SkyDB: Skyline Aware Query Evaluation Framework

Venkatesh Raghavan[1], Advisor: Elke A. Rundensteiner[2]
Department of Computer Science, Worcester Polytechnic Institute, Worcester MA, USA
{[1]venky, [2]rundenst}@cs.wpi.edu

ABSTRACT

In recent years much attention has been focused on evaluating skylines, however the existing techniques primarily focus on skyline algorithms over single sets. These techniques face two serious limitations, namely (1) they define skylines to work on a single set only, and (2), they treat skylines as an "add-on", loosely integrated on top of the query plan. In this work, we investigate the evaluation of skylines over disparate sources via joins. We then propose *SkyDB* - a skyline aware query evaluation framework that addresses four key issues that enable the treatment of skylines as a first-class citizen in query processing. First, we extend the relational model to include skyline-aware operators. Second, for there new operators we design execution strategies that are tuned to exploit the skyline knowledge. Third, we propose our skyline aware query optimizer to effectively choose between the query plan execution strategies. In the literature, we observe that evaluating of skylines over joins is considered to be *blocking*. Therefore, existing approaches focus only on reducing the skyline evaluation time - rendering them inapplicable for response-time sensitive applications. Fourth, we thus aim to transform the execution of skylines over joins to *non-blocking* so that SkyDB can produce progressive output of results. Our preliminary performance study demonstrates the superiority of our proposed methodologies over existing techniques by outperforming them in many cases by several orders of magnitude.

1. INTRODUCTION

The rapid growth in the number of Internet users[1] has resulted in the development of a variety of on-line services to facilitate commerce, social networking and information sharing. This phenomenon has highlighted the need for supporting complex multi-criteria decision support (MCDS) queries [4]. The intuitive nature of specifying a set of user preferences has made Pareto-optimal (or *skyline*) queries a popular class of MCDS queries [2,4,17]. Several efficient algorithms [2, 4, 8, 15, 19] have been proposed to evaluate skyline queries over a *single* homogeneous data set.

[1]http://www.internetworldstats.com/

Proceedings of the Third SIGMOD PhD Workshop on Innovative Database Research (IDAR 2009) June 28, 2009, Providence, RI, USA
Copyright 2009 ACM ...$5.00.

1.1 Motivation

It is a common assumption to view skyline as an add-on operator on top of the traditional SPJ queries. This is rather limiting since a vast majority of MCDS applications in practice do not operate on just a single source. Instead, MCDS applications require to (1) access data from disparate sources via join, and (2) combine several attributes across these sources through possibly complex user-defined functions to characterize the final composite product. To leverage mature DBMS technology and treat the skyline operation as a first-class citizen requires several key ingredients including: (1) introducing skyline-aware query operations and re-write rules, (2) query processing tuned to exploit skyline knowledge, and (3) query optimization considering skylines.

In the literature, executing skylines over joins is viewed as a *blocking* operation. State-of-the-art techniques [4,16] therefore tend to focus on reducing the total execution costs, rendering them a non-viable alternative for many response-time sensitive applications such as on-line auction aggregators, fire monitoring and detection systems [1], and supply chain management (e.g., UPS package tracking system). Such systems need the query processing to have a high degree of responsiveness while still being efficient in producing the overall results. For instance, a user may submit a long running query with the need for continuous output of partial results to facilitate iterative query refinement [18]. To further substantiate these needs we draw from a wide diversity of applications as listed below:

Supply-Chain Management. A manufacturer in a supply chain aims to maximize profit, market share, etc., and minimize overhead, delays, etc. This is achieved by structuring an optimal production and distribution plan through the evaluation of various alternatives.

```
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>=100000
PREFERRING LOWEST(tCost) AND LOWEST(delay)
```

$Q1$ identifies suppliers that can produce "100K" units of part "*P1*" and pairs them with transporters wo can deliver it. The preference is to minimize both cost ($tCost$) and delays ($delay$). In this work, we target such queries which combine the skyline and mapping functions over the join, here known as **SkyMapJoin** (SMJ) queries.

Drug Discovery. The drug discovery life cycle begins with the identification of the lead compound. Molecular modeling plays a vital role by identifying protein-ligand pairs that can points to potential lead compound. This involves screening large data banks of ligands against a protein, and then ranking the protein-ligand pair interactions based on scoring functions over their structure, energy

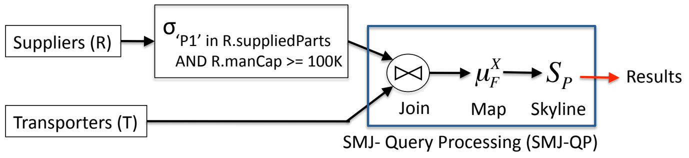

Suppliers (R)
Transporters (T)
σ 'P1' in R.suppliedParts AND R.manCap >= 100K
Join
μ_F^X Map
S_P Skyline
Results
SMJ- Query Processing (SMJ-QP)

Figure 1: Traditional Query Plan for $Q1$

forces or empirical data. The goal is to maximize the intermolecular interaction energy between the two molecules of interest [7].
Query Relaxation. Databases expect precisely defined queries while users seldom have exact knowledge [16, 18]. Therefore, queries over large databases may output an empty answer set even though results of a slightly reformulated query may satisfy the user's needs equally well. However, careless query relaxation can potentially result in large answer sets. The goal is thus to return results that are as close as possible to the original query, i.e., a skyline of results [16].

1.2 Research Challenges

In this dissertation, our overall goal is to integrate the skyline operation into the core query processing framework. Within this broad context, the research challenges we address are the following:

1. As a foundation, we extend the canonical relational model to incorporate skyline-aware operators.
2. Propose alternative methodologies to execute query plans that utilize our skyline-aware operators.
3. Design strategies to support the progressive generation of results thereby reducing response-time for time-critical systems.
4. Develop a skyline aware query optimizer that can effectively choose between alternative execution strategies.
5. Build *SkyDB* a skyline aware query processing framework, that exploits skyline knowledge at all stages of the process.

1.3 SkyDB: Our Proposed Approach

Skyline Aware Algebra. To facilitate the skylines-aware operations, we: (1) extend the canonical relational operators to include operators that are skyline and mapping aware, (2) provide equivalence rules that transform the traditional query plans into plans that now incorporate skyline sensitive operators.
Skyline Aware Query Execution Strategies. To draw a parallel to *Select* in *Select-Project-Join* queries, there are cases when the *select* can be *pushed inside* as well as in some scenarios *pushed-through* joins. Such rewrites can facilitate efficient processing. For instance, pushing *Select* inside the Cartesian product results in theta-join.

In our context, the skyline operation can be viewed as a complex and expensive filter operation. Even though [4, 11] presented some initial thoughts, the problem of efficiently pushing skylines *inside* joins is still an open problem. To tackle this problem, we propose our *SKIN* (SKyline-INside join) technique which is shown to outperform current approaches by several orders of magnitude [20].

In the case of skyline *push-through*, [16] observed that skylines cannot be correctly pushed-through in many scenarios when joins are involved and especially with mapping functions. In some scenarios when the selectivity of the skyline operation is low, the overhead of the skyline *push-through* surpasses its benefits. Further, under scenarios when skylines can be *partially pushed-through*, exploiting the principles of *push-in* can still be beneficial. To exploit both ends of the "*push-in* to *push-through* spectrum" we propose our *Hybrid* approach that utilizes both these principles.

The **optimization-mantra** applied in this work is to "*avoid joining tuples that will not result in a skyline result and avoid evaluating skylines for tuples that will not join*".
Progressive Result Generation: Processing queries such as in Figure 1 with a skyline operation over a join is traditionally viewed as *blocking*. In this dissertation, we plan to: (1) convert the blocking evaluation of an SMJ query into a *non-blocking* operation, (2) guarantee *correctness* while *progressively generating results*, and (3) ensure the *completeness* in results without penalizing the overall execution time of the query.

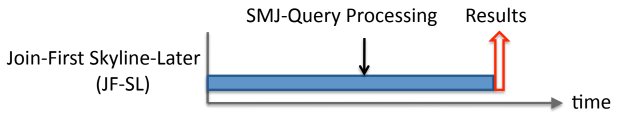


Figure 2: State-of-the-art (Blocking) Approaches

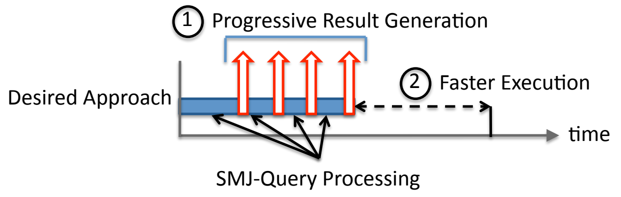


Figure 3: Response-Time Sensitive (Progressive) Approach

Skyline Aware Query Optimization: In order to pick the best execution strategy, we design a cost estimation model for the above proposed execution strategies as well as state-of-the-art techniques. In addition to execution time we also propose to model their system resource utilization such as the amount of buffer space utilized. Next, we design an skyline-sensitive optimizer that utilizes an effective search strategy to pick the best query plan implementation for an SMJ query. To facilitate this, we are looking at designing a cost model to determine the execution costs of the proposed strategies and a cardinality model for the skyline-aware operators.

2. RELATED WORK

Skyline Algorithms over Disparate Sources. Existing techniques view the skyline operation as an add-on after the join (as in Figure 1) and thus follow a **join-first skyline-later** (JF-SL) paradigm [4, 16]. JFSL divides query evaluation into two steps, namely: first produce all possible join results, and second perform skyline evaluation over the entire join results. A marginal improvement fails to avoid the generation of supplier-transporter join results, nor does it significantly reduce the number of comparisons undertaken for skyline computation. [16] extended the popular Fagin's algorithm [9] that still follows the JF-SL paradigm. This approach is attractive only for correlated data where the join operation can be stopped early [4]. These approaches are *blocking* since the skyline evaluation must wait for the join operation to first be completed to return a single result. Alternatively, [13, 21] for each participant table of the join operation first performs operations such as group-wise partial-skyline. Next, the results of these operations are passed on to the join operation to produce join results. Finally, the skyline evaluation over these join results returns the final query results. This approach suffers from the following drawbacks: (1) it does not resolve the blocking nature of skylines over joins, (2) in many scenarios such as anti-correlated or independent data distribution, the pruning capacity of the group-wise partial-skyline is low [6] in comparison to when the data is correlated, and (3) when mapping functions are considered as in $Q1$ the previously proposed property of avoiding skyline computation for some join results [13] are shown to no longer hold [20]. However, the underlying principle used in [13,21] of exploiting skyline partial push-through are complimentary to our work. In other words, whenever partial push-through is a viable option it can be attempted as a filtering operation to our proposed approach. To summarize, as depicted in Figure 2 these techniques have to wait until the end of the join before the skyline evaluation can output a single query result.

Progressive Skyline Algorithms. Progressive skyline algorithms over single-set [19,22] assume that the entire data-set is preloaded into *bitmap* and *R-Tree* indices first. However, these techniques are not efficient in the context of SkyMapJoin queries due to the following two reasons: first, to ensure correctness, the skyline evaluation must be delayed until all possible join results have been generated and is therefore blocking. Second, the input to the skyline operation is generated on the fly based on the join and mapping operations. Since these indices have to be built on the fly, the performance benefits gained in [19,22] are much reduced.

3. PRELIMINARIES

In this section, we review the *preference model* [14] and the *algebra model* used to represent a SMJ query such as $Q1$.

3.1 Preference Model

Each d-dimensional tuple is defined by a set of attributes $A = \{a_1, \ldots, a_d\}$. For a given tuple r_i, the value of the attribute a_k can be accessed as $r_i[a_k]$. $Dom(a_k)$ is 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 tuples 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 $(\forall(a_k \in E)\ (r_i[a_k] \succeq r_j[a_k]) \wedge \exists(a_l \in E)\ (r_i[a_l] \succ r_j[a_l]))$.*

3.2 Relational Algebra Model

For each input tuple r_i the mapping function f_j, in a set of k mapping function $\mathcal{F}$=$\{f_1, \ldots f_k\}$, takes as input a set of distinct attributes $B_j \subseteq A$ to return a value x.

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

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 tuples in R.

4. OUR PROPOSED APPROACH

4.1 Extended Algebra Model

To facilitate the pushing of skylines into as well as through map, join and group-by, we introduce several skyline-aware operators, for example, *SkyMap* ($\widehat{\mu}$), *SkyJoin* ($\widehat{\bowtie}$) and *SkyMapJoin* ($\widehat{\Psi}$).

SkyMap ($\widehat{\mu}_{[\mathcal{F},X,P]}$) performs the following operations in order: (1) apply the set of k mapping functions $\mathcal{F}$ to transform each d-dimensional tuple $r_i \in R$ into a k-dimensional tuple r_i' defined by the set of attributes X, and then (2) generate the skyline of tuples by the preference $P = (E, \succ_P)$ where $E \subseteq X$.

SkyJoin ($\widehat{\bowtie}_{[\mathcal{C},P]}$) combines tuples from its input data sets based on the conditions in $\mathcal{C}$ and returns a set of non-dominated join results based on the preference P.

SkyMapJoin ($\widehat{\Psi}_{[\mathcal{C},\mathcal{F},X,P]}$), a binary operator, performs the following in order: (1) combines tuples from the input data sets based on the conditions in $\mathcal{C}$, (2) applies the set of mapping functions $\mathcal{F}$ to transform each join result into a transformed intermediate result with attributes X, (3) computes the skyline of such results using preference $P = (E, \succ_P)$ where $E \subseteq X$.

4.2 Skyline Aware Execution Framework

We now propose an execution framework methodology that efficiently exploits the skyline knowledge at various steps of query processing. For instance, in the case of join instead of performing the join evaluation at the individual tuple-level, we form a higher-level abstraction of the multi-dimensional data space using partitioning. Thereafter, we exploit the insight that skyline and mapping operations can now be performed at this coarser granularity of partitions instead of directly on individual tuples in the data space. The execution framework facilitates query processing at two levels of data abstractions. The first step of our proposed framework called **marco-level processing** performs the join, mapping and skyline based reasoning at a higher abstraction. The goal of this step is to first generate these abstractions and then exploit the skyline knowledge to prune early as well as plan the strategy of performing the tuple-level processing such that more tuples are outputted early. The macro-level processing iteratively calls for the tuple-level execution performed in the second step of our execution framework called **micro-level processing**.

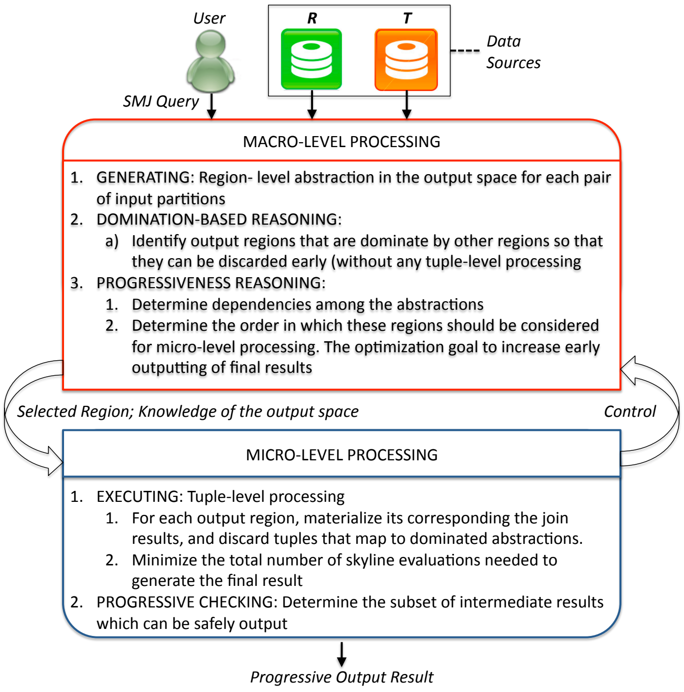

User
R
T
Data Sources
SMJ Query
MACRO-LEVEL PROCESSING
1. GENERATING: Region- level abstraction in the output space for each pair of input partitions
2. DOMINATION-BASED REASONING:
a) Identify output regions that are dominate by other regions so that they can be discarded early (without any tuple-level processing
3. PROGRESSIVENESS REASONING:
1. Determine dependencies among the abstractions
2. Determine the order in which these regions should be considered for micro-level processing. The optimization goal to increase early outputting of final results
Selected Region; Knowledge of the output space
Control
MICRO-LEVEL PROCESSING
1. EXECUTING: Tuple-level processing
1. For each output region, materialize its corresponding the join results, and discard tuples that map to dominated abstractions.
2. Minimize the total number of skyline evaluations needed to generate the final result
2. PROGRESSIVE CHECKING: Determine the subset of intermediate results which can be safely output
Progressive Output Result

Figure 4: Overview of *SkyDB* Execution Framework

Our framework can utilize either our *SKIN* (SKyline INside join) that exploits only the *push-in* principles or our *Hybrid* approach which utilizes both the *push-in* as well as *push-through* principles, for the actual execution strategy. Next, we elaborate on our framework by describing the *SKIN* (SKyline INside join) based execution. The *Hybrid* approach is discussed later in Section 4.3. In this work, we assume the input data sets are already *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 continue to be applicable.

4.2.1 *Macro-Level Processing*

The aim of this step is to perform query execution as well as progressive reasoning at a higher granularity. First, we generate the 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 $I_a^R \in R$ and $I_b^T \in T$, 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 [20]. 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.

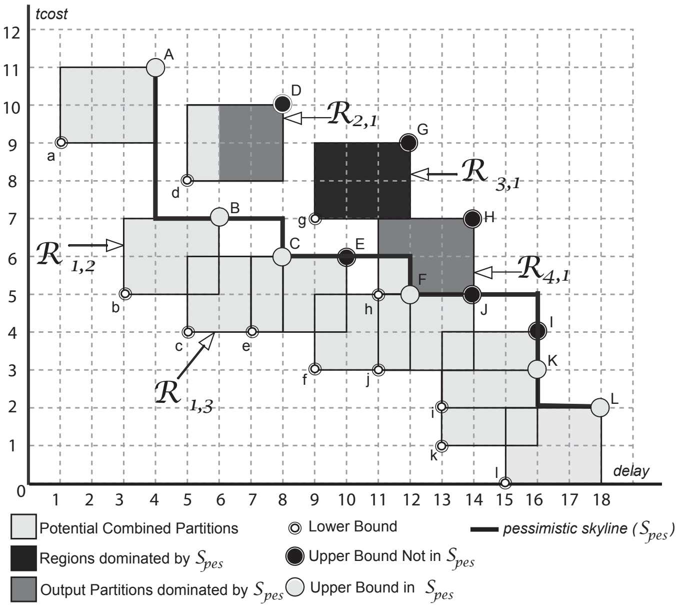


Figure 5: Macro-Level Processing: Avoid Join and/or Skyline Costs

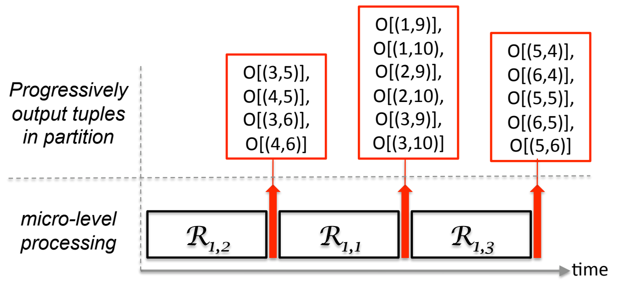
EXAMPLE 1. *Tuples in the input partition of Supplier (R), I_1^R $[(0,4)(1,5)]$, when joined with tuples in input partition $I_2^T[(0,4)(1,5)]$ from Transporter (T), will result in join results that are guaranteed to fall in the region bounded by the lower-bound point $b(3,5)$ and the upper-bound point $B(6,7)$ in Figure 5, denoted as $\mathcal{R}_{1,2}$.*

For a given set of regions that are guaranteed to be populated during micro-level analysis, 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.

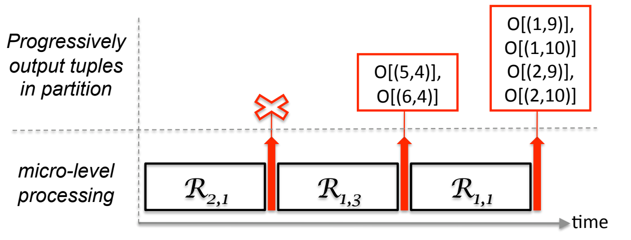
EXAMPLE 2. *In Figure 5, UPPER($\mathcal{R}_{1,3}$) $\succ$ LOWER($\mathcal{R}_{3,1}$. Since $\mathcal{R}_{1,3}$ is guaranteed to be populated during mirco-level execution there exists at least one join result $r_f t_g \in \mathcal{R}_{1,3}$ that dominates the intermediate results that map to $\mathcal{R}_{3,1}$. 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, 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 tuples that map to such dominated partitions since they will not contribute to the final result.

EXAMPLE 3. *In Figure 5 and 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 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.*


Progressively output tuples in partition
micro-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 Outputted Earlier



(b) Less Results Outputted Earlier

Figure 6: Effects of Ordering on Progressiveness

Next, we present the main intuition of our **progressive reasoning** by highlighting the effects of ordering of regions for *micro-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 6.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 5, 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 micro-level processing of region $\mathcal{R}_{1,3}$. The region $\mathcal{R}_{1,1}$ is considered for micro-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 6.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 6.a over that in Figure 6.b.

To support *progressive reasoning*, we propose a methodology to identify abstractions that produces the most number of results early with the least amount of time spent on micro-level processing. Our proposed approach translates this problem into a graph-based job sequencing problem. In this dissertation, we propose a benefit model to accurately predict the number of early results in each abstraction. 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. Lastly, we design a progressiveness driven ordering techniques.

4.2.2 *Micro-Level Processing*

In this query processing phase we finally perform the actual tuple-level join, map and skyline query executions. The optimization goals of this phase are: (1) to reduce the total number of domination comparisons, and (2) to identify results that can be output early. These goals are achieved by piggy backing on the knowledge about the output space gained from the previous step.

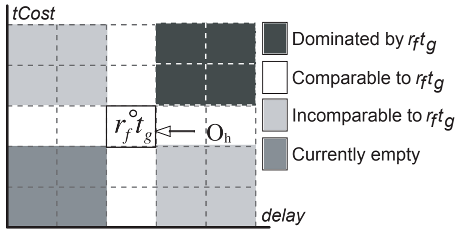


Figure 7: Micro-Level Processing: Avoid Skyline Comparisons

In this step for each output region $\mathcal{R}_{i,j}$, we begin by first **executing** the join operation between the tuples in I_i^R and those in I_j^T. Join results are then mapped to their respective output partition by applying the mapping functions given by the SMJ query. Intermediate results that map to dominated output partitions are discarded. Next, for each intermediate result $r_f t_g$ we **reduce** the dominance comparisons to only a small set of output partitions containing tuples that can potentially dominate it. Based on Figure 7 we observe:

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 have been marked as being dominated.

3. Intermediate results $r_f t_g \in O_h$ can be only dominated by results that map to the slice of partitions that either have the same $tCost$ or the same $delay$ attribute value as O_h.

We quantify the optimization benefits (that is, reduction in the number of skyline comparisons) achieved by our *micro-level processing*. 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 smaller set of $\big((k \cdot d) - (d-1)\big) \approx (k \cdot d)$ partitions.

4.3 The Hybrid Approach

In this approach we aim to efficiently exploit the principles of *partial push-through* as well as *push-in*. More specifically, we use the *partial push-through* principle at a higher level of abstraction. To elaborate consider the scenario with input partitions I_1^R[(2,2)(3,3)] and I_2^R[(4,5)(5,6)]. Additionally, I_1^R has suppliers from *Argentia, Brazil* and *India*, while I_2^R has those from *China and Brazil*. In this scenario, clearly the suppliers from Brazil in I_2^R are dominated by the Brazilian suppliers in I_1^R and therefore can be safely deleted. Next, consider the case when there are no transporters in a number of countries even though these same countries have many suppliers. For example, if there is no authorized transporter in *Indonesia*, it is a waste of execution time pushing skyline-through and calculating the group-level skyline for suppliers in *Indonesia*. The classical *push-through* is blind to the distribution of the domain values. In our proposed approach we instead aim to not perform individual table level skyline evaluation that generates join results that are guaranteed to be later dominated in the mapped output space.

4.4 Skyline-Aware Query Optimization

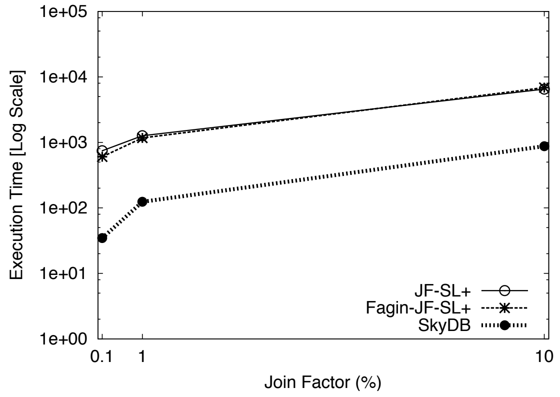
To treat the *skyline operation* as a first class citizen, the query optimizer must be able to choose the best possible plan execution strategy based on the cost estimation for a given SMJ query. To achieve this, we take into consideration the physical implementation of the skyline-aware operators as well as system resource constraints such as the amount of buffer space available. For instance, if there is sufficient memory to hold the intermediate skyline results, then a basic skyline operation can be done in a single scan, where each tuple is compared with all maximal (skyline) tuples found so far. To make an informed decision, we design a cost estimation model for the above proposed execution strategies as well as state-of-the-art techniques. Our proposed model estimated for each execution strategy both execution time we also propose to model their system resource utilization. A key ingredient for designing the cost estimation model is to study the cardinality estimation of the skyline operation and its interaction with other relational operators. Unlike traditional filters such as *select*-clauses, when processing a skyline operator adding a preference can increase the cardinality, even up to the size of the entire relation [6]. Therefore, skyline cardinality estimation is a challenging problem even under the basic assumption such as attribute value independence. The attribute value independence assumption is known to lead to erroneous cardinality estimates even in canonical operators such as joins [12]. This problem is further exacerbated in the context of skyline-aware operators which is very sensitive to the correlation of the different skyline attributes. [6] highlights through experimental analysis that the estimation techniques proposed in [3, 5, 10] are not adequate for real databases. It then presents a robust cardinality estimation technique that relaxes the independence assumption by applying uniform random sampling techniques. Following the approach proposed in [6], we design a cardinality estimation models for our skyline-aware operators. Lastly, propose a query optimizer than explores the various alternative plans to generate the best query execution strategy.

5. PRELIMINARY PERFORMANCE STUDY

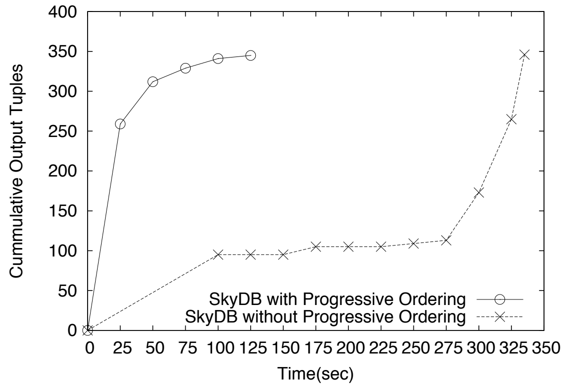
Experimental Platform. All experiments are conducted on a Linux machine(s) with AMD 2.6GHz Dual Core CPUs and 2GB memory. All algorithms have been implemented in Java.
Evaluation Metrics. We measure: (1) the total execution time taken by each algorithm to return the skyline of combined-objects, (2) progressiveness, (3) the total number of intermediate results generated by each algorithm, (3) the total number of domination comparisons required to generate the final skyline, and (4) the time taken by each phase of our proposed algorithm.
Popular Algorithms Compared Against. In our experimental study, we compared primarily against two state-of-the-art techniques. JF-SL$^+$ is an improvement to the JF-SL described in Section 2 that incrementally maintains the skyline. Second, *Fagin-JF-SL*$^+$ [16]

an extension of the popular Fagin algorithm [9] in the context of query relaxation. Our finding confirms [16] that *Fagin-JF-SL*$^+$ is beneficial only for correlated data.

Data Sets. While also conducting with real data, here we describe our experiments conducted over the the *de-facto* standard for stress test data set provided by [4]. These data sets contain three extreme attribute correlations: *independent*, *correlated*, or *anti-correlated*. For each data set R (and T), we vary the cardinality N in the range [10K–500K] and the # of skyline dimensions d. The attribute values are real numbers in the range [1–100]. We use mapping functions, such as addition of the attribute value of the corresponding dimensions similar to those in our motivating queries. We set $|R| = |T| = N$. In addition, to confirm the effectiveness of our proposed approach we also conducted experiments on data sets from UCI KDD Archives[2].



(a) Effects of Join Factor; $|R|$=$|T|$=$|N|$=500K; d = 3; Independent



(b) Measuring Progressiveness

Figure 8: Preliminary Results

Preliminary Performance Comparison Results. *SkyDB* is shown to successfully reduce the total number of comparisons in several cases by orders of magnitude against state-of-the-art approaches (JF-SL$^+$ and Fagin-JFSL$^+$ [16] (as shown in Figure 8.a), for the various data distributions as generated by [4] as well as real data sets. The preliminary findings can be summarized as follows:

1. Robust to all distributions, cardinality and join factors.

2. Unlike JF-SL$^+$ and Fagin-JFSL$^+$ which have to wait until the end to output results, SkyDB is successful in converting the execution of the SkyMapJoin query to be non-blocking by producing *progressive results* (as in Figure 8.b).

3. Outperform JF-SL$^+$ and Fagin-JF-SL$^+$, in many scenarios by several orders of magnitude.

4. Effective in reducing the # join results generated as well as requiring fewer dominance comparisons to compute the final skyline than JF-SL$^+$ and Fagin-JF-SL$^+$.

5. Performance trends observed in experiments over synthetic data sets hold for real data set experiments.

Acknowledgement

This work is supported by the National Science Foundation under Grant No. IIS-0633930 and CRI-0551584. I thank Dr. Donald Kossmann for the de-facto benchmark synthetic data generator. I thank Shweta Srivastava for her help in implementing parts of the SkyDB. Lastly, appreciate the valuable feedback from Dan Dougherty, Murali Mani and DSRG members.

6. REFERENCES

[1] M. H. Ali, W. G. Aref, R. Bose, A. K. Elmagarmid, A. Helal, I. Kamel, and M. F. Mokbel. Nile-pdt: A phenomenon detection and tracking framework for data stream management systems. In *VLDB*, pages 1295–1298, 2005.
[2] I. Bartolini, P. Ciaccia, and M. Patella. Salsa: computing the skyline without scanning the whole sky. In *CIKM*, pages 405–414, 2006.
[3] 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*, 25(4):536–543, 1978.
[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
[5] C. Buchta. On the average number of maxima in a set of vectors. *Inf. Process. Lett.*, 33(2):63–65, 1989.
[6] S. Chaudhuri, N. N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE*, page 64, 2006.
[7] R. Chen, L. Li, and Z. Weng. Zdock: An initial-stage protein docking algorithm. *Proteins*, 52(1), 2003.
[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816, 2003.
[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.
[10] P. Godfrey. Skyline cardinality for relational processing. In *FoIKS*, pages 78–97, 2004.
[11] B. Hafenrichter and W. Kießling. Optimization of relational preference queries. In *ADC*, pages 175–184, 2005.
[12] W.-S. Han, J. Ng, V. Markl, H. Kache, and M. Kandil. Progressive optimization in a shared-nothing parallel database. In *SIGMOD Conference*, pages 809–820, 2007.
[13] W. Jin, M. Ester, Z. Hu, and J. Han. The multi-relational skyline operator. In *ICDE*, pages 1276–1280, 2007.
[14] W. Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.
[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.
[16] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.
[17] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.
[18] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, 2009.
[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, pages 467–478, 2003.
[20] V. Raghavan, S. Srivastava, and E. Rundensteiner. Skyline and mapping aware evaluation over disparate sources. In *submission*, 2009.
[21] D. Sun, S. Wu, J. Li, and A. K. H. Tung. Skyline-join in distributed databases. In *ICDE Workshops*, pages 176–181, 2008.
[22] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[2]http://kdd.ics.uci.edu/databases/census-income/census-income.html