215	799541	1	6001215
	4	24004860	3198164		
	16	96019440	12792656		
	64	384077760	51170624		
Partsupp	1	800000	800000	[0,24]; avg. 7.5	
	4	3200000	3200000		
	16	12800000	12800000		
	64	51200000	51200000		

Table 6: Statistics of Datasets♯3

Table	m_3	#Inputs	#Unique-Keys	#Matchs-Per-Record	#Outputs
Lineitem	1	6001215	799541	1	6001215
	4	24004860	3198164	4	24004860
	16	96019440	12792656	16	96019440
	64	384077760	51170624	64	384077760
Partsupp	1	800000	800000	[0,24]; avg. 7.5	6001215
	4	3200000			24004860
	16	12800000			96019440
	64	51200000			384077760