Progressive Result Generation for Multi-Criteria Decision Support Queries

Venkatesh Raghavan[1], Elke A. Rundensteiner[2]

Department of Computer Science, Worcester Polytechnic Institute, 100 Institute Road, Worcester MA, USA
{[1]venky,[2]rundenst}@cs.wpi.edu

***Abstract*— Multi-criteria decision support (MCDS) is crucial in many business and web applications such as web searches, B2B portals and on-line commerce. Such MCDS applications need to report results early; as soon as they are being generated so that they can react and formulate competitive decisions in near real-time. The ease in expressing user preferences in web-based applications has made Pareto-optimal (*skyline*) queries a popular class of MCDS queries. However, state-of-the-art techniques either focus on handling skylines on single input sets (i.e., no joins) or do not tackle the challenge of producing progressive early output results. In this work, we propose a progressive query evaluation framework *ProgXe* that transforms the execution of queries involving skyline over joins to be *non-blocking*, i.e., to be progressively generating results early and often. In *ProgXe* the query processing (join, mapping and skyline) is conducted at multiple levels of abstraction, thereby exploiting the knowledge gained from both input as well as mapped output spaces. This knowledge enables us to identify and reason about abstract-level relationships to guarantee correctness of early output. It also provides optimization opportunities previously missed by current techniques. To further optimize *ProgXe*, we incorporate an ordering technique that optimizes the rate at which results are reported by translating the optimization of tuple-level processing into a job-sequencing problem. Our experimental study over a wide variety of data sets demonstrates the superiority of our approach over state-of-the-art techniques.**

I. Introduction

A. Skyline over Disparate Sources

The rapid growth in the number of Internet users has resulted in a variety of on-line services that facilitate commerce, information retrieval and social networking. This phenomenon has highlighted the need for real-time support of complex multi-criteria decision support (MCDS) queries [1]. The intuitive nature of specifying a set of user preferences has made Pareto-optimal (or *skyline*) queries a popular class of MCDS queries [1]–[3]. These MCDS systems need to support *real-time result generation* while processing queries that: (1) access data from disparate sources via joins, and (2) combine several attributes across these sources through possibly complex user-defined mapping functions to characterize the final result. In this work, we target queries which evaluate user preferences over the join, here known as *SkyMapJoin* (SMJ) queries.

Real-time MCDS applications need to process queries with a high degree of responsiveness. Therefore, the query execution strategy must report partial results as early as possible rather than waiting until the end of query processing, commonly known as *progressive result generation* [4], [5]. Also, they must guarantee correctness, i.e., an early reported partial result must be guaranteed to remain in the final result set. Lastly, the query execution strategy must produce the complete result set, i.e., no promising candidates should ever be discarded. We substantiate these requirements by studying a wide variety of applications such as those listed below.

B. Motivating Real-World Applications

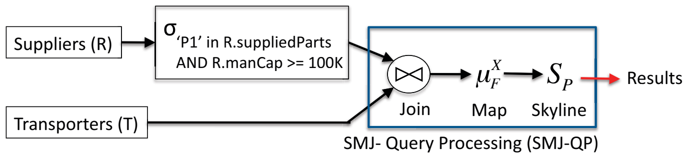
Example 1: Internet Aggregators. The increase in the use and popularity of on-line vendors has resulted in Internet aggregators such as *mySimon.com* for durable goods and *kayak.com* for travel services. Aggregators access and combine data from several sources to produce complex results that are then pruned by the skyline operation. For example, a Kayak-user planning a holiday in Europe visiting both Rome and Paris has different preferences in each leg of the journey. For instance, as Rome is an ancient city with many historic sites, the user is willing to walk twice as much in Rome than in Paris. In addition, the user has a cumulative goal of minimizing the total cost of the trip. Rather than waiting to see 1000's or more matches all at once, a user of such an application may want to have results progressively displayed as soon they are being computed.

Example 2: On-line Search Refinement. The underlying databases of any on-line search application expect precisely defined queries while users may seldom have the exact knowledge [6]. A query against a large database may be long running and could potentially output an empty answer set. In such applications, the results of a slightly reformulated query may satisfy the user's needs equally well. However, careless relaxation of queries can potentially result in large and therefore unusable answer sets. Therefore, one must only return results that are as close as possible to the original query, i.e., a skyline of results [6]. To avoid wasting resources on producing unnecessary relaxations, it is prudent to produce early results as they are generated - thereby providing an opportunity to the user for providing immediate feedback [7].

Example 3: Supply-Chain Management. A manufacturer in a supply chain aims to maximize profit and market share, while minimizing overhead and delays. This is achieved by structuring a production and distribution plan through the evaluation of various alternatives. To illustrate, $Q1$ (as in Figure 1.a) identifies the suppliers that can produce "100K" units of the part "*P1*" and couples them with transporters that deliver it. The preference is to minimize both total cost ($tCost$) and delays ($delay$).

```
Q1: SELECT R.id, T.id,
(R.uPrice + T.uShipCost) as tCost,
(2 * R.manTime + T.shipTime) as delay
FROM Suppliers R, Transporters T
WHERE R.country=T.country AND
'P1' in R.suppliedParts AND R.manCap>=100K
PREFERRING LOWEST(tCost) AND LOWEST(delay)
```

(a)


Suppliers (R)
σ 'P1' in R.suppliedParts AND R.manCap >= 100K
Transporters (T)
Join
μ_F^X Map
S_P Skyline
Results
SMJ- Query Processing (SMJ-QP)

(b)

Fig. 1. Motivating Example: a) Query $Q1$ b) Traditional Query Plan for $Q1$

C. *State-of-the-art Techniques*

The traditional approach [1], [6] is to view skyline processing independent from join evaluation (Figure 1.b). Such a query evaluation is divided into disjoint steps. Thus, the skyline operation has to wait until all join results have been generated and inspected to even begin to generate a skyline result over them. This approach renders the query execution to be *blocking*– making it not viable for *progressive result generation*. Recently, [8], [9] proposed techniques exploiting the principle of skyline *partial push-through* [1], [10] on each individual data source. However, as the number of dimensions increases the pruning capacity of the push-through principle is greatly reduced, sometimes up to the size of the entire source [11]. Since the skyline *partial push-through* is itself blocking, the local pruning employed in [8], [9] may be computationally intensive without yielding a single progressively generated partial result. In addition, [8], [9] are unable to look ahead into the output space to make decisions that can further optimize progressive result generation.

D. *Our Proposed Solution: The ProgXe Framework*

We propose a progressive query execution framework called **ProgXe**. *ProgXe* avoids diving directly into the expensive tuple-level join and skyline processing as in [1], [6], [8], [9]. Instead, we look ahead into the mapped output space to determine interrelations between the input and output spaces to identify progressive result generation opportunities missed by current techniques. The proposed approach facilitates progressive result generation by: (1) efficiently looking ahead into the output space without having to conduct any query processing at the granularity of individual tuples, (2) exploiting the mapped output space knowledge to determine where partial results lie that can be reported early, and (3) ensuring correctness and completeness guarantees by analyzing the dependencies in the output space.

E. *Our Contributions*

- We design a pipelined execution framework *ProgXe* that represents the foundation of our *progressive result generation* approach. *ProgXe* exploits the skyline knowledge at various levels of data abstraction in both the input and output space.
- We propose the *progressive driven ordering* (*ProgOrder*) optimization which employs a cost benefit model to determine the order in which we perform the expensive tuple-level processing; such that the rate at which the partial results can be output early is maximized.
- During the tuple-level processing, to ensure the correct reporting of early results, we present the *progressive result determination* (*ProgDetermine*) technique. *ProgDetermine* enables us to identify the subset of results generated so far which are guaranteed to be in the final skyline and therefore can be output early.
- Our experimental analysis demonstrates the superiority of our proposed techniques over state-of-the-art techniques across a wide variety of data sets.

The rest of the paper is organized as follows: in Section II we review the preference model used in the skyline operation; and the algebra model used to represent queries such as $Q1$. We introduce the main intuition of our *ProgXe* framework in Section III. The core algorithms proposed in this work, namely *progressive driven ordering* and *progressive result determination* are presented in Sections IV and V respectively. Our experimental evaluation is described in Section VI. Section VII reviews related work while Section VIII concludes the paper.

II. PRELIMINARIES

A. *Preference Model: Used in Skyline Operation*

Each d-dimensional object is defined by a set of attributes A = $\{a_1, \ldots, a_d\}$. For a given object r_i, the value of the attribute a_k can be accessed as $r_i[a_k]$. $Dom(a_k)$ denotes the domain of the attribute a_k and $Dom(A) = Dom(a_1) \times \ldots \times Dom(a_d)$. Given a set of attributes $E \subseteq A$, the preference P_i over the set of objects R is defined as $P_i := (E, \succ_P)$ where $\succ_P$ is a *strict partial order* on the domain of E. Given a set of preferences $\{P_1, \ldots, P_m\}$, their combined Pareto preference P is defined as a set of equally important preferences.

Definition 1: For a set of d-dimensional tuples R and preference $P = (E, \succ_P)$ over R, a tuple $r_i \in R$ **dominates** tuple $r_j \in R$ based on the preference P (denoted as $r_i \succ_P r_j$), iff $(\forall (a_k \in E)\ (r_i[a_k] \succeq r_j[a_k]) \wedge \exists (a_l \in E)\ (r_i[a_l] \succ r_j[a_l]))$.

B. *Query Model Containing Map and Skyline Operation*

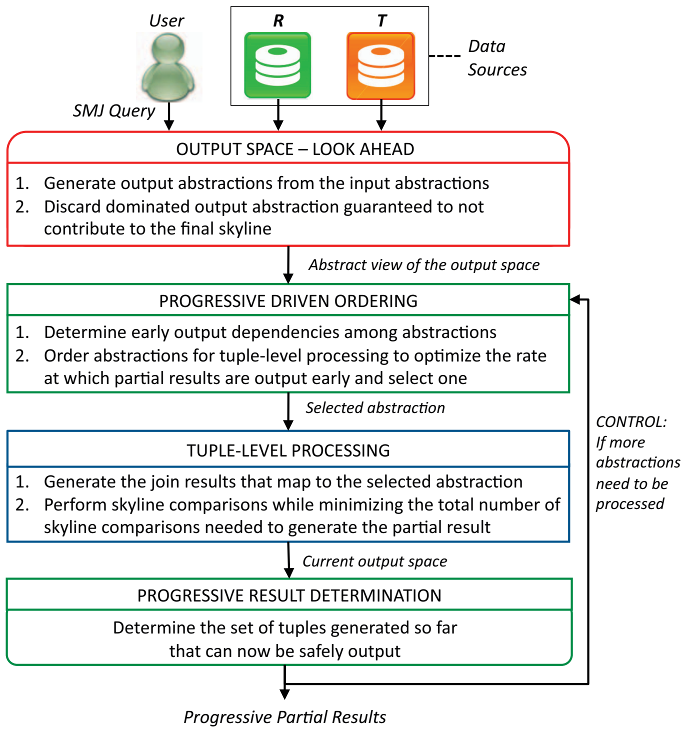
For each input tuple r_i the mapping function f_j in the set of k mapping function $\mathcal{F}$=$\{f_1, \ldots f_k\}$, takes as input a set of attributes $B_j \subseteq A$ and returns a value x. That is, $f_j : Dom(B_j) \rightarrow Dom(x)$.

Map ($\mu_{[\mathcal{F},X]}$) operator applies a set of k mapping functions $\mathcal{F}$ to transform each d-dimensional object $r_i \in R$ into a k-dimensional output object r'_i defined by a set of attributes $X = \{x_1, x_2, \ldots, x_k\}$, with x_j generated by the function $f_j \in \mathcal{F}$.

Skyline ($\mathcal{S}_P$). Given a set of tuples R and a preference P, $\mathcal{S}_P(R)$ returns the subset of all non-dominated objects in R.

III. Progressive Query Execution Framework

In this section, we provide an overview of the main steps of our proposed progressive query execution framework, *ProgXe*. In this work, we assume the input data sets are *partitioned* into a multi-dimensional grid structure. Other space-partitioning methodologies such as quad-tree and R-tree structures can also be utilized. The principles proposed in this work are still applicable with some modifications.



User
R
T
Data Sources
SMJ Query
OUTPUT SPACE – LOOK AHEAD
1. Generate output abstractions from the input abstractions
2. Discard dominated output abstraction guaranteed to not contribute to the final skyline
Abstract view of the output space
PROGRESSIVE DRIVEN ORDERING
1. Determine early output dependencies among abstractions
2. Order abstractions for tuple-level processing to optimize the rate at which partial results are output early and select one
Selected abstraction
TUPLE-LEVEL PROCESSING
1. Generate the join results that map to the selected abstraction
2. Perform skyline comparisons while minimizing the total number of skyline comparisons needed to generate the partial result
Current output space
PROGRESSIVE RESULT DETERMINATION
Determine the set of tuples generated so far that can now be safely output
CONTROL: If more abstractions need to be processed
Progressive Partial Results

Fig. 2. Overview of the Progressive Query Execution Framework

The *ProgXe* framework (Figure 2) can efficiently exploit the skyline knowledge at various steps of query processing (both in evaluation and early output) as well as at different levels of data abstraction. Without directly having to dive into expensive tuple-level processing like existing state-of-the-art techniques [6], [8], [9], we look ahead into the output space in our first step, thus named **output space look-ahead**. The goals of this step are to: (1) generate the higher-level abstraction of the output space and (2) prune dominated abstractions early on.

Next, in our **progressive driven ordering** step we investigate the output space to identify output abstractions that have a higher likelihood of generating tuples that can be output early. The goal of this step is to maximize the rate at which the results are output early. This is achieved by ordering the sequence in which the output abstractions are considered for the expensive operation of tuple-level processing. The chosen output abstraction is then sent for **tuple-level processing**. In this step we (1) generate the join results that map to the output abstraction, (2) map the join results by user defined mapping functions, and (3) minimize the number of dominance comparisons needed to generate the intermediate tuples.

From the generated intermediate results, we need to next determine the subset of these results that safely belong into the final skyline so that they can be output early. In our **progressive result determination** step, we analyze the dependencies in the output space to determine which tuples can be output early since they are guaranteed to belong into the final result set. The last three phases of the *ProgXe* pipelined steps are repeated until all output abstractions have either been considered for tuple-level processing or are dominated (and thus guaranteed to not contribute to the final result).

Next, we present a brief overview of the *output space look-ahead* and *tuple-level processing* in Sections III-A and III-B respectively, while the full details can be found in our technical report [12]. In Section IV we elaborate on the progressive benefits of ordering, followed by our *progressive driven ordering* algorithm to achieve this goal. Lastly, in Section V we present the details of our *progressive result determination* phase. The notations used in this work are summarized in Table I.

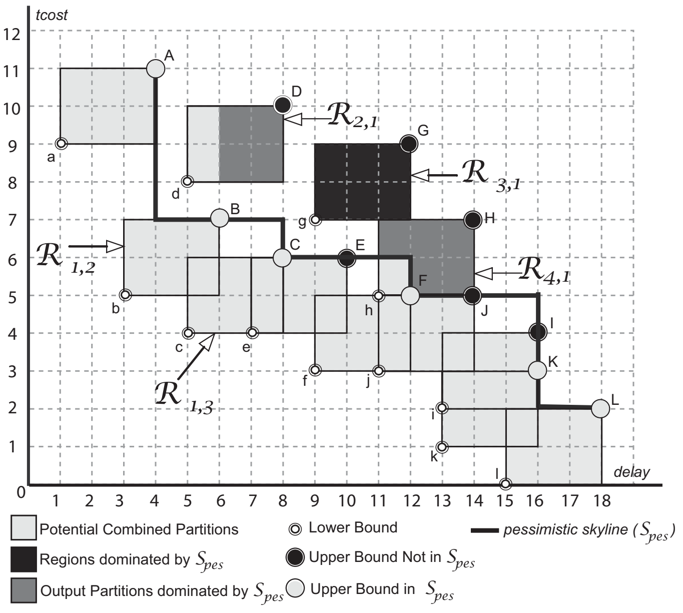


Fig. 3. Output Space Look-Ahead: Avoid Join and/or Skyline Costs

A. *Output Space Look-Ahead*

The aim of this step is to perform the join and skyline query execution at a higher granularity of abstraction rather than at the level of individual tuples. More specifically, for a pair of input partitions, one from each table $I_a^R \in R$ and $I_b^T \in T$, we first determine if the set of tuples in these partitions are guaranteed to produce at least one join result. To avoid tuple-level comparison, we maintain for each partition the signature of the list of join domain values of the tuples contained in the partition. These signatures can be efficiently maintained by either Bloom Filter or a bit vector. Partitions that do not share even a single join domain value are guaranteed to not generate any join result. Therefore such input partition pairs

TABLE I
NOTATIONS USED IN THIS WORK

Notation	Meaning
I_i^R	Input partition in R
$\mathcal{I}^R$	Set of all input partitions in R
$r_f t_g$	Join result, $r_f \in R$; $t_g \in T$
$\mathcal{R}_{i,j}$	Region in the output space to which the join results from the input partitions $[I_i^R, I_j^T]$ are mapped to
$\mathfrak{R}$	Set of all regions in the output space
LOWER(X)	*Lower-bound* point of a region or partition
MARK(O_i)	Mark partition O_i as "*non-contributing*"
IS_MARKED(O_i)	*True*, if partition O_i is marked, else *false*.

are no longer considered for further processing. Second, we apply the mapping functions to determine the *region* of the output space into which the generated join results would fall (denoted as $\mathcal{R}_{a,b}$) if they were to be generated.

Example 1: Consider an input partition from Supplier(R), I_1^R with bounds $[(0,4)(1,5)]$ and the partition $I_2^T[(3,1)(4,2)]$ from Transporter (T), that share at least one join value. By applying the mapping functions in $Q1$, we can determine that during tuple-level processing the tuples in the above partitions when joined with each other would generate join result(s) that would fall into the region bounded by the lower-bound point $b(3,5)$ and the upper-bound point $B(6,7)$. In Figure 3 we denote this region as $\mathcal{R}_{1,2}$.

For a given set of regions that are guaranteed to be populated during tuple-level processing, we apply domination-based reasoning to prune dominated regions as they are guaranteed to not contribute to the final skyline. This avoids the join evaluation and any subsequent dominance comparison costs altogether for such dominated regions.

Example 2: In Figure 3, UPPER($\mathcal{R}_{1,3}$) $\succ$ LOWER($\mathcal{R}_{3,1}$). Since $\mathcal{R}_{1,3}$ is guaranteed to be populated during tuple-level execution there exists at least one intermediate result $r_f t_g \in \mathcal{R}_{1,3}$ that dominates the intermediate results that map to $\mathcal{R}_{3,1}$. Thus, $\mathcal{R}_{3,1}$ is guaranteed to never contribute to the final result.

To further reduce the number of skyline comparisons, we partition the output space such that each region is composed of a set of output partitions. Next, we identify output partitions that are dominated by other output regions. This allows us to discard all intermediate join results that map to such dominated partitions as they will not contribute to the final result.

Example 3: In Figure 3 the output region $\mathcal{R}_{1,2}$ is partially dominated. That is, its output partitions: O[(6,8) (7,9)], O[(7,8) (8,9)], O[(6,9) (7,10)] and O[(7,9) (8,10)] are dominated by the upper-bound point of output region $\mathcal{R}_{1,2}$ with upper bound $B(6,7)$. Since $\mathcal{R}_{1,2}$ is guaranteed to be populated we can mark the dominated partitions as "non-contributing".

B. Tuple-Level Processing

Next, in the **tuple-level processing** phase we perform the expensive join, map and skyline operations at the granularity of individual tuples. The optimization goals in this phase are to: (1) reduce the total number of domination comparisons, and (2) identify intermediate results that can be output early. We achieve this by piggy backing on the knowledge about the output space gained from the previous steps.

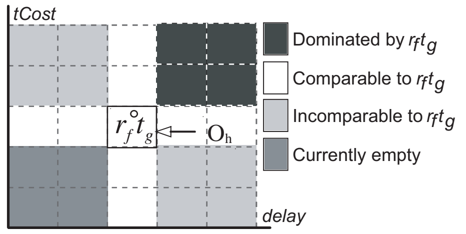


Fig. 4. Tuple-Level Processing: Avoid Skyline Comparisons

For the output region $\mathcal{R}_{a,b}$ chosen for *tuple-level processing*, we first evaluate the join conditions between the tuples in I_a^R and those in I_b^T. Join results are then mapped to its output partition by applying the mapping functions given in the query. Intermediate results that map to dominated output partitions are discarded without performing any dominance comparisons. Next, for each intermediate result $r_f t_g$ that does not map to a dominated output partition, we reduce the dominance comparisons to only a small set of output partitions containing tuples that can potentially dominate it. In Figure 4 we observe the following:

1) Results that map to partitions in the top-left corner and the bottom right corner of O_h cannot dominate $r_f t_g$ and vice versa. Thus, such comparisons can be avoided.
2) Partitions in the bottom-left corner of O_h are currently empty, else O_h would be marked as being dominated.
3) Intermediate results $r_f t_g \in O_h$ can be only dominated by result that map to the slice of partitions that either have the same $tCost$ or $delay$ attribute value as O_h.

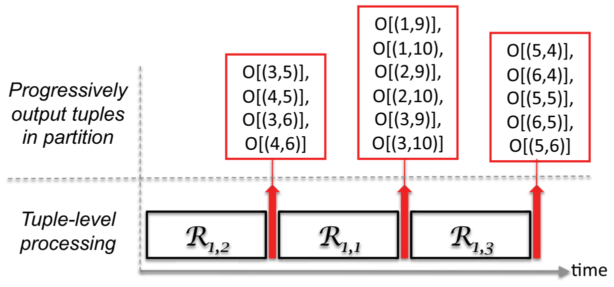
The optimization benefits (i.e., the reduction in the number of skyline comparisons) achieved by our *tuple-level processing* can be quantified as follows. Let us assume that each dimension in the output space is partitioned into k partitions. That is, the d-dimensional grid structure has a total of k^d output partitions. For any skyline algorithm in the worst case scenario all tuples are in the final skyline. Therefore, a naïve approach in the worst case scenario would have to compare against tuples in all k^d partitions. Instead, for each newly generated tuple $r_f t_g \in O_h$ in the worst case we only perform dominance comparisons against tuples that are mapped to a much smaller set of $[k^d - (k-1)^d]$ partitions.

IV. PROGRESSIVE DRIVEN ORDERING

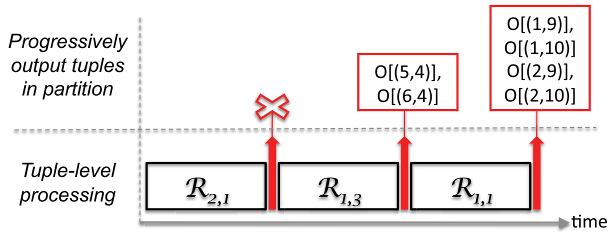
In this section, we highlight the impact that ordering of tuple-level processing can have on progressive result generation. We then propose our technique to optimize the query execution strategy such that rate at which results are output early is maximized. In particular, our solution orders the regions based on their respective progressiveness capacity versus penalty (that is, their respective processing costs required to gain that benefit).

A. *Effects of Ordering*

The order in which we conduct the *tuple-level processing* of regions can affect the rate at which the partial results are output. To elaborate, let us compare two orderings of regions for tuple-level processing. Consider a good ordering that produces more results early: $\mathcal{R}_{1,2}$, $\mathcal{R}_{1,1}$, $\mathcal{R}_{1,3}$, and so on, as depicted in Figure 5.a. Following this ordering, the join results that map to the region $\mathcal{R}_{1,2}$ are generated and then their corresponding dominance comparisons are performed. While examining the output space, as shown in Figure 3, we observe that results that map to the partitions O[(3,5)], O[(3,6)], O[(4,5)], and O[(4,6)] cannot be dominated by any future generated tuples belonging to other regions. Therefore, tuples that map to these partitions (4 of 6 partitions in $\mathcal{R}_{1,2}$) can be safely output early. However, results that map to the remaining two partitions O[(5,5)] and O[(5,6)] cannot yet be output. They can potentially still be dominated by future generated tuples that map to the partitions O[(5,4)] and O[(5,5)] during the tuple-level processing of region $\mathcal{R}_{1,3}$. Thus, they must be held in the output buffer.


Progressively output tuples in partition
Tuple-level processing
O[(3,5)], O[(4,5)], O[(3,6)], O[(4,6)]
O[(1,9)], O[(1,10), O[(2,9)], O[(2,10), O[(3,9)], O[(3,10)]
O[(5,4)], O[(6,4)], O[(5,5)], O[(6,5)], O[(5,6)]
$\mathcal{R}_{1,2}$
$\mathcal{R}_{1,1}$
$\mathcal{R}_{1,3}$
time

(a) More Results Output Earlier



(b) Less Results Output Earlier

Fig. 5. Effects of Ordering on Progressiveness

Next, $\mathcal{R}_{1,1}$ is considered for tuple-level processing. At its completion, we safely return tuples that map to all of $\mathcal{R}_{1,1}$'s partitions. At the end of processing the third region $\mathcal{R}_{1,3}$ we would have reported results from 15 output partitions. In contrast, consider the ordering in Figure 5.b where at the end of processing three regions we could only report results that map to 6 partitions. Therefore, orderings such as in Figure 5.a are clearly preferable over those in Figure 5.b.

To effectively support progressive result generation we propose a **progressive driven ordering** technique that is able to identify abstractions that can produce the largest number of results early using the least amount of CPU time spent on tuple-level processing. Our proposed approach translates the problem into a graph-based job sequencing problem.

B. *Benefit Model: Progressiveness Capacity of a Region*

We define *progressiveness capacity* of an output region $\mathcal{R}_{a,b}$ as the number of skyline results in $\mathcal{R}_{a,b}$ that can be estimated to be output early if tuple-level processing was conducted on $\mathcal{R}_{a,b}$. To estimate the progressiveness capacity of each output region we first determine the maximum number of partial results it can produce. Second and more importantly, we identify the relationship between any two regions and the impact of this relationship on the ability to safely release these results at this point of time into the output.

First, we estimate the maximum number of tuples that an output region could output early. In the context of computational geometry, [13], [14] addressed the problem of estimating the average number of maxima in a set of vectors to be $\Theta((ln(n))^{d-1}/(d-1)!)$, where d is the number of dimensions and n is the cardinality of the input data set. Given I_a^R and I_b^T as the corresponding input partitions of the output region $\mathcal{R}_{a,b}$, we estimate the maximum number of skyline results that each region can produce as follows:

$$Cardinality(\mathcal{R}_{a,b}) = ln(\sigma \cdot n_a^R \cdot n_b^T)^{d-1}/(d-1)! \qquad (1)$$

where n_a^R and n_b^T are the number of tuples in the input partition I_a^R and I_b^T respectively.

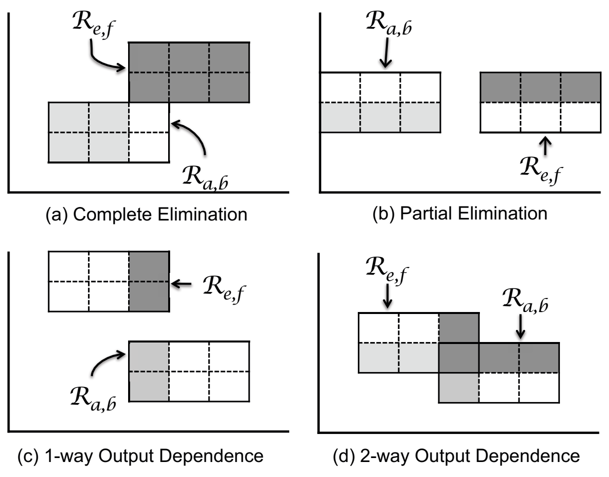


Fig. 6. Relationships between Output Space Abstractions

Next, we examine interrelationships which exist in the output space that can prevent the tuples in $\mathcal{R}_{a,b}$ to be output after the tuple-level processing of region $\mathcal{R}_{a,b}$. Three types of relationships are depicted in Figure 6. First, in Figure 6.a the lightly shaded partitions in the region $\mathcal{R}_{a,b}$ can completely eliminate *all* partitions in the region $\mathcal{R}_{e,f}$. Such a relationship is called **complete elimination**. Second, in Figure 6.b the lightly shaded partitions of $\mathcal{R}_{a,b}$ only dominated a subset of the partitions (darkly shaded) in $\mathcal{R}_{e,f}$. In other words, $\mathcal{R}_{a,b}$ **partially eliminates** $\mathcal{R}_{e,f}$. Lastly, in Figures 6.c and 6.d the tuple-level processing of any one region does not eliminate any part of the other region. However, way future generated tuples that map to the lightly shaded partitions can "potentially" dominate tuples in the darkly shaded partitions of the other regions. Therefore, in Figure 6.c to safely output tuples mapped to the region $\mathcal{R}_{e,f}$ we must wait for $\mathcal{R}_{a,b}$ to finish its tuple-level processing as they *potentially* still can become invalid and then must be discarded. Such a relationship is called **output dependence**. For a pair of regions the *output dependencies* can either be uni- or bi- directional as in Figures 6.c and 6.d respectively.

Next, we introduce a graph representation to capture these relationships and the methodology to determine the *progressiveness capacity* of a region given its dependencies.

Elimination Graph (EL-Graph). A directed graph, denoted as EL-Graph $(\mathfrak{R}, E, W)$: (1) $\mathfrak{R}$ is the set of vertexes, where each vertex represents an output region; (2) E is a set of directed edges between the regions, where an edge exists between the regions $\mathcal{R}_{a,b}$ and $\mathcal{R}_{e,f}$ if and only if there exists an output partition $O_h \in \mathcal{R}_{a,b}$ such that it either partially or completely dominates $\mathcal{R}_{e,f}$.

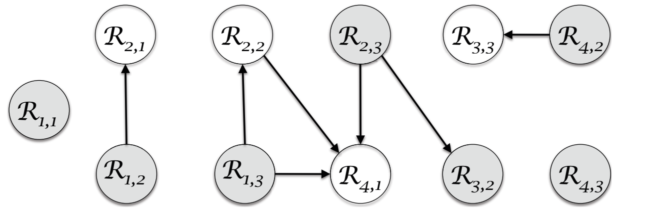


Fig. 7. Elimination Graph (EL-Graph) (Root noded depicted as shaded nodes)

Example 4: In our running example, as depicted in Figure 3, the output region $\mathcal{R}_{1,3}$ has partitions ($O[(5,4)]$ $O[(6,4)]$ and $O[(7,4)]$) if populated during tuple level processing can completely dominate the output region $\mathcal{R}_{1,3}[(4,1)]$. In addition, the above mentioned partitions can dominate a subset of partitions in the region $\mathcal{R}_{2,2}[(7,4)]$. In other words, $\mathcal{R}_{1,3}$ completely dominates $\mathcal{R}_{4,1}$ and partially dominates $\mathcal{R}_{2,2}$. Thus in our elimination graph, as shown in Figure 7, we have a directed edge from node $\mathcal{R}_{1,3}$ to $\mathcal{R}_{4,1}$ and $\mathcal{R}_{2,2}$.

The precise distribution of final tuples in each region will only be determined after all tuples within that region have been joined, mapped and dominance comparison have been completed. The roots of the elimination graph (depicted as shaded circles in Figure 7) represent regions whose processing can neither be completely nor partially eliminated by other regions and therefore have a higher probability of reporting results early. We further investigate dependencies among these root nodes to determine the next output region with the highest expected benefit. The chosen region is then sent for the expensive tuple-level processing. As each output region is considered for tuple-level processing, other non-root regions can become root nodes, making them potential candidates for execution. In our proposed approach we incrementally maintain the elimination graph.

Progressiveness capacity an output region $\mathcal{R}_{a,b}$ is defined as the percentage of its estimated cardinality (from Equation 1) that can be safely output early at a given instance. The main intuition is to identify all output partitions in a region that solely depend on the tuple-level processing of itself to be able to output early. To illustrate in Figure 8 for the region $\mathcal{R}_{1,2}$, the tuples in partition $O_h[(3,5)(4,6)]$ can be output at the end of tuple-level processing of $\mathcal{R}_{1,2}$.

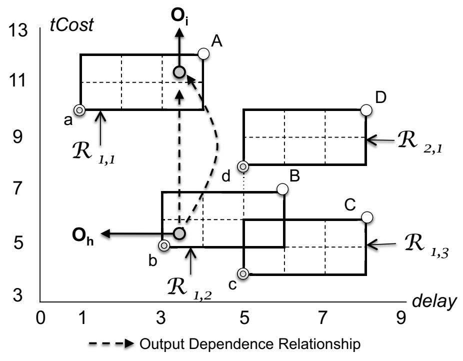


Fig. 8. Calculating *progressiveness capacity*

Definition 2: The progressive partition count ($ProgCount$) for an output region $\mathcal{R}_{a,b}$ is defined as the number of partitions in $\mathcal{R}_{a,b}$ that can neither be eliminated nor have output dependencies to partitions belonging to other output regions.

Example 5: In Figure 8, the progressiveness count for $\mathcal{R}_{1,2}$ and $\mathcal{R}_{1,1}$ are both 4 while $ProgCount(\mathcal{R}_{1,3})$=3. Since all partitions of the region $\mathcal{R}_{2,1}$ are either dependent or eliminated by partitions belonging to other regions, $ProgCount(\mathcal{R}_{2,1})$=0.

The benefit of processing $\mathcal{R}_{a,b}$ is now defined to be the product of its cardinality and the percentage of partitions that are guaranteed to be output immediately at the end of its tuple-level processing.

$$Benefit(\mathcal{R}_{a,b}) = \frac{ProgCount(\mathcal{R}_{a,b})}{PartitionCount(\mathcal{R}_{a,b})} \cdot Cardinality(\mathcal{R}_{a,b}) \tag{2}$$

where $PartitionCount(\mathcal{R}_{a,b})$ is the total number of output partitions in the region $\mathcal{R}_{a,b}$.

C. Cost Model: Tuple-Level Processing of Output Region

The time required, here considered a penalty, to complete the tuple-level processing of the region $\mathcal{R}_{a,b}$ includes: (1) the cost for materializing the join results (C_{join}), (2) cost of mapping (C_{map}), and (3) cost of skyline comparisons (C_{sky}).

$$Cost(\mathcal{R}_{a,b}) = C_{join}(\mathcal{R}_{a,b}) + C_{map}(\mathcal{R}_{a,b}) + C_{sky}(\mathcal{R}_{a,b}) \tag{3}$$

For an output region $\mathcal{R}_{a,b}$ we estimate the cost of join as the product of the cardinalities of their respective input partitions.

$$C_{join} = (|I_a^R| \cdot |I_b^T|) \tag{4}$$

The cardinality of the join results is $\sigma \cdot |I_a^R| \cdot |I_b^T|$, where σ is the join selectivity between two input sources. If the amortized cost to map each of the join results is $O(1)$, then:

$$C_{map} = (\sigma \cdot |I_a^R| \cdot |I_b^T|) \tag{5}$$

Assuming independent data distribution, the average skyline execution time by Kung et. al. [2] is $\mathcal{O}(|S| \cdot log^{\alpha}|S|)$, where

$|S|$ is the number of tuples to be compared against and $\alpha = 1$ for $d = 2$ or 3, and $\alpha = d-2$ for $d \geq 4$. In Section III-B we concluded that for each newly generated join result we need to perform dominance comparison with tuples in at most $(k \cdot d)$ partitions. Then for each newly generated tuple $r_f t_g \in O_h$ we need to conduct $(CP_{avg} \cdot s_{avg})$ comparisons, where CP_{avg} is the average number of **c**omparable **p**artitions for a tuple mapped an output partition (see Section III-B) and s_{avg} is the average number of tuples in each output partition. Thus, the amortized time for evaluating the dominance comparison for a single intermediate result $r_f t_g \in O_h$ is:

$$O\left(\left(CP_{avg} \cdot s_{avg}\right) \cdot log^{\alpha}\left(CP_{avg} \cdot s_{avg}\right)\right) \qquad (6)$$

where $\alpha = 1$ for $d = 2$ or 3, and $\alpha = d-2$ for $d \geq 4$.

Let us denote $|I_a^R|$ as n_a^R and $|I_b^T|$ as n_b^T. By substituting the respective terms in Equation 3 by those in Equations 4, 5 and 6, the amortized time for processing the output region $\mathcal{R}_{a,b}$ now is modelled as:

$$O\Bigg(\left(n_a^R \cdot n_b^T\right) + \left(\sigma \cdot n_a^R \cdot n_b^T\right) + (\sigma \cdot n_a^R \cdot n_b^T)\left(\left(CP_{avg} \cdot s_{avg}\right) \cdot log^{\alpha}\left(CP_{avg} \cdot s_{avg}\right)\right)\Bigg) \qquad (7)$$

where $\alpha = 1$ for $d = 2$ or 3, and $\alpha = d-2$ for $d \geq 4$.

D. The $ProgOrder$ Algorithm: Putting it all together

We now propose the *progressive-ordering* (*ProgOrder*) algorithm that iteratively determines the order in which these regions are considered for *tuple-level processing*. *ProgOrder* ranks each output region that is a root in the *elimination graph*, where $rank(\mathcal{R}_{a,b})$ is derived from Equations 2 and 3 as:

$$rank(\mathcal{R}_{a,b}) = \frac{Benefit(\mathcal{R}_{a,b})}{Cost(\mathcal{R}_{a,b})} \qquad (8)$$

The list of all such root regions is maintained in an inverted priority queue. We pick the next region to be considered for tuple-level processing from the top of this queue. After the tuple-level processing of the chosen region is completed, the graph and benefit models are incrementally updated in order to accurately chose the next region. This process is repeated until all regions in the mapped output space have either been considered for tuple-level processing or have been dominated by newly generated tuple(s).

The step-by-step description of *ProgOrder* is listed in Algorithm 1. The algorithm maintains a list of current root nodes in a priority queue $PQueue$ (Line: 3-5). In $PQueue$ the regions are ranked in descending order of their respective *rank*. In each iteration (Line: 6–19), we pick the next output region from the top of the $PQueue$. The chosen region, say $\mathcal{R}_{a,b}$, is then sent for tuple-level processing (Line: 8). Thereafter, we update the benefit model for all regions affected by the execution of $\mathcal{R}_{a,b}$ (Line: 13). Next, we identify all regions that have "newly" become root nodes in EL-Graph after removing $\mathcal{R}_{a,b}$ from the graph (Line: 15). For all such regions, we calculate its *rank* (Equation 8) and insert them into $PQueue$ (Line: 18). The above steps are iterated until all non-dominated regions have been considered for tuple-level processing.

Algorithm 1 *ProgOrder*

```
Input: $\mathfrak{R}$ {Region Collection}; input partitions ($\mathcal{I}^R$, $\mathcal{I}^T$)
Output: 0 to denote a successful execution; else return 1; {Iteratively
    pick the a region for tuple-level processing}
 1: Build the initial elimination graph, $EL$-Graph.
 2: Initialize $EL_{root}$ to the list of all root nodes in $EL$-Graph.
 3: for each $\mathcal{R}_{a,b}$ in $EL_{root}$: do
 4:     analyse-Cost-vs-Benefit($\mathcal{R}_{a,b}$)
 5:     Add $\mathcal{R}_{a,b}$ to the inverted priority queue $PQueue$ {$PQueue$
        sorted by the scoring function $rank(\mathcal{R}_{a,b})$}
 6: while $|\mathfrak{R}| \neq \phi$ do
 7:     $\mathcal{R}_{a,b} \leftarrow$ remove($PQueue$) {Removes top of the list}
 8:     Perform tuple-level processing for region $\mathcal{R}_{a,b}$
 9:     Discard those regions now dominated by the newly generated
        tuple(s) in $\mathcal{R}_{a,b}$ using EL-Graph.
10:     for each edge $e = \overrightarrow{\mathcal{R}_{a,b}, \mathcal{R}_{e,f}} \in EL$-Graph do
11:         Remove $e$
12:         if $\mathcal{R}_{e,f} \in PQueue$ then
13:             Update its benefit, $Benefit(\mathcal{R}_{e,f})$.
14:         else
15:             $EL_{new-root} \leftarrow$ nodes $EL$-Graph that became a root
                due to the removal of edge e
16:             for each $\mathcal{R}_{i,j} \in EL_{new-root}$ do
17:                 analyse-Cost-vs-Benefit($\mathcal{R}_{i,j}$)
18:                 Add $\mathcal{R}_{i,j}$ to $PQueue$
19:         $EL_{root} \leftarrow EL_{root} \cup EL_{new-root}$
20:     Remove $\mathcal{R}_{a,b}$ from $\mathfrak{R}$
21: return 0;
22: procedure analyse-Cost-vs-Benefit($\mathcal{R}_{a,b}$)
23: Compute the progressiveness count $ProgCount(\mathcal{R}_{a,b})$ and the
    benefit, $Benefit((\mathcal{R}_{a,b})$.
24: Calculate the cost to process the region, $COST(\mathcal{R}_{a,b}))$
```

Time Complexity. Let $n = |\mathfrak{R}|$ be the total number of regions in the mapped output space. For our algorithm, in the worst case scenarios all regions overlap each other. Then, after the first iteration (first pick), we need to touch $n-1$ regions, in the second iteration $n-2$ regions and so on. Therefore, the time complexity for updating the benefit model is $\mathcal{O}(n^2)$. The time complexity to build $PQueue$ is $\mathcal{O}(n \cdot log(n))$. Therefore, *ProgOrder* has a worst case time complexity of $\mathcal{O}(n^2)$. However, in another extreme scenario when there is no relationship between any two region, the time complexity is reduced to the cost of maintaining the priority queue which is $\mathcal{O}(n \cdot log(n))$. Since typically, $n << N$ (number of tuples in one source), $\mathcal{O}(n^2) << \mathcal{O}(N^2)$.

V. Progressive Result Determination

At the completion of the tuple-level processing of region $\mathcal{R}_{a,b}$, not all of its tuples can be output since they could potentially be discarded by tuples generated by other regions (as in Figures 6.c and 6.d). Therefore, we need a strategy to determine which subset of the tuples generated so far can be output early as partial results. Our *progressive result reporting* technique assures the results output early are correct. To ensure correctness, our technique avoids reporting: (1) *false positives*: partial results that were reported early even though they are found to eventually not belong into the final skyline result, and (2) *false negative(s)*: not report or worst yet drop results that eventually should have been in the final query result set.

To investigate whether a tuple generated by the tuple-level processing can be output is a potentially blocking operation since it requires the knowledge about the output space. To support *progressive result reporting*, we first translate this problem to decision making at the coarser granularity of output partitions. More precisely, we translate the problem into the process of determining tuples that map to a partition O_h belong in the final result set and therefore can be safely output early on. The intuition behind our approach is to guarantee that any partition (say O_l) that may contain results that can dominate those in O_h, have all been generated and their skyline computations completed. That is, no future tuples will map to O_l.

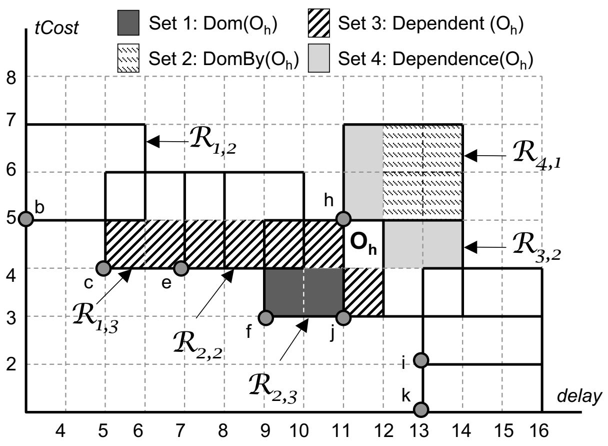


Fig. 9. Data Maintained for O_h[(11,4)(12,5)]

To illustrate the intuition consider the running example in Figure 9 for the output partition $O_h[(11, 4)]$. To safely output the tuples contained in O_h, we must first ensure that no future tuples will map to the partition O_h. In other words, the tuple-level processing for all output regions to which O_h contributes has been completed. Second, we must ensure that the output partition in Set 3 of Figure 9 will be empty in the final output space. Lastly, we must also ensure that all tuples that map to the output partitions in Set 1 of Figure 9 have already been generated. When these conditions are met we can safely output the generated tuples that map to O_h since none of the yet to be generated tuples could ever dominate those in O_h. We translate this core intuition into our *correctness principle*.

Principle *1 (Correctness Principle):* The output partition O_h containing result tuples can be safely output provided it satisfies the following conditions:

1) Tuples that map to O_h have all been generated and their corresponding skyline comparisons performed.
2) For all partitions O_l such that $UPPER(O_l) \succ LOWER(O_h)$, O_l is guaranteed to be empty in the final output space.
3) For all partitions O_k such that $UPPER(O_k) \not\succ LOWER(O_h)$, it is guaranteed that no future generated tuple $r_x t_y$ will map into O_l that may satisfy the condition $r_x t_y \succ r_f t_g$ where $r_f t_g \in O_h$.

To check for the first condition, for each partition O_h, we maintain the count of all output regions (say $\mathcal{R}_{a,b}$) that contain O_h. **Region count** for a partition is denoted as $RegCount(O_h)$. We decrement $RegCount(O_h)$ after the tuple-level processing of $\mathcal{R}_{a,b}$. To evaluate the second condition in Principle 1, we maintain a list of partitions that are guaranteed to dominate O_h if they were populated. Set 1 in Figure 9 is called the **dominate list**, denoted by $Dom(O_h)$. Inversely, the list of partitions that are guaranteed to be dominated by O_h if $|O_h| \neq \phi$, called the **dominated-by list** (Set 2 in Figure 9). It is denoted as $DomBy(O_h)$.

To evaluate the third condition in Principle 1, we maintain for each output partition O_h a list of partitions (say O_k) which satisfies the conditions: (1) $UPPER(O_k) \not\succ LOWER(O_h)$, and (2) there potentially could be tuples in O_k that can dominate those in O_h. For example, Set 3 in Figure 9 can potentially have tuples that can dominate those in O_h. This list is called **dependent list** and denoted as $Dependent(O_h)$. Conversely, we also maintain a list called output **dependence list** denoted as $Dependence(O_h)$ (Set 4 in Figure 9).

Algorithm 2 *ProgDetermine*

```
Input: $\mathcal{R}_{a,b}$ {Region whose tuple-level processing has just been
    completed}; $\mathfrak{R}$ {Region Collection};
Output: Set of output partitions that can be output early.
 1: $Output = \phi$
 2: for each partition $O_h \in \mathcal{R}_{a,b}$ do
 3:     Decrement $RegCount(O_h)$;
 4:     if $RegCount(O_h) = 0$ then
 5:         Call Progressive-Maintenance($O_h$);
 6:         Call Progressive-Output($O_h$);
 7: return $Output$
 8: procedure Progressive-Maintenance($O_h$)
 9:     for each partition $O_g \in DomBy(O_h)$
10:         Remove $O_h$ from $Dom(O_g)$
11:         Call Progressive-Output($O_g$);
12:     for each partition $O_g \in Dependent(O_h)$
13:         Remove $O_h$ from $Dependence(O_g)$
14:         Call Progressive-Output($O_g$);
15: end procedure
16: procedure Progressive-Output($O_h$)
17:     if $|Dom(O_h)| = 0) \wedge \neg IS\_MARKED(O_h)$
18:         if $|Dependence(O_h)| = 0$ then Add $O_h$ to $Output$
19: end procedure
```

A. The ProgDetermine Technique: Putting it all together

Next, we present the technique (see Algorithm 2) that utilizes the above mentioned lists to determine a set of partitions that can be output early yet safely based on Principle 1. First, we assume that Algorithm 2 is triggered after the tuple-level processing of each output region $\mathcal{R}_{a,b}$. For each output partition $O_h \in \mathcal{R}_{a,b}$, we first decrement the *partition region count* (Line: 3). If partition O_h is guaranteed to have no future tuples mapped to it (i.e., $RegCount(O_h)$= 0) we trigger the progressive maintenance that updates the corresponding lists associated with O_h. For example, in Line: 12-13 assuming that no future tuples will fall in O_h, for each partition $O_g \in Dependent(O_h)$ we can now safely remove O_h from their corresponding *dependence* list. While updating these lists, we investigate if the partitions that are affected by O_h can themselves be output (Line: 14). Finally, in Line: 6 we investigate whether O_h can itself be output early. If $|Dom(O_h)| = 0$ we can guarantee that tuples that map to partitions that dominate O_h have already been generated. To verify condition (2) of Principle 1 we check if its *dependence list* is empty, namely $|Dependence(O_h)| = \phi$. To avoid having to add and remove partitions from each of the lists, we instead utilize a count-based realization. That is, we maintain a dedicated count for each list. Now, instead of removing a partition from a list (as in Line: 10, 13), we merely decrement the corresponding count.

Time Complexity. Similar to our *ProgOrder*, in the worst case scenarios all regions overlap each other and therefore the time complexity of *ProgDetermine* is $\mathcal{O}(n)$, needed to update all counts. However, in the best case it is $\mathcal{O}(1)$.

VI. Performance Study

A. Experimental Setup

Alternative Techniques. State-of-the-art techniques that handle skylines over joins are: first, *JF-SL* using a hash-based join [6]. Second, an optimized *JF-SL*$^+$ which uses the principle of skyline partial push-through to prune each data source. Third, *SAJ* [6] extended the popular Fagin technique [15] following the JF-SL paradigm. Recently [8] proposed the *Skyline-Sort-Merge-Join* (*SSMJ*) technique to handle skylines over joins. SSMJ maintains for each data source two active lists of objects: (1) those objects that are in the source-level skyline generated by ignoring the join condition (denoted as LS(S)), and (2) the objects that are in group-level skyline for each join attribute value (denoted as LS(N)). Next, the approach performs the query evaluation across the different data sources. First, the tuples in the source-level sets are joined with those in the source-level sets of the other source i.e., $LS(S) \bowtie LS(S)$. SSMJ can only output the first batch of partial results once all the join results have been generated, mapped and skyline comparisons conducted. Next, *SSMJ* performs query evaluation to generate tuples generated from $LS(S) \bowtie LS(N)$, $LS(N) \bowtie LS(S)$ and $LS(N) \bowtie LS(N)$ and the results reported at the end of query evaluation. In short, *SSMJ* produces results at two distinct moments of time in batches. Since this method cannot exploit the knowledge of the output space, SSMJ cannot support the early output of the results as generated like in our proposed techniques. [9] noted that for low join selectivity of ≤ 0.000001, *SSMJ* is ineffective in pruning many objects both at the source- and group-level of each data source.

Experimental Platform. All algorithms were implemented in Java. All measurements were obtained on a workstation with AMD 2.6GHz Dual Core CPUs and 4GB memory running Java HotSpot 64-Bit Server VM and Java heap set to 2GB.

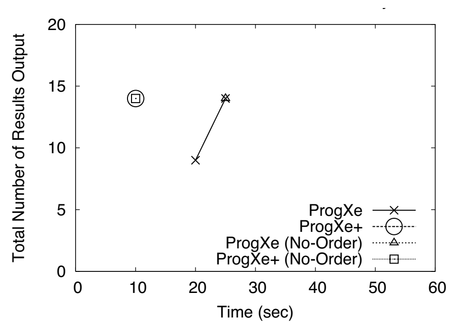
Evaluation Metrics. We study the robustness of our approach by varying: (1) data distributions, (2) cardinality N, and (3) dimensions d. For each setting we measure the following: (1) the time stamp of when the output results were reported by the various algorithms to measure progressiveness, (2) the total execution time to return the complete result set.

Data Sets. We conducted our experiments using data sets that are the *de-facto* standard for stress testing skyline algorithms in the literature [1]. The data sets contain three extreme attribute correlations, namely *independent*, *correlated*, or *anti-correlated*. For each data set R (and T), we vary the cardinality N [10K–500K] and the # of skyline dimensions d. The attribute values are real numbers in the range [1–100]. The join selectivity σ is varied in the range $[10^{-4}$–$10^{-1}]$. The mapping function used is an addition operation between the attribute-values of the corresponding dimensions similar to those in our motivating queries. We set $|R| = |T| = N$.

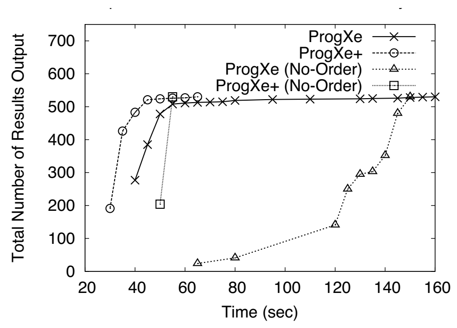
B. Experimental Analysis of ProgXe Variations

Variations of ProgXe. To get a better understanding of the benefits and the cost incurred due to progressive ordering, we implemented the core *ProgXe* framework with the ability to enable or disable the *progressive driven ordering*. Therefore, we now have the first variation, **ProgXe-(No-Order)**, where regions are chosen for tuple-level processing in random. However, in *ProgXe (No-Order)* we enable the *progressive result determination* feature to support early output. The principle of skyline *partial push-through* [1], [10] is complimentary to *ProgXe*. To study the effects of *skyline partial push-through* with ordering, we extended our core approach and *no-ordering* based technique to exploit the *push-through* principle. Thus, we introduce two more variations, namely **ProgXe+** – the core *ProgXe* approach with *push-through* and **ProgXe+ (No-Order)** – which exploits *push-through* but with random ordering. For each dimension d, we chose the same partition size δ for all variations of our proposed approach and all data distributions. For a given dimension d, our core framework is shown to exhibit stable performance across all distributions and thus this enables us to find a good partition size δ [12].

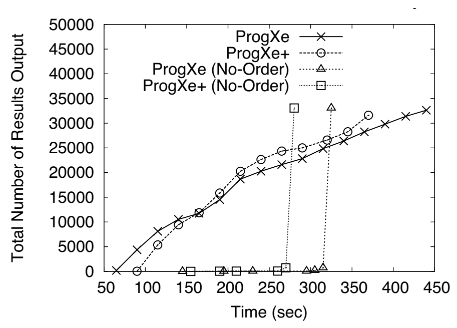
Progressive Result Generation. In Figures 10.a, 10.b and 10.c we compare the total number of results output over time. Correlated data is a skyline friendly distribution since a few 10s of tuples can dominate the entire table. In such a scenario, we observe that both variations *ProgXe+* and *ProgXe+ (No-Order)* show identical performance. *ProgXe* starts producing results from t=20 seconds instead of t=10 seconds achieved by *ProgXe+* and *ProgXe+ (No-Order)*. In Figure 10.b we show that for independent data sets *progressive drive ordering* is able to produce both early as well as faster results. For the anti-correlated data sets, both *ProgXe* and *ProgXe+* techniques are able to report $\approx 25\%$ of the total results before the random ordering techniques start reporting results. For anti-correlated



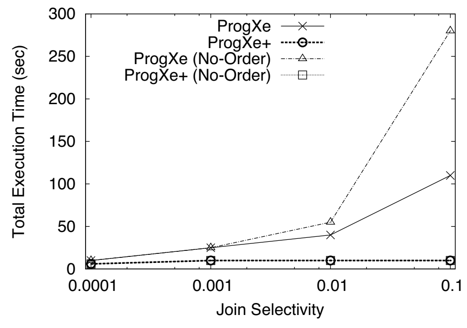
(a) Correlated; σ=0.001



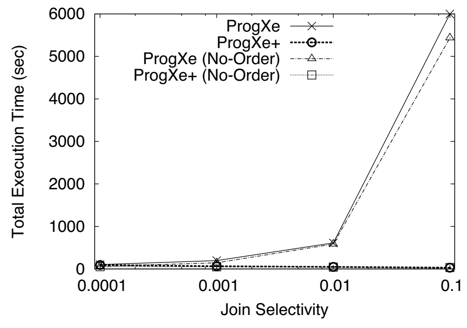
(b) Independent; σ=0.001



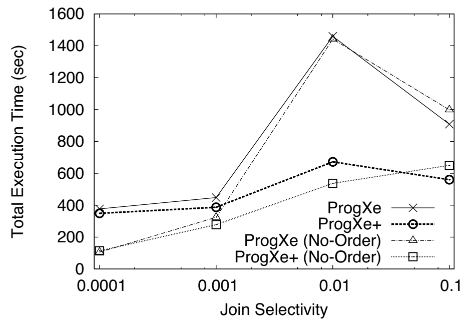
(c) Anti-correlated; σ=0.001



(d) Correlated Distribution

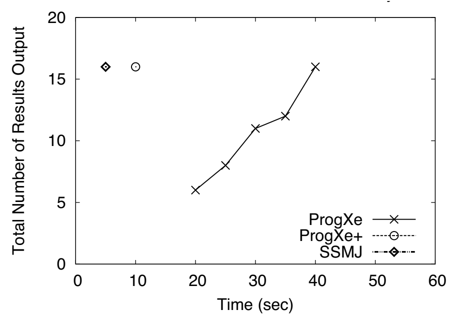


(e) Independent Distribution

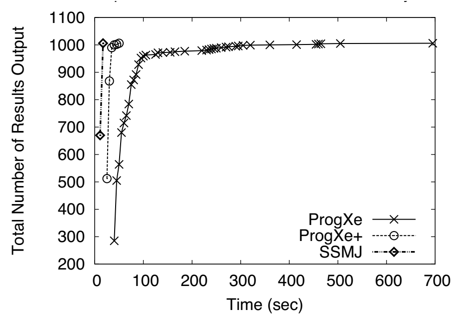


(f) Anti-correlated Distribution

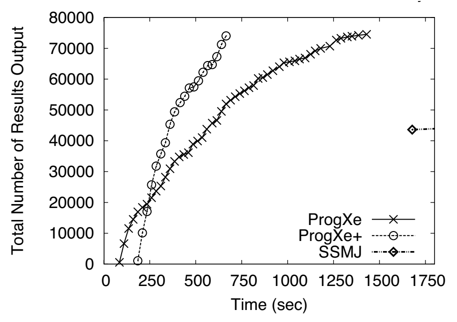
Fig. 10. Performance Study of *ProgXe* and its variations $\big[$*ProgXe+*, *ProgXe (No-Order)* and *ProgXe+ (No-Order)*$\big]$ when d=4 and $|N|$=500K. Progressiveness Comparisons (a, b, and c); Total Execution Time Comparisons (d, e, and f)



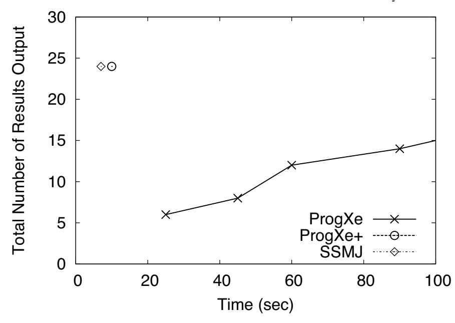
(a) Correlated; σ = 0.01



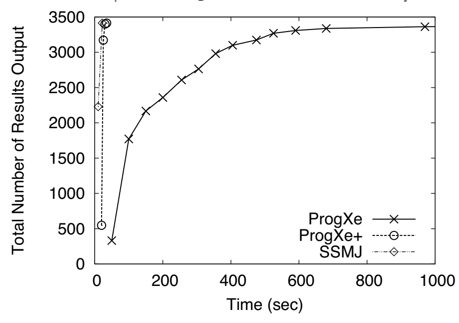
(b) Independent; σ = 0.01



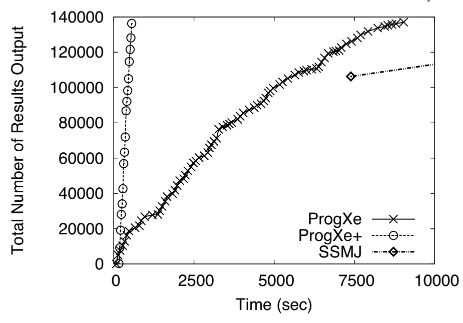
(c) Anti-correlated; σ = 0.01



(d) Correlated; σ = 0.1



(e) Independent; σ = 0.1



(f) Anti-correlated; σ = 0.1

Fig. 11. Progressiveness Comparisons of *ProgXe*, *ProgXe+* and *SSMJ*; $d = 4$ $N = 500K$

data sets, as in Figure 10.c, we observe that *ProgXe* is able to produce earlier results than *ProgXe+*. This is due to the fact that *ProgXe+* consumes time trying to prune the individual sources which in the case of anti-correlated data sets is not cost effective. To summarize, *ProgXe+* is effective in producing early results across all distributions. In contrast, *ProgXe* is best suited for anti-correlated data while still being competitive for the independent and correlated data sets.

Total Execution Time. To measure the overhead of ordering we compare the total query execution time. When $\sigma < 0.01$, Figures 10.d, 10.e and 10.f show that *ProgXe* has identical execution time as *ProgXe (No-Order)*. This highlights that our *ProgOrder* and *ProgDetermine* algorithms are cheap and in fact can be considered negligible in overhead. For $\sigma \geq 0.01$ we observe that ordering tuple-level processing helps reduce the total execution cost of the algorithm. *ProgXe+* and *ProgXe+ (No-Order)* is shown to take about the same time to finish the query evaluation. To summarize, the overhead incurred due to ordering is insignificant but has good progressiveness benefits as shown in Figures 10 a–c.

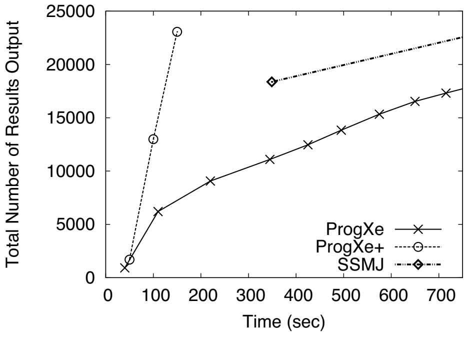
C. Comparisons with State-of-the-art Techniques

JF-SL, *JF-SL+* and *SAJ* techniques all follow the join first skyline later methodology and therefore are *blocking* in nature. Hence, we ignore their comparisons here. However their *execution time* comparisons is presented in [12]. [9] acknowledged that their technique has identical performance characteristics to *SSMJ* for all join selectivities $\sigma \geq 10^{-5}$. Thus we limit our comparative study henceforth to *SSMJ*.

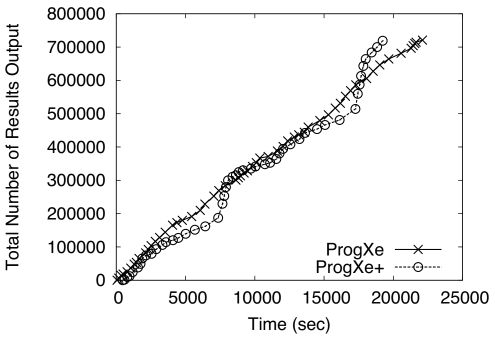
In Figure 11, we compare the progressiveness of the different algorithms when **d=4**. For the non-friendly anti-correlated data *ProgXe* and *ProgXe+* outperforms *SSMJ* by **3 to 4 orders of magnitude**, as shown in Figures 11.c and 11.d. For correlated data, Figures 11.a, and 11.d, we observe that *ProgXe+* has almost similar performance to *SSMJ*. In Figures 11.b and 11.e, for independent distribution, *ProgXe+* has a slightly better performance than *SSMJ*.

For **d=5** and independent data (Figure 12.a) the performance of *SSMJ* is unacceptable as it starts producing tuples later when $t > 350$ $seconds$. In contrast, *ProgXe* and *ProgXe+* take 40 and 50 seconds respectively. Under one minute is considered an acceptable wait time for an interactive system. The slower performance of *SSMJ* is due to the fact that as the number of skyline dimensions increases the pruning capacity of skyline *partial push-through* is dramatically reduced. As *SSMJ* cannot exploit the knowledge of the output space, it can neither optimize for early output of results nor its query execution time. For anti-correlated distribution, *SSMJ* fails to return a single result even after several hours. In Figure 12.b, we observe *ProgXe+* has near identical performance to *ProgXe* since the *push-through* re-write is not as effective and it has to solely rely on the optimization methods proposed in *ProgXe*.

Summary of Experimental Conclusions. (1) The progressive query execution framework *ProgXe* and its optimized *ProgXe+* are robust for all distributions, cardinalities and join factors. (2) The principle of skyline *partial push-thorough* is complimentary for lower dimensions. (3) For anti-correlated data sets, our proposed techniques have superior performance since they output results early; in many cases by 2-4 orders of magnitude. (4) For correlated and independent data sets and ($d \leq 4$), *ProgXe+* is shown to have competitive performance with *SSMJ*. (5) For higher dimensions (d=5) *ProgXe* and *ProgXe+* are superior to *SSMJ* across all distributions and selectivity. In particular, for anti-correlated data sets *SSMJ* is unable to return results even after several hours (Figure 12.b).



(a) Independent



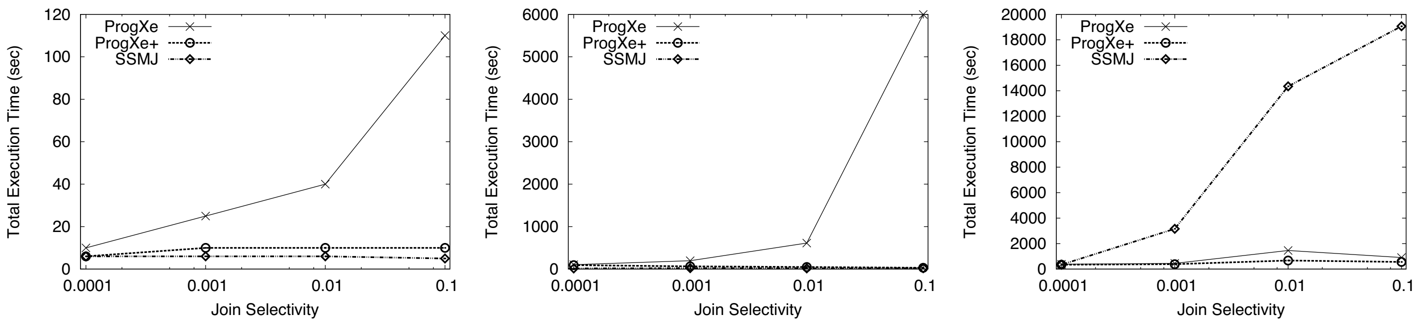
(b) Anti-correlated (SSMJ did not return results)

Fig. 12. Higher Dimension of $d = 5$ and $\sigma = 0.1$; *SSMJ* for Anti-correlated data set fails to return any results even after several hours

VII. Related Work

Progressive Skyline Algorithms. In the context of single-set skyline algorithms, [4], [5] proposed progressive algorithms by pre-loading the entire data-set into *bitmap* or *R-Tree* indices first. However, these techniques are not efficient in the context of SkyMapJoin queries for the following two reasons. First, to ensure correctness when applied to existing methods, the skyline evaluation must be delayed until all possible join results have been generated and loaded into the respective indices, rendering the process fully blocking. Second, for SMJ queries the input to the skyline operation is generated on the fly based on the pipeline of join and mapping operations. In our context of skylines over join, if we used such techniques we would now add on index loading cost as a part of the query processing costs and yet we cannot take advantage of the performance benefits gained in [4], [5].

Skyline Algorithms over Disparate Sources. In the context of returning meaningful results by relaxing user queries, [6] presented various strategies that follow the *"blocking"* paradigm known here as join-first, skyline-later (JF-SL) (as in Figure 1.b). Very recently, [8] proposed techniques to handle skylines over join by exploiting the principle of *skyline partial push-through*. They focus on optimizing the total execution time. Furthermore, this approach suffers from the following three drawbacks. First, *SSMJ* [8] is only beneficial when the local level pruning decisions can successfully prune a large number of objects, like for skyline friendly data sets such as correlated and independent data sets or very high join selectivity [9]. In our experimental study we show that even



(a) Correlated Distribution (b) Independent Distribution (c) Anti-correlated Distribution

Fig. 13. Total execution time comparison: Proposed techniques vs. *SSMJ* ($d = 4$; $N = 500K$)

in data sets where *SSMJ* has good performance our approach performs equally well. Second, after partial push-thorough *SSMJ* for queries such as $Q1$ involving mapping functions once all local decisions have been made, this approach reverts back to the traditional (JF-SL) evaluation of SMJ queries. Since *SSMJ* cannot exploit the knowledge about the output space it is unable to use it to maximize the number of partial results output early. Third, their claim that objects in the source-level skyline of an individual table are guaranteed to be in the output no longer holds when mapping functions are considered as in our context as the later may affect dominance characteristics.

Blocking vs. Non-Blocking Query Operators. Relational algebra operators can be classified as either being *blocking* or *non-blocking* [16]. Operations such as *Select* (σ) and *Project* (π) that can return results immediately after processing each input tuple are called *non-blocking*. On the other hand, operators such as *Group-By* ($\mathcal{G}$), and *Skyline* ($\mathcal{S}$) that require at least one full scan of the entire input data to return any results are called *blocking*. Transforming a *blocking* operation or even a full query into a *non-blocking* (in some case at least *partially-blocking*) operation has received much attention in the literature [16]–[18]. Techniques include to push *aggregates through* the join operations in certain cases or a window-based evaluation of *aggregates*. However, these techniques are not applicable for processing SMJ queries.

VIII. Conclusion

Real-time multi-criteria decision support (MCDS) require to support the early output of results rather than waiting until the end of query processing. To achieve progressive result generation one must be able to take advantage of optimization opportunities that are available by looking ahead into the mapped output space and exploiting this knowledge in various steps of query processing as well as different level abstraction in both individual sources and the complete query. In this work, we propose a progressive query evaluation framework *ProgXe* that is successful in achieving this goal. We further optimize *ProgXe* by presenting an effective ordering technique that optimizes the rate at which partial results are reported by translating the optimization of tuple-level processing into a job-sequencing problem. We demonstrate the superiority of our approach over state-of-the-art techniques in many cases by several orders of magnitude, for a wide range of data sets, confirming the robustness of our methodology.

Acknowledgement

This work is supported by the National Science Foundation under Grant No. IIS-0633930 and CRI-0551584. We thank Dr. Donald Kossmann for giving us access to the de-facto benchmark synthetic data generator for skyline evaluation. We thank David Sampson for his valuable feedback.

References

[1] S. Börzsönyi, D. Kossmann, and K. Stocker, “The skyline operator,” in *ICDE*, 2001, pp. 421–430.
[2] H. T. Kung, F. Luccio, and F. P. Preparata, “On finding the maxima of a set of vectors,” *J. ACM*, vol. 22, no. 4, pp. 469–476, 1975.
[3] I. Bartolini, P. Ciaccia, and M. Patella, “Salsa: computing the skyline without scanning the whole sky,” in *CIKM*, 2006, pp. 405–414.
[4] K.-L. Tan, P.-K. Eng, and B. C. Ooi, “Efficient progressive skyline computation,” in *VLDB*, 2001, pp. 301–310.
[5] D. Papadias, Y. Tao, G. Fu, and B. Seeger, “An optimal and progressive algorithm for skyline queries,” in *SIGMOD*, 2003, pp. 467–478.
[6] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica, “Relaxing join and selection queries,” in *VLDB*, 2006, pp. 199–210.
[7] C. Mishra and N. Koudas, “Interactive query refinement,” in *EDBT*, 2009, pp. 862–873.
[8] W. Jin, M. Ester, Z. Hu, and J. Han, “The multi-relational skyline operator,” in *ICDE*, 2007, pp. 1276–1280.
[9] D. Sun, S. Wu, J. Li, and A. K. H. Tung, “Skyline-join in distributed databases,” in *ICDE Workshops*, 2008, pp. 176–181.
[10] B. Hafenrichter and W. Kießling, “Optimization of relational preference queries,” in *ADC*, 2005, pp. 175–184.
[11] S. Chaudhuri, N. N. Dalvi, and R. Kaushik, “Robust cardinality and cost estimation for skyline operator,” in *ICDE*, 2006.
[12] V. Raghavan, S. Srivastava, and E. Rundensteiner, “Skyline and mapping aware evaluation over disparate sources,” Dept. of Computer Science, Worcester Polytechnic Institute, Tech. Rep. WPI-CS-TR-09-03, 2009.
[13] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson, “On the average number of maxima in a set of vectors and applications,” *J. ACM*, vol. 25, no. 4, pp. 536–543, 1978.
[14] C. Buchta, “On the average number of maxima in a set of vectors,” *Inf. Process. Lett.*, vol. 33, no. 2, pp. 63–65, 1989.
[15] R. Fagin, A. Lotem, and M. Naor, “Optimal aggregation algorithms for middleware,” in *PODS*, 2001, pp. 102–113.
[16] J.-H. Hwang, U. Çetintemel, and S. B. Zdonik, “Fast and reliable stream processing over wide area networks,” in *ICDE Workshops*, 2007, pp. 604–613.
[17] S. Chaudhuri and K. Shim, “Including group-by in query optimization,” in *VLDB*, 1994, pp. 354–366.
[18] W. P. Yan and P.-Å. Larson, “Performing group-by before join,” in *ICDE*, 1994, pp. 89–100.