ProgXe: Progressive Result Generation Framework for Multi-Criteria Decision Support Queries*

Venkatesh Raghavan
Worcester Polytechnic Institute
venky@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute
rundenst@cs.wpi.edu

ABSTRACT

We demonstrate *ProgXe*, a practical approach to support Multi-Criteria Decision Support (MCDS) applications that need to report results as they are being generated to enable the user to make competitive decisions. *ProgXe* transforms the execution of MCDS queries involving skyline over joins to be *non-blocking* by progressively generating results early and often. The demonstration highlights key features of our progressive execution framework that optimizes for early output generation by: (1) evaluating the query at multiple levels of abstraction, (2) exploiting the skyline knowledge gained from both input as well as mapped output spaces. The audience will be able to submit MCDS queries. We provide visualization tools that enable the user to make quick decisions, compare alternative techniques, and provide capability to fine-tune the query predicates based on the early output results.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query Processing

General Terms

Algorithm, Design, Management

Keywords

Progressive Result Generation, Skyline over Join Queries

1. INTRODUCTION

The rapid growth in the number of Internet users has resulted in a variety of on-line services that facilitate commerce and information retrieval. This 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, 6]. 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. Current techniques either handle skylines on single input sets [1, 3, 6] (i.e., no joins) or focus only on reducing the execution time [2, 5] while ignoring the problem of optimization for the production of early results [4].

1.1 Motivating Real-World Applications

Internet Aggregators. The increase in the number 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 form several sources to produce complex results that are then pruned by the skyline operation. For example, an on-line user planning a holiday in Europe visiting both Rome and Paris may have different preferences in each leg of the journey. For instance, as Rome is an ancient city, the user is willing to walk twice as much in Rome than in Paris. In addition, the user may also have 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 an on-line application may want to have these results progressively displayed as they are being computed.

In large real-time applications exploratory queries are common which may or may not deliver satisfactory results even after running for a long duration. In such scenarios, it is essential to support progressive generation which gives the user an opportunity to modify their constraints and preferences in an interactive manner.

Supply-Chain Management. A manufacturer in a supply chain aims to maximize profit and minimize delays. This is achieved by structuring a production and distribution plan through the evaluation of various alternatives. To illustrate, $Q1$ identifies the suppliers that can produce "100K" units of the part "*P1*" and couples them with transporters that can deliver it. The preference is to minimize both total cost and delays. In our demostration, we work with TPC-H benchmark data-sets extended to provide transporter data.

```
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 MAX(tCost) AND MAX(delay)
```

In addition, users with other roles such as *retailers* can submit queries like $Q2$ and others from the literature [2, 5] based on the TPC-D and TPC-DS benchmarks.

```
Q2: SELECT * FROM Web_Sales W, Item I
WHERE W.item_sk = I.item_sk
PREFERRING HIGHEST(W.quantity) AND
MAX(W.net_profit) AND MAX(I.curr_price)
AND MAX(I.wholesale_cost)
```

*This work is supported by the National Science Foundation under Grant No. IIS-0633930 and CRI-0551584.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

1.2 State-of-the-Art Techniques

Several practical considerations make the progressive evaluation of skylines over joins challenging. First, the common approach [1] is to view skyline processing as a disjoint step from join evaluation. Thus, the skyline operation has to wait until all join results have been generated and inspected to even begin to generate a single skyline result over the join results. Second, employing local pruning and sorting at each individual data source as in [2,5] is not feasible for a wide variety of data sets as shown in our full paper [4]. For instance, there are cases when the pruning capacity of local decisions is greatly reduced while still being computationally intensive: (1) as skyline dimensions increases, (2) for anti-correlated or independent distributions, and (3) for higher join selectivity [4]. Therefore, current techniques [2,5] are not robust. Third, these techniques do not resolve the blocking nature of the skyline operation over joins.

Progressive skyline algorithms [3,6] are applicable for skylines over a single set, but do not tackle our problem of processing skylines over joins. This is because in the later case the input to the skyline operation is only known at run-time after join evaluation and thus cannot be preloaded into *bitmap* or *R-Tree* indices.

Top-K queries are different from *skyline queries* since the former fetch the Top-K results from a *totally ordered* set of objects based on a *single* scoring function. Thus the objects returned by Top-K queries may not be part of the skyline since skylines form a *strict partially ordered* set of objects or vice versa [3].

2. THE PROGXE SYSTEM

2.1 System Architecture

Figure 1 depicts the conceptual overview of the progressive query execution framework called **ProgXe**. The **optimization principles** employed in *ProgXe* are: (1) "*avoid generating join results that will not be in the skyline result and avoid evaluating skylines for tuples that will not produce join results*," and (2) "*generate join results that have a higher likelihood being output early first*."

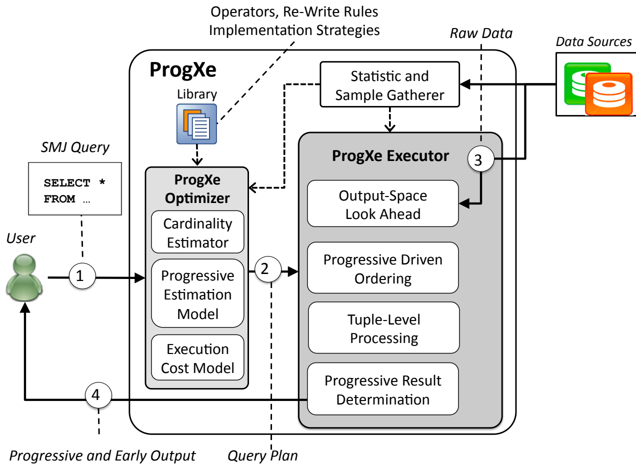


Figure 1: Conceptual Overview of the ProgXe Framework

In our demonstration, a user specifies a preference query over disparate sources (Step 1). The *ProgXe Optimizer* re-writes the query to incorporate our *skyline sensitive operators* that exploit the interactions between the join and skyline operations. To avoid diving directly into the expensive tuple-level join and skyline processing, we propose techniques that can look ahead into the **output join result space** (*output space* for short). This aids us in determining interrelations between the input and output spaces thereby identifying progressive result generation opportunities missed by current techniques. *ProgXe Optimizer* chooses an execution strategy based on our cardinality and cost estimation model (Step 2). Then, the *ProgXe Executor* executes the chosen strategy (Step 3). The *ProgXe Executor* employs our *progressive driven ordering* technique [4] to optimize for early output generation. Lastly, our *progressive result determination* module is able to identify early output results in the *output space* during query execution (Step 4).

2.2 Progressive Execution Strategy

Figure 2 shows the key building blocks of the *ProgXe Executor*. The execution framework is independent of our chosen implementation strategy; the sequence in which these modules are called may differ for each strategy. In this work, we assume the input data sets are already *partitioned* into a multi-dimensional grid structure. The underlying principles of *ProgXe* are general and other space partitioning techniques may also be used.

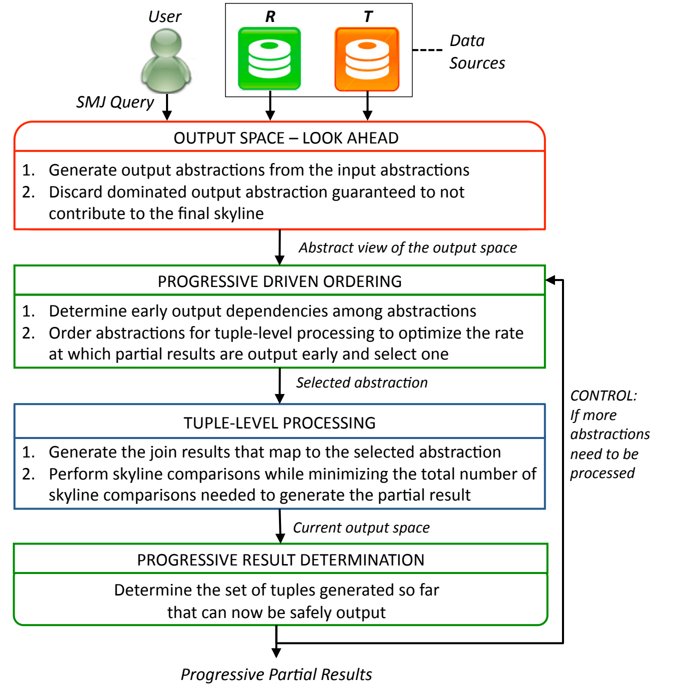


Figure 2: Pipelined ProgXe Executor

Our first step, called **output space look-ahead**, avoids directly diving into tuple-level processing as in [2,5]. Instead, we generate higher-level abstractions in the output space by performing the join operation over the input partitions. More specifically, for a pair of input partitions, one from each table $I^R_a \in R$ and $I^T_b \in T$, we determine: (1) if tuples in these partitions will produce at least one join result, and (2) the *region* of the output space into which the generated join results will fall (denoted as $\mathcal{R}_{a,b}$). To illustrate, let us assume that the domain values of the join attributes are finite and known, while the full treatment of joins is described in [4]. In such a scenario, for each input partition we maintain a list of domain values of the join attribute(s) for the tuples mapped into that particular partition. Therefore, if two partitions share at least one join domain value we can guarantee that their join will result in at least one join result. Subsequently we consider for further processing only output regions that are guaranteed to be populated.

EXAMPLE 1. *Tuples in the input partition of Supplier (R), I^R_1 $[(0,4)(1,5)]$, when joined with tuples in input partition $I^T_2[(0,4)(1,5)]$ from Transporter (T), will result in join results that are guaranteed to fall in the region bounded by the lower-bound point $b(3,5)$ and the upper-bound point $B(6,7)$ in Figure 3, denoted as $\mathcal{R}_{1,2}$.*

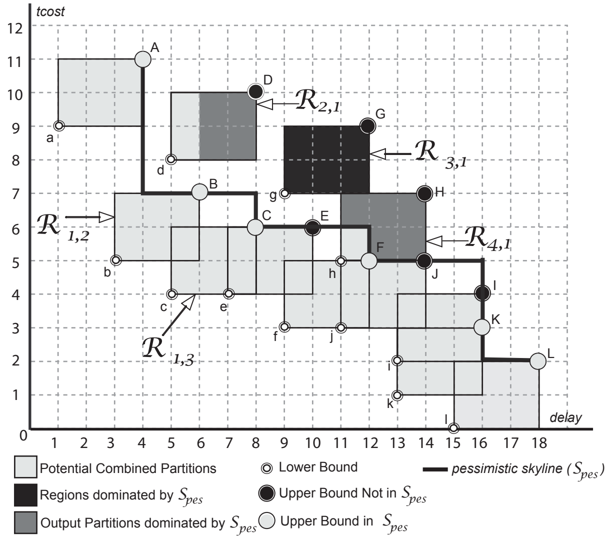


Figure 3: Output Space Look Ahead

For a given set of regions that are guaranteed to be populated during tuple-level processing, we now apply **domination-based reasoning** to further eliminate output regions before they are ever processed. Since such dominated output regions are guaranteed to not contribute to the final skyline they can be safely discarded. This early discarding of dominated regions avoids their join evaluation and any subsequent dominance comparison costs altogether.

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 processing there exists at least one join 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 and can be safely discarded.*

To further reduce the number of skyline comparisons, each region is partitioned into a set of output partitions. Next, we identify output partitions that are dominated by other output regions. This allows us to discard all tuples that map to such dominated partitions since they will not contribute to the final result.

EXAMPLE 3. *In Figure 3 and the 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 above mentioned dominated partitions as "non-contributing". As a consequence, any tuple which is mapped to it will be immediately discarded without even having to conduct skyline computations on it.*

In our second step, **progressive driven ordering**, we investigate the output space to identify 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. We employ a execution-time vs. progressiveness benefit (number of early results) analysis to determine the order in which the output abstractions are sent for the expensive tuple-level processing.

Next, we present the main intuition of our **progressive driven ordering** by highlighting the effects of ordering of regions for *tuple-level processing* to maximize the rate at which the results can be outputted (earlier than later). To motivate, consider a good ordering (produces more results early): $\mathcal{R}_{1,2}$, $\mathcal{R}_{1,1}$, $\mathcal{R}_{1,3}$, and so on, as depicted in Figure 4.a. Following this ordering the join results that map to the region $\mathcal{R}_{1,2}$ are materialized and then their corresponding dominance comparisons are performed first. While examining the entire output space, as shown in Figure 3, we observe that results that map to partitions O[(3,5)], O[(3,6)], O[(4,5)], and O[(4,6)] cannot be dominated by any future generated tuples that map 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 partitions O[(5,5)] and O[(5,6)] may 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 $\mathcal{R}_{1,3}$. The region $\mathcal{R}_{1,1}$ is considered for tuple-level processing next. At its completion, we can safely return tuples that map to all of $\mathcal{R}_{1,1}$'s partitions. To summarize, 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 shown in Figure 4.b. In this ordering, at the end of processing three regions we can only report results that map to 6 partitions. Therefore, we choose the ordering shown in Figure 4.a over that in Figure 4.b.

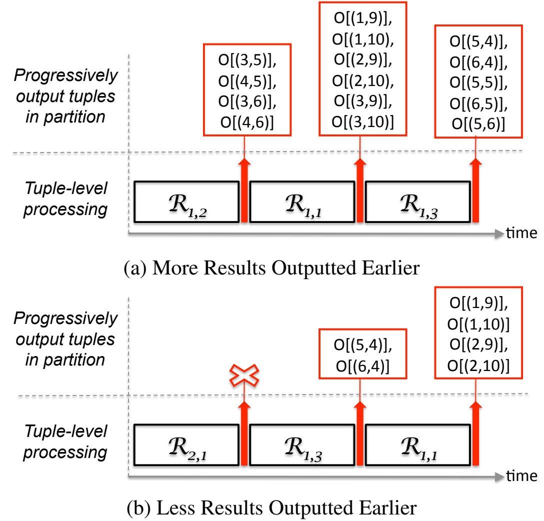


Figure 4: Effects of Ordering on Progressiveness

To support *progressive result generation*, we propose a methodology to identify abstractions that produce the most number of results early with the least amount of time spent on tuple-level processing. To accomplish this, we first determine the maximum number of possible results to be output from each abstraction. More importantly, we identify the relationship between any two abstractions and its effects on returning results early [4].

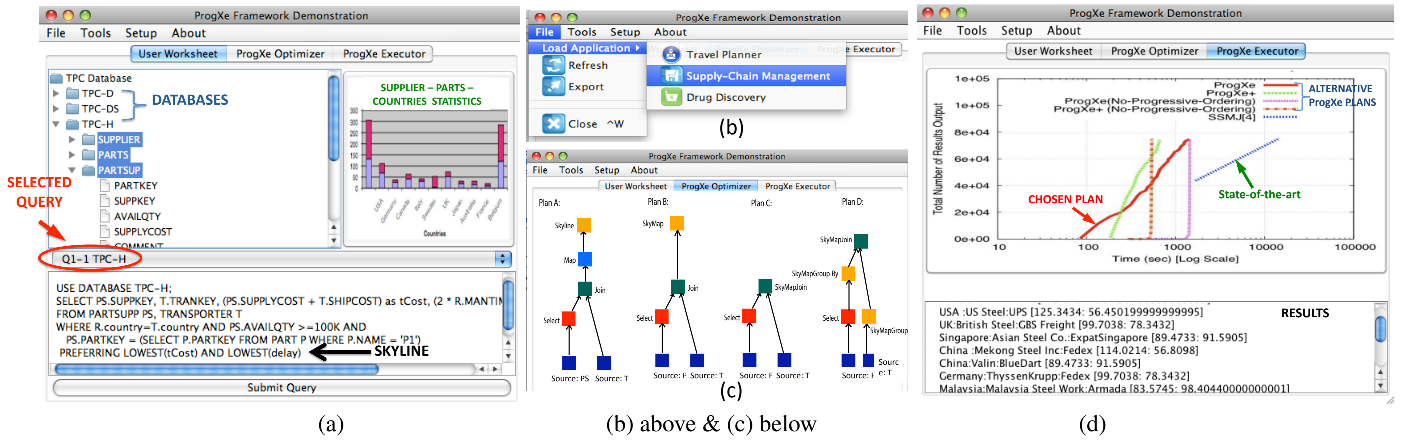
Output abstraction once chosen is then sent for **tuple-level processing**. In this step we: (1) generate the join results that map to their respective output abstraction, (2) map the join results by user-defined mapping functions, and (3) minimize the number of dominance comparisons needed to generate the intermediate tuples.

From the generated intermediate results, our **progressive result determination** step analyzes the dependencies to determine the subset of tuples that can be output early during query execution. The first phase is done once, while the last three phases of *ProgXe executor* are repeated until all output abstractions have either been considered for tuple-level processing or are dominated (thus guaranteed to not contribute to the final result).

3. DEMONSTRATION

3.1 Applications and Data Sets

In our demo, the audience will be able to choose between the "*Trip Planner*" and "*Supply-Chain Management*" applications.



(a) (b) above & (c) below (d)

Figure 5: Demonstration Steps: (b) Select Application; (a) Formulate Query: Submit query and view statistical information; (c) Explore alternative ProgXe implementation strategies; (d) Observe progressive output generation of ProgXe strategies and SSMJ [2]

3.2 Walk-through

We provide the audience with two tracks of visualization, namely the **comparative view** and the **under-the-hood view**. In the *comparative view*, the audience can submit the same MCDS query to both the *ProgXe* framework as well as to the standalone implementation of existing techniques [2, 5], and then visually compare their respective ability to produce early results. Using our GUI, for a query, our *under-the-hood view* provides a step-by-step visualization of the different optimization steps in our *ProgXe* framework. The later view aids the user to understand the intuition behind our progressive estimation and execution model.

Formulating Ad-hoc Queries. The audience will be able to submit queries (similar to $Q1$ and $Q2$ in Section 1.1). Alternatively, users can select from sample queries and customize them if desired.

3.2.1 Comparative View

The audience can dispatch the query to both the *ProgXe* framework as well as the implementation of existing techniques [2, 5].

• **Utilize Progressively Generated Results.** For each implementation strategy, we visualize the output space and plot the results as they are being reported by the different techniques. This provides the audience with the ability to: (1) actually see the attribute values of each reported result (Figure 5.d), (2) compare the quality and diversity of the early results reported by the different techniques, (3) help make an early decision and accept the currently reported results, (4) halt, analyze (the partial result set) and re-formulate the query predicates so that it better fits the user's criteria, (5) get a sense of what data is being returned by the MCDS queries, and (6) we provide a real-time plot of the total number of early results produced over time by each technique (see snapshot Figure 5.d).

• **Performance Showcases.** We visualize the performance benefits of *ProgXe* by comparing it against existing techniques based on the run-time statistics such as: (a) latency of query execution, (b) overhead time spent for each step, (c) total number of join results generated, and (d) total number of skyline comparisons performed.

3.2.2 Under-The-Hood View

The audience will be presented a step-by-step visualization of the different steps undertaken by the *ProgXe* framework.

• **ProgXe Optimization.** The audience will be able to visualize the implementation strategies available in *ProgXe* (see snapshot in Figure 5.c). The GUI will enable the user to see for each strategy: (i) progressiveness characteristics (expected time to receive first tuple), and (ii) expected cardinality of the skyline aware operators.

• **ProgXe Executor.** The GUI will highlight the key principles employed in the four modules of the *ProgXe Executor* (see Figure 2). First, we visualize the coarse grained output space into which the future generated intermediate results will fall. Using the trackpad, the audience can explore this abstract output space. When the user hovers over a particular region in this space, we display the region's estimated cardinality, % of its future generated results that can be output early and output dependencies with other regions in this space. As different regions are selected for execution we highlight the chosen region in our GUI. As intermediate results are generated, we plot them on our output visualization. Intermediate results that are determined to be in the final result are output early and highlighted by a different color. At any point during our demonstration, the audience can pause the query execution to pose questions (via our visual aid) such as (1) Why has a particular region in the output space not been considered for execution? (2) Why has subset of the intermediate results not been marked for early output?

4. CONCLUSION

We demonstrate the *ProgXe* framework, a practical approach to support the progressive generation of results early and often for real-time MCDS applications. *ProgXe* achieves this by taking advantage of optimization opportunities that are available by looking ahead into the output join result space at a high-level of abstraction and exploits this knowledge in all steps of query processing.

5. REFERENCES

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[2] W. Jin, M. Ester, Z. Hu, and J. Han. The multi-relational skyline operator. In *ICDE*, pages 1276–1280, 2007.

[3] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[4] V. Raghavan and E. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In *ICDE*, pages 733–744, 2010.

[5] D. Sun, S. Wu, J. Li, and A. Tung. Skyline-join in distributed databases. In *ICDE Workshops*, pages 176–181, 2008.

[6] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.