2012-04-17

Supporting Multi-Criteria Decision Support Queries over Disparate Data Sources

Venkatesh Raghavan
Worcester Polytechnic Institute

Supporting Multi-Criteria Decision Support Queries over Disparate Data Sources

by

Venkatesh Raghavan

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

April 17, 2012

APPROVED:

Professor Elke A. Rundensteiner Worcester Polytechnic Institute Advisor	Professor Daniel J. Dougherty Worcester Polytechnic Institute Committee Member
Professor Murali Mani University of Michigan, Flint Committee Member	Dr. Haixun Wang Microsoft Research Asia External Committee Member
Professor Craig Wills Worcester Polytechnic Institute Head of Department	

Abstract

In the era of "big data revolution," marked by an exponential growth of information, extracting value from data enables analysts and businesses to address challenging problems such as drug discovery, fraud detection, and earthquake predictions. Multi-Criteria Decision Support (MCDS) queries are at the core of big-data analytics resulting in several classes of MCDS queries such as OLAP, Top-K, Pareto-optimal, and nearest neighbor queries. The intuitive nature of specifying multi-dimensional preferences has made Pareto-optimal queries, also known as *skyline queries*, popular. Existing skyline algorithms however do not address several crucial issues such as performing skyline evaluation over disparate sources, progressively generating skyline results, or robustly handling workload with multiple skyline over join queries. In this dissertation we thoroughly investigate topics in the area of skyline-aware query evaluation.

In this dissertation, we first propose a novel execution framework called *SKIN* that treats skyline over joins as first class citizens during query processing. This is in contrast to existing techniques that treat skylines as an "add-on," loosely integrated with query processing by being placed on top of the query plan. SKIN is effective in exploiting the skyline characteristics of the tuples within individual data sources as well as across disparate sources. This enables SKIN to significantly reduce two primary costs, namely the cost of generating the join results and the cost of skyline comparisons to compute

the final results.

Second, we address the crucial business need to report results early; as soon as they are being generated so that users can formulate competitive decisions in near real-time. On top of SKIN, we built a progressive query evaluation framework *ProgXe* to transform the execution of queries involving skyline over joins to become non-blocking, i.e., to be progressively generating results early and often. By exploiting SKIN's principle of processing query at multiple levels of abstraction, ProgXe is able to: (1) extract the output dependencies in the output spaces by analyzing both the input and output space, and (2) exploit this knowledge of abstract-level relationships to guarantee correctness of early output.

Third, real-world applications handle query workloads with diverse Quality of Service (QoS) requirements also referred to as *contracts*. Time sensitive queries, such as fraud detection, require results to progressively output with minimal delay, while ad-hoc and reporting queries can tolerate delay. In this dissertation, by building on the principles of *ProgXe* we propose the **C***ontract-* **A***ware* **Q***uery* **E***xecution* (*CAQE*) framework to support the open problem of contract driven multi-query processing. CAQE employs an adaptive execution strategy to continuously monitor the run-time satisfaction of queries and aggressively take corrective steps whenever the contracts are not being met.

Lastly, to elucidate the portability of the core principle of this dissertation, the reasoning and query processing at different levels of data abstraction, we apply them to solve an orthogonal research question – to auto-generate recommendation queries that facilitate users in exploring a complex database

system. User queries are often too strict or too broad requiring a frustrating trial-and-error refinement process to meet the desired result cardinality while preserving original query semantics. Based on the principles of SKIN, we propose *CAPRI* to automatically generate refined queries that: (1) attain the desired cardinality and (2) minimize changes to the original query intentions.

In our comprehensive experimental study of each part of this dissertation, we demonstrate the superiority of the proposed strategies over state-of-the-art techniques in both efficiency, as well as resource consumption.

Acknowledgements

The growth of my knowledge over the last few years culminating with my dissertation is to a huge part due to the inspiration, guidance and friendship I received from my advisor, Professor Elke A. Rundensteiner. She gave me the freedom to explore any topic in database research, provided sound directions at every turn, and the frequent 1 a.m. curve ball emails that pushed my research envelope. I have been fortunate to have her as my advisor, colleague and a friend. I express my sincere thanks for her support, advice, patience, and encouragement throughout my graduate studies. Her excellence in teaching as well as tackling complex research problems will always be my inspiration to return back to academia.

I sincerely thank the members of my Ph.D. committee, Prof. Daniel J. Dougherty, Prof. Murali Mani, and Dr. Haixun Wang for providing me valuable feedback during the various milestones in my Ph.D. Their insight and critique helped me improve the contents of this dissertation. I am thankful for the financial support I have received from my advisor Prof. Elke A. Rundensteiner, Dr. John Woycheese during his tenure as the Professor in the Department of Fire Protection Engineering at WPI, and my department. My thanks also goes to the National Science Foundation (NSF) and National Institute of Standards and Technology (NIST) for providing funding for the computing resources used in my dissertation.

I would like to thank DSRG team members – in particular Manasi Varak,

Shweta Srivastava, Dr. Nam Hun Park, and the students of Mass Academy – Nikhil Thorat, Ariel Wexler and Kwame Siriboe for their hard work in building quality software that we could share and demonstrate together as a team. In addition, I would also like to thank the CAPE team members for their help in my early Ph.D. study. I very much appreciate the discussions as well as friendship of Bin Liu, Maged El-Sayed, Mo Lui, Rimma Nehme, Yali Zhu, Abhishek Mukherji, Mingzhu Wei, Di Wang, Ming Li, and all the other previous and current DSRG members.

The knowledgeable and approachable professors in the Department of Computer Science who are leaders in their respective areas helped me to expand the breadth of my knowledge. The systems and support staff, Mark Taylor, Mike Voorhis, Jesse Banning, Diane Baxter, Refie Cane and Christine Caron, in our department and from the school for providing a state-of-the-art computing infrastructure and a welcoming work environment.

I would like to thank my spouse David Sampson and his family for their patience, support and love during the past few years. His proof reading skills and encouragement was in the end what made this dissertation possible. My parents receive my most sincere gratitude. Their passion to achieve bigger and better things ingrained in me a drive to reach excellence. I would also like to thank my family Vaidyanathan and Radhika for their support throughout my academic pursuit. Lastly, to Irving D. Press for his confidence in me for doing great work.

My Publications

Publications Contributing to this Dissertation

Part I: Skyline and Mapping Aware Query Evaluation

Part I of this dissertation addresses the problem of efficiently evaluating skylines over disparate sources.

1. Venkatesh Raghavan, Shweta Srivastava and Elke A. Rundensteiner, *Skyline and Mapping Aware Join Query Evaluation*, **Information Systems**, Volume 36:6, 2011, pages 917-936.

   Relationship to this dissertation: In this work, we propose SKIN (SKyline INside Join) - an efficient methodology to evaluate SkyMapJoin queries. Chapters 2 and 3 in Part I of this dissertation are based on this work.

2. Venkatesh Raghavan, and Elke A. Rundensteiner, *SkyDB - Skyline Aware Query Evaluation*, **SIGMOD Ph.D. Workshop** 2009.

   Relationship to this dissertation: In this vision paper, we present the overall framework of the skyline-aware query evaluation framework that addresses three key issues that enable the treatment of skylines as a first-class citizen in query processing. First, we extend the relational model to now include skyline aware operators.

Second, for these 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. Third, we thus aim to transform the execution of skylines over joins to non-blocking so that we can produce progressive output of results. This vision paper includes key elements of Parts I and II of this dissertation.

Part II: Progressive Result Generation for Multi-Criteria Decision Support Queries

Part II of this dissertation addresses the motivating need for real-time multi-criteria decision support (MCDS) applications — supporting the early output of results rather than waiting until the end of query processing.

3. Venkatesh Raghavan and Elke A. Rundensteiner, *Progressive Result Generation for Multi-Criteria Decision Support Queries,* **ICDE** 2010, pages 733-744.

   Relationship to this dissertation: In this work, we present ProgXe – a progressive evaluation framework that transforms the execution of MCDS queries involving skyline over joins to be non-blocking by progressively generating results early and often. Part II (Chapters 6-10) of this dissertation is based on this work.

4. Venkatesh Raghavan, and Elke A. Rundensteiner, *ProgXe: Progressive Result Generation Framework for Multi-Criteria Decision Support Queries,* **SIGMOD** 2010, pages 1135-1138, demonstration.

   Relationship to this dissertation: This demonstration highlights the key ideas of *ProgXe*. We provide visualization tools that enable the user to make quick decisions, compare alternative techniques, and provide the capability to fine-tune the

query predicates based on the early output results.

Part III: Contract-Driven Processing of Concurrent Decision Queries

Part III of this dissertation addresses the problem of processing multiple MCDS queries – each augmented by Quality of Service (QoS) requirements.

5. Venkatesh Raghavan and Elke A. Rundensteiner, *Contract-Driven Processing of Concurrent Decision Support Queries: A Piece of CAQE,* **in submission**.

   Relationship to dissertation: In this work, we propose our **C***ontract-***A***ware* **Q***uery* **E***xecution* (**CAQE**) framework that unblocks query processing of multiple skyline-over-join queries by using a multi-granular adaptive execution strategy. Part III (Chapters 11-18) of this dissertation is based on this work.

Part IV: Cardinality Assurance Via Proximity-driven Refinement

We successfully apply the principles of *SKIN* to address an orthogonal research problem – the problem of *Proximity-driven Cardinality Assurance* (*PCA*) that seeks to generate refined queries meeting both cardinality and proximity constraints. This is a collaborative work with Manasi Vartak during her undergraduate studies at WPI.

6. Manasi Vartak, Venkatesh Raghavan and Elke A. Rundensteiner, *CAPRI: Cardinality Assurance Via Proximity-driven Refinement,* **in submission**.

   Relationship to dissertation: In this work, we formally establish the NP-hardness of *PCA*, and propose **CAPRI** – the first framework to address this problem. Part IV (Chapters 19-24) of this dissertation is based on this work.

7. Manasi Vartak, Venkatesh Raghavan, and Elke Rundensteiner, *QRelX: Generating meaningful queries that provide cardinality assurance,* demonstration, **SIGMOD** 2010, pages 1215-1218.

   Relationship to dissertation: In this demonstration we present QRelX – a novel framework that exploits the principle of CAPRI to automatically generate alternate queries that meet the cardinality and closeness criteria.

Extensions

We also extended the core principles of SKIN to present TI-Sky – a continuous evaluation framework to handle skyline queries over time-interval data streams. TI-Sky strikes a perfect balance between the costs of continuously maintaining the result space upon the arrival of new objects or the expiration of old objects, and the costs of computing the final skyline result from this space whenever a pull-based user query is received.

8. Nam Hun Park, Venkatesh Raghavan, and Elke A. Rundensteiner, *Supporting Multi-Criteria Decision Support Queries Over Time-Interval Data Streams,* **DEXA** 2010, pages 281-289.

In the below listed work we present an adaptive approach for determining the processing abstraction levels dynamically at run time instead of assigning a static one at compile time. The adaptive re-partitioning method is triggered by the observed potential benefit for dominance-driven region purging.

9. Shweta Srivastava, Venkatesh Raghavan, and Elke A. Rundensteiner, *Adaptive Processing of Multi-Criteria Decision Support Queries*, **VDLB Workshop**, 2011.

Other Publications

The below listed publications are outcomes of my *Ph.D. Research Qualifier* in continuous stream processing systems and my *Masters'* thesis in XPath query processing at WPI.

10. Yali Zhu, Venkatesh Raghavan, and Elke A. Rundensteiner, *A new look at generating multi-join continuous query plans: A qualified plan generation problem.* **Data Knowledge Eng.** 69(5): 2010, pages 424-443.

11. Venkatesh Raghavan, Yali Zhu, Elke A. Rundensteiner, Daniel Dougherty, *Multi-Join Continuous Query Optimization: Covering the Spectrum of Linear, Acyclic, and Cyclic Queries*, **BNCOD** 2009, pages 91-106.

12. Abhishek Mukherji, Elke A. Rundensteiner, David Brown, and Venkatesh Raghavan, SNIF TOOL: Sniffing for patterns in continuous streams, **CIKM** 2008, pages 369-378.

13. Venkatesh Raghavan, Elke A. Rundensteiner, John Woycheese, Abhishek Mukherji, *FireStream: Sensor Stream Processing for Monitoring Fire Spread,* demonstration, **ICDE** 2007, pages 1507-1508.

14. Venkatesh Raghavan, Kurt W. Deschler, Elke Rundensteiner, *VAMANA - A Scalable Cost-Driven XPath Engine*, **ICDE Workshop** 2005, pages 1278-1287.

Contents

List of Figures

List of Tables

1

Introduction

In recent years we have witnessed the growing accessibility of smart hand-held devices such as iPad™, iPhone™, Android™ devices, and GPS monitoring systems. This coupled with affordable data storage mechanisms over the cloud has resulted in massive amounts of data to be acquired from a wider variety of sources such as Facebook™, Twitter™feeds, web-click streams, and sensor streams. Gartner™(DF11) estimates that in this era of "big data revolution", extracting value from data will not only enable researchers and businesses to address challenging problems such as fire movement predictions and fraud alerts, but also make at least 20% more profit than their counter-parts who do not exploit such valuable nuggets of information. Multi-Criteria Decision Support (MCDS) queries are at the core of big-data analytics providing users with a query framework that supports the efficient execution of multi-criteria data analytical queries over large databases (BKS01, Kie02).

1.1 Background

To address the need to extract value by analyzing multi-dimensional datasets a rich variety of MCDS queries have been proposed over the years. For instance, *Top-K* or ranking queries (IAE03, FLN01, NCS+01, LCIS05), convex hull (Cha93, MZ93, Bre96), nearest neighbor (FTAA01, LLN02, Agg02, APPK08) as well as Pareto-optimal queries known in the database literature as **skyline queries** (BKS01, Kie02).

1.1.1 Skyline Operation

The growing use of the smart-devices has increased the popularity of many real-time smart tools and applications such as on-line text search, Internet aggregators and stock market tickers. Such applications require an intuitive query formulation that enables the user to state the different criteria that are of interest, and return a set of partially ordered results that meet the user preferences. These queries are known in the literature as Pareto-optimal or skyline queries. Skyline computation are characterized by the following features: (1) the user is interested in minimizing (or maximizing) a variety of criteria, and (2) the user wants to analyze the pros and cons of the different options across multiple dimensions rather than being provided with single overall best match (BKS01, CET05). Therefore, the goal of skyline queries is to return a set of alternative results, where each result is better than the others in at least one criteria.

In contrast, a Top-K query requires the user to provide a singular scoring-function (BGS07). Coming up with a single scoring function can be counter-productive since such functions reduce the multi-dimensional comparisons to a single scalar value which losses the multi-dimensional nature of skyline queries. For instance, on a travel website such as Orbitz.com™ to formulate a singular scoring function to combine price, duration of flight and number of stops does not make sense. Instead, if the user is presented with a set of

flight options, each desirable in a subset of dimensions, this will enable the user to make an informed decision in choosing a flight by weighting the pros and cons of the different options.

```
SELECT ID, Features, Reviews
FROM CAR
PREFERENCE HIGHEST(Features), HIGHEST(Reviews)
```

Example 1.1 *To illustrate the concept of skylines, consider the above mentioned skyline query where the user is shopping for a car that has the maximum feature score and the best customer reviews. For the sample data set depicted in Figure 1.1, we observe that the car* $d_5(17, 22)$ *has a lower feature score and customer reviews than the car* $d_7(25, 23)$*. Therefore,* d_7 *is a better choice than* d_5*. Next, consider the pair of cars* $d_7(25, 22)$ *and* $d_9(27, 19)$*. Even though* d_9 *has a higher feature score, its reviews however are lower than* d_7*. Therefore based on the above mentioned preference functions, both* d_7 *and* d_9 *are viable alternatives that need to be reported to enable the user to make an informed decision by weighing the trade-offs between the different result tuples in the skyline.*

1.1.2 Top-K or Ranked Queries

Top-K or ranked queries retrieve the best K objects that minimize a user-defined *scoring (or ranking) function* (FLN01, NCS+01, LCIS05). In other words, from a *totally ordered* set of objects, such queries fetch the top K objects, where the ordering criterion is a *single* scoring function. To illustrate, (FLN01) models the problem of answering the Top-K queries as follows: *"given* n *data objects, and the scoring function* $\mathcal{F}$*, then the problem of Top-K queries is to find the* K *data items with the highest score"* (FLN01). In our motivating skyline query (Example 1 Figure Chapter 1.1) against the CAR table (see Figure 1.1) where each car object has two numerical attributes, *features* and *reviews*.

ID	Features	Reviews
d_1	29	19
d_2	11	26
d_3	26	15
d_4	27	13
d_5	17	22
d_6	14	27
d_7	25	23
d_8	22	20
d_9	27	19

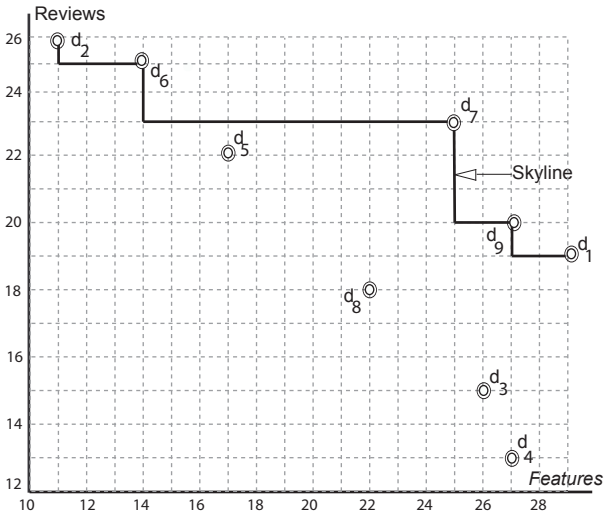


Figure 1.1: A Motivating Skyline Example

To illustrate, let the scoring function add both of these attributes. Then Figure 1.2 shows that the result of the Top-K query where $K = 5$ is $\{d_1, d_7, d_9, d_2, d_8\}$. We can observe that even though d_8 is dominated by the car d_7 it is still part of the Top-5 result set. The result of the Top-K query is in stark contrast to the skyline results $\{d_1, d_2, d_6, d_7, d_9\}$ that contains d_6, which is not in the Top-5 since its score $\mathcal{F}(d_6) = 41$ is low. The reasoning is that the skyline operation returns a set of non-dominated objects based on *multiple* criteria in a multi-dimensional data space from a *strict partially ordered* set of objects, while *Top-K* generates a totally ordered set based on a single scoring function. Thus, there is ultimately no relationship between the results produced by Top-K queries and those generated by skyline queries.

1.1.3 Convex-Hull Query

In computational geometry, *convex hull*, also known as *convex envelope*, is defined as the minimal convex set that contains a given set X of multi-dimensional points (PS85). Convex hull queries are popular in pattern recognition, image processing, and outlier detection applications. In Figure 1.2, the convex hull for the given set is found to be

ID	Features	Reviews	Total Score
d_1	29	19	48
d_2	11	26	47
d_3	26	15	41
d_4	27	13	40
d_5	17	22	29
d_6	14	27	41
d_7	25	23	48
d_8	22	20	42
d_9	27	19	48

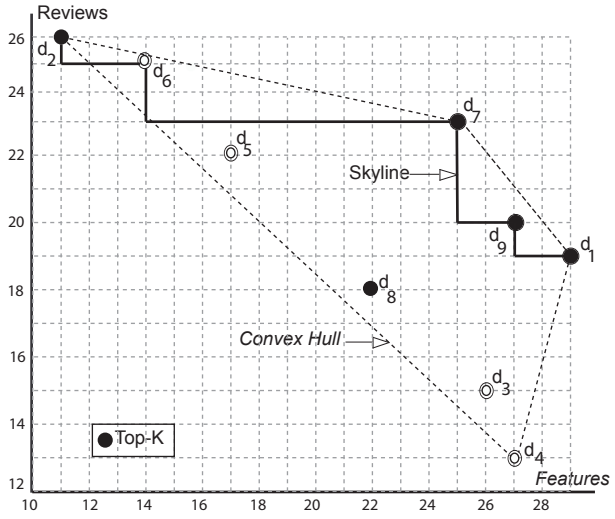


Figure 1.2: Comparison: Top-K vs. Skyline vs. Convex-Hull

$\{d_1, d_4, d_2, d_7\}$. We observe that in the geometric space some skyline points can be hidden behind a convex segment, for example d_6 and d_9. In contrast the results of the convex hull query, representing the outlier points, may not be in the set of skyline results. For instance d_4 is not in the skyline result due to the fact that d_7 is better than d_4 in both *Feature* and *Reviews*, but d_4 is the outlier in the 2-D space and therefore in the convex hull.

1.2 Motivation

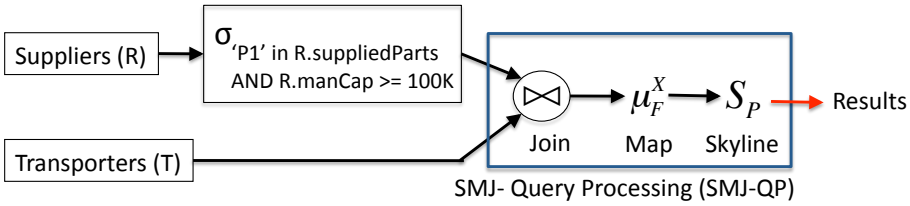
1.2.1 Skyline Evaluation Across Disparate Sources

In recent years, several skyline algorithms have been proposed (BKS01, KRR02, CGGL03, PTFS03, BCP06). The skyline operation, similar to aggregate computations, is traditionally evaluated as the final computation after join and group-by operations in a query plan, thereby assuming its input to be a *single set* of homogeneous data (BKS01). In practice, this common assumption is rather limiting since a vast majority of MCDS applications do not operate on just a single data source (JEHH07). Instead, they are required to: (1) access data from disparate sources with varying schemas, and (2) combine several attributes

across these sources through possibly complex user-defined functions to characterize the final composite product. Below, we first substantiate these requirements by drawing from a wide diversity of applications.

```
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)



(b)

Figure 1.3: Motivating Example: a) Query $Q1$ b) Traditional Query Plan for $Q1$

Example 1.2 (Supply-Chain Management) *A manufacturer in a supply-chain pipeline aims to maximize profit and market share, while minimizing overhead, delays and losses. This is achieved by structuring an optimal production and distribution plan through the evaluation of various alternatives. For example, query* $Q1$ *as shown below represents a user query submitted in a globalized market where products can be purchased in any of the world markets.* $Q1$ *in Figure 1.3.a first identifies the suppliers that supply the part "P1", with the manufacturing capacity of "*$100K$*" and couples them with transporters, that supply to the same country as the supplier, to generate potential supplier-transporter pairs. The user preference in* $Q1$ *aims to reduce both total per-unit cost as well as time delay. The total per-unit cost includes the per-unit wholesale cost combined with the per-unit shipping charges, while the delay encompasses the time needed to ship the finished*

products in addition to the manufacturing time.

Example 1.3 (Internet Aggregators) *The rapid increase in the number of on-line vendors has resulted in growing popularity of Internet aggregators such as Froogle™ for durable goods and Kayak™ for travel services. However, these aggregators do not support skyline-over-join queries such as $Q2$, limiting the user experience by producing potentially less desirable query results. For example, in motivating query $Q2$, as shown below, the user is planning a vacation in Europe that includes visiting both Rome and Paris. In $Q2$ the user has different preferences in each leg of the journey. For instance, the user is willing to walk twice as much in Rome than in Paris, since the ancient parts of Rome have narrow roads and have to be discovered on foot. In addition, the user has a cumulative goal of minimizing the total cost of the trip.*

```
Q2: SELECT R.id, T.id, (R.price + T.price) as tCost,
(2 * R.distance+ T.distance) as tDistance
FROM RomeHotels R, ParisHotels T
PREFERRING LOWEST(tCost) AND LOWEST(tDistance)
```

Example 1.4 (Drug Discovery) *The life cycle for drug discovery is a long process spanning a decade or more, beginning with the identification of an initial lead compound. An iterative process of synthesis and assaying analogs of the compound is carried out until a final drug is discovered and later introduced to the market. Molecular modeling plays a vital role in drug discovery and is used to identify protein-ligand pairs that can point to potential directions of further investigation. This involves screening large data banks of ligands against a protein, and then ranking the protein-ligand pair interactions according to a multi-dimensional scoring functions based on structure, energy forces or empirical data of each pair with the goal of maximizing the intermolecular interaction energy between the two molecules of interest (CLW03).*

In this dissertation, we target applications such as those listed above that require multi-criteria decision support over disparate data sources.

1.2.2 Progressive Evaluation of MCDS Queries

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. 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** (TEO01, PTFS03). 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. To substantiate consider the following use case:

Example 1.5 (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 (KLTV06). 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 (KLTV06). 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 refinement (MK09).*

1.2.3 Contract-Driven Query Processing

Multi-Criteria Decision Support (MCDS) systems supporting applications from business intelligence to real-time event detection must handle workloads composed of queries with varying degrees of responsiveness (a.k.a. Quality of Service) (ACc$^+$03). For example, some applications such as a stock market trend analyzer cater to user queries that need to identify real-time trends, while the overall analysis can be delayed to the end of the trading day. In contrast, applications that require complete results such as drug modeling require the overall execution of the user query to be optimized and are not interested in partial results. To summarize, some real-world applications can indeed receive queries that predominantly have one optimization goal while others may cater to queries that have a mixture of optimization goals. Therefore, there is a need to provide an evaluation methodology that can cater to multiple optimization goals.

In this work, we tackle the problem of handling queries with diverse QoS demands, henceforth called **Contract-driven Multi-Query Processing** (Contract-MQP). In this dissertation, we target *skyline-over-joins* query workloads. Consider the following real-world use case:

Example 1.6 (Travel Planner) *A travel planner enables users to search Hotels (H) and Tours (T) to find competing packages. Consider the following workload of preference queries $S_Q = \{Q3, Q4, Q5\}$.*

- *Q_3: John Smith is planning a business trip to Paris that minimizes the distance from the venue; while maximizing the rating. John is on a break in-between meetings, and has 10-15 minutes to quickly narrow down his top choices.*
- *Q_4: Student Jane Doe is searching for vacation deals in Paris that are cheap while compromising on the distance. She wishes to be alerted about attractive packages as soon as they are identified to facilitate immediate action.*

- *Q_5: ACME travel agency designs competitive European tours. The preference is to maximize ratings while minimizing costs and distance to produce hourly reports.*

These three skyline-over-join queries perform joins across the same base tables but differ in criteria that the users wish to optimize for. Furthermore, the users differ in their expected system responsiveness i.e., QoS requirements.

1.2.4 Automated Query Refinement in MCDS Systems

Databases today are becoming adept at processing petabytes of data, handling complex schemas, and supporting computationally expensive queries (CDD+09). However, users often endure a frustrating, repetitive query specification process before they can get the desired information from a database (JCE+07). The primary cause of this difficulty is that end-users are ill-equipped to formulate precise queries: (1) they are often "in the dark" about the available data (LFW+11), and (2) the inherent flexibility of user requirements cannot be captured through SQL (BP05). Consequently, the user must manually refine queries through a cumbersome and resource-intensive trial-and-error process with the aim to meet the desired cardinality while simultaneously preserving original query semantics. We drive home the importance of the automation of the query refinement process in a competitive MCDS system via the following use cases:

Example 1.7 (Medical Research) *Jane Doe has received a grant for obesity research where she has funding to study 5000 volunteers through interviews and health checks. Jane runs query $Q6$ to identify adults having BMI greater than 30, aged below 30 years, low income, and engaging in moderate weekly exercise. For Jane, the first two predicates are rigid or hard constraints because the definition of obesity is universal and the age threshold delimits her target study group. In contrast, the last two predicates are flexible or soft constraints since the precise definitions of "low income" and "moderate exercise"*

may vary with geographic areas and relevant patient records. For Jane to analyze volunteer data following the study, she has to know not only which volunteers were selected but also how they were selected, i.e., the exact demographic attributes used for their selection. To illustrate, to draw a hypothetical conclusion that lower income leads to obesity, Jane must know the precise income and BMI selection criteria for all her volunteers; merely having a set of result tuples or a ranking function provides insufficient information.

```
Q6:SELECT * FROM PatientRecords WHERE
(18 < age < 30) AND (BMI > 30) AND
(income < $60,000) AND (weeklyExercise > 5)
```

Assume that when Jane runs $Q6$, she can only find 1600 volunteers. Since her initial query was too strict, she must now relax the *soft constraints* of income and exercise to identify more volunteers. At the same time, she must be careful to not grossly alter the original query predicates. For instance, increasing the income threshold to $65,000 may be acceptable, but increasing it to $95,000 will violate the spirit of low income criteria. While refining $Q6$, Jane must thus satisfy two orthogonal constraints: (1) Desired Cardinality: to return exactly 5000 volunteers, and (2) Query Proximity: to preserve original semantics by minimally changing $Q6$. Manual query refinement with these goals is a frustrating trial-and-error process because Jane is unfamiliar with the underlying database. She could (potentially) attempt a very large number of refinements without any guarantee of success, and therefore waste time, effort, and system resources.

Example 1.8 (Financial Services) *John Smith's bank is offering special low-interest property loans to 2000 young customers who have good credit scores, high incomes and are planning to buy homes in specific areas across the country. The bank uses query $Q7$ shown below to select 2000 customers. Similar to our previous example, John requires to know not only which customers were selected for the offer, but also how they were selected.*

Response to the current offer can be analyzed based on customer selection attributes to plan future strategies. However, unlike query $Q6$, we observe that all predicates in $Q7$, i.e., "young customers", "good credit", "high income" and location, are soft constraints because their definitions can vary.

```
Q7: SELECT * FROM Customers C, Locations L
WHERE (C.zipcode = L.zipcode)
AND (C.age < 35) AND (C.creditScore > 730)
AND (C.income > $75,000)
```

Suppose John only obtains 1200 results from $Q7$ and must manually refine the query. Since all constraints in $Q7$ can be categorized as *soft constraints*, the query can be modified by relaxing any combination of predicates. Once again, John must strive to ensure that his refinements don't grossly alter the original query. Refining the location-based join predicate in $Q7$ is particularly attractive because it can enable the bank to target customers meeting the age, income and credit criteria but buying property at a slightly different zip code. However, manual join refinement is very challenging because the consequent change in cardinality is unpredictable and repeated join execution is resource-intensive in terms of time and system utilization.

1.3 State-Of-The-Art Techniques

1.3.1 Skyline Algorithms over a Single Relation

The majority of research on skylines has focused on the efficient computation of a skyline over a single set (BKS01, KRR02, CGGL03, PTFS03, BCP06). This can be broadly categorized as *non-index* and *index-based* solutions. *Block Nested Loop* (BNL) (BKS01) is the straightforward non-index based approach that compares each new object against the

skyline of objects considered so far. The *Sort Filter Skyline* (SFS) (CGGL03) improves on BNL by first sorting the input data by a monotonic function. Nearest Neighbor (NN) (KRR02) and Branch & Bound Search (BBS) (PTFS03) are index-based algorithms. In Chapter 5 we present a more detailed description of these state-of-the-art techniques.

1.3.2 Skylines over Disparate Sources

The naïve strategy to execute the skyline-over-join is to follow the **Join-First, Skyline-Later** paradigm, which divides the execution of the operator into two disjoint steps. First, the join operation is performed in totality and then the skyline computation is performed over the join results. A marginal improvement over this strategy would be to incrementally compute the skyline after the generation of each join result, thereby slightly reducing the number of skyline comparisons. However, this approach fails to avoid any join evaluations which is an expensive process. In the context of returning meaningful results by relaxing user queries, (KLTV06) is one such approach to follow the join-first, skyline-later (JF-SL) paradigm. This approach does not consider mapping functions. In fact, it is shown to be effective only for correlated data where the combined-object generation can be stopped early (BKS01) confirming the findings presented in (KLTV06).

(JEHH07, JMP$^+$10) proposed *SSMJ* (*Skyline Sort Merge Join*) technique to handle skyline-over-join by primarily exploiting the principle of *skyline partial push-through*. This approach suffers from the following three drawbacks: First, SSMJ is only beneficial when the local level pruning decisions can successfully prune a large number of objects, like skyline friendly data sets such as correlated and independent data sets or those with very high selectivity (SWLT08). Second, the guarantee that objects in the set-level skyline of an individual table clearly contribute to be in the output no longer holds here. This is so because SSMJ does not consider mapping functions which can affect dominance characteristics. Third, since SSMJ follows the same *skyline first- join later* it is unable to

exploit this knowledge to reduce the number of dominance comparisons. Following the principles proposed in (JMP+10), recently (VDP11, KML11) proposed alternative sorting based techniques.

1.3.3 Progressive Skyline Query Evaluation

In the context of single-set skyline algorithms, (TEO01, PTFS03) 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 skyline-over-join 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 skyline-over-join 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 skyline-over-join, if we used such techniques we would now add the cost of load the join results into an index and yet without being able to take advantage of the performance benefits gained in (TEO01, PTFS03).

1.3.4 Handling Concurrent MCDS Queries

In recent years we have witnessed a rapid increase in the size of databases and the number of concurrent users for any particular application in the domain of consumer and/or business services, scientific and engineering applications, critical services such as law enforcement, and defense/crisis management (BKS01, WOT10). The user queries for such applications range from trivial look-ups to computationally intensive queries such as those in drug modeling and fire detection and monitoring. It is easy to envision in such domains the application receiving multiple concurrent queries over the same large data

sets. To provide scalable performance and acceptable quality of service to all end-users such load intensive systems must: (1) avoid processing each computationally intensive query independently, (2) identify concurrent queries that can exploit the sharing of the results of sub-expressions that are common, (3) minimize unnecessary retrieval of tuples from disk to avoid processing and storing of tuples that will not contribute to the results of any user queries.

Multi-query processing is typically solved by one of two methodologies. The **time-shared approach** (KGM92, RCL01, NW11) partitions the total available processing time into slices and allocates it to different queries in a round-robin fashion. In this approach each query is processed separately with no sharing of intermediate results for common sub-expressions. This makes the *time-shared approach* a non-attractive choice for processing workloads with resource intensive skyline-over-join queries.

To elaborate consider tables R and T with cardinalities of 200K and 100K respectively. A query plan containing a single join filter with selectivity of $\sigma = 0.1$ and skyline dimensions $d = 2$ will generate ≈ 1 million join results and require > 1 million pairwise comparisons (CDK06). Clearly, processing multiple skyline-over-join queries individually is prohibitively expensive.

Alternatively, the **shared query plan approach** pipelines each tuple through a shared multi-operator plan (KFHJ04, WOT10, MPK00, HRK$^+$09, DSRS01). Each query is viewed as a *subscriber* that consumes the results produced by a *producer* operator within this larger *integrated plan*. The later approach is superior to the former due to its effectiveness in reducing the computation load.

1.3.5 Automated Query Refinement

In recent years, several existing techniques such as empty result refinement, skyline queries, Top-K, and approximate query answering (Gaa97, ML05, Mus04, BKS01, KLTV06,

CG99, CD03, KWFH04, IAE03, CK97, MMY09) only produce individual result tuples and ignore the challenge of providing a concise, easy-to-understand SPJ query specification for generating the tuples. Such techniques address the question of *what* results are produced rather than *how* the results are produced. However, result tuples alone are insufficient in many applications. For instance, scientific research applications (Example 1.7 in Chapter 1.2.4) require precise query specification for post-experimental analysis of data and repeatability of experiments. Similarly, business intelligence applications (Example 1.8 in Chapter 1.2.4) require queries for market analysis and strategic planning. Recently, (MK09) proposed an interactive framework to help users to manually refine queries. This approach however can still be tedious and may not ensure query proximity. In the context of testing, (BCT06, MKZ08) proposed techniques to generate test queries meeting cardinality constraints while disregarding query proximity. Furthermore, existing techniques largely address select query refinement as opposed to the challenging problem of join query refinement.

1.4 Research Challenges Addressed in This Dissertation

In this section, we highlight the research challenges in addressing the four research topics of this dissertation, namely **skyline aware evaluation over disparate source**, **progressive query evaluation of MCDS queries**, **contract-driven processing of multiple MCDS queries** and **proximity-driven cardinality assurance**.

1.4.1 Efficiently Processing Skylines over Disparate Sources

In the literature (BKS01, KRR02, CGGL03), the skyline operation, similar to aggregate computations, has traditionally been evaluated last after the join and group-by operations in a query plan. Therefore, these techniques make the assumption that the input is a *single*

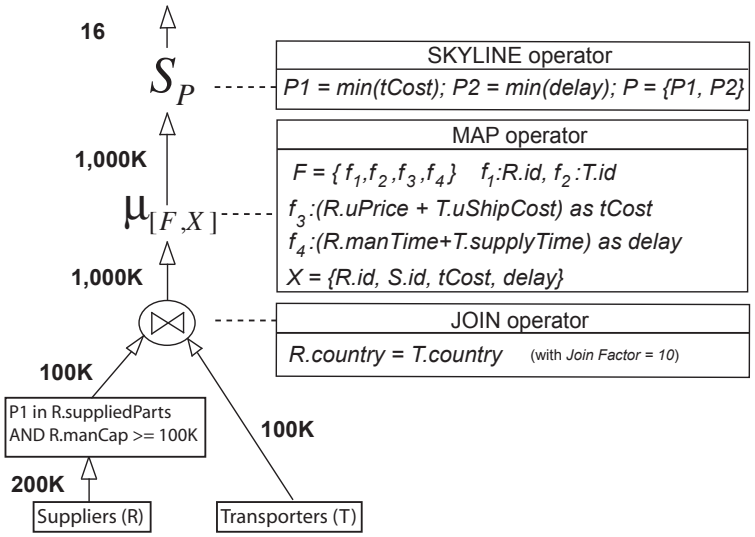


Figure 1.4: Traditional Query Plan For User Query $Q1$

set of homogeneous data (BKS01). In other words, prior to our publications (RR09, RR10, RRS11), core techniques proposed in this dissertation, the common strategy in the literature to execute queries such as $Q1$ and $Q2$ (see Chapter 1.2) had been the *join-first, skyline-later* (JF-SL) paradigm discussed earlier in Chapter 1.3.2 that divides the query execution into several disjoint steps.

Example 1.9 *Figure 1.4 illustrates the traditional Join-First Skyline-Later (JF-SL) execution strategy using the motivating query $Q1$. First, the objects in relation R that satisfy the selection conditions, part "P1" in R.suppliedParts AND R.manCap $\geq$ 100K are selected. Next, the join operation is applied to generate supplier-transporter pairs. These join results are transformed by the mapping operation. In this dissertation, we refer to the mapped join results as* **combined-objects***. Finally, the skyline computation is performed to return the set of non-dominated supplier-transporter pairs.*

In this data set with cardinalities $|R|$=200K and $|T|$=100K objects, the pruning capacity of the selection conditions is 0.5 while the join factor is 10. Therefore, JFSL will first generate all 1 million supplier-transporter join results. Next, consistent with the estimation proposed by (CGGL03) the skyline computation requires on the order of mil-

lions of comparisons. A marginal improvement would be to incrementally compute the skyline over the supplier-transporter pairs produced thus far, after each newly generated supplier-transporter pair. This reduces the total number of intermediate results maintained by the algorithm. As our experimental evaluation in Chapter 4 later on confirms, this would still require ≈ 1.2 million comparisons. However, this marginal improvement fails to avoid the generation of any supplier-transporter join results nor does it significantly reduce the number of comparisons undertaken for skyline computation.

In many real world applications the attribute values exhibit anti-correlation. For example, in a hotel reservation application the cost of the hotel increases with the nearness to popular tourist interests, and in an automobile purchase system the mileage on the car is inversely proportional to its asking price. Existing techniques proposed to handle skyline over joins (JEHH07, JMP$^+$10, SWLT08) are not effective for such anti-correlated data sets. For this reason, these techniques in the literature focus primarily on the skyline-friendly correlated and independent data sets (SWLT08). The focus of this work is to present a robust evaluation strategy that handles all the three extreme distributions, namely *independent*, *correlated* and *anti-correlated* data distribution, identified by the pioneering work in (BKS01) as the de-facto standard for testing skyline algorithms.

As illustrated in Example 1.9 by treating the skyline operation as an "add-on" to the query plan these state-of-the-art techniques miss potential optimization opportunities. We address this shortcoming in the literature we first extend the mature DBMS technology such as pushing (when possible) skyline operation through joins (BKS01, Kie02) to now be applied at various levels of data processing. Next, we design skyline-aware query operators that can exploit the properties of Pareto-optimal queries at various stages of query processing. And lastly, we build sophisticated evaluation methodologies for these skyline-aware operators.

1.4.2 Progressive Skyline over Join Evaluation

Multi-Criteria Decision Support (MCDS) applications such as web searches, B2B portals and on-line commerce need to report results early; while they are being generated, so that users can react and formulate competitive decisions in near real-time. However, state-of-the-art skyline techniques that support progressive query evaluation focus only on handling skylines on single input sets (i.e., no joins).

Since traditional techniques (KLTV06, BKS01) (Figure 1.3.b) follow the JF-SL execution paradigm, 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*. (JEHH07, JMP$^+$10, SWLT08) proposed techniques exploiting the principle of skyline *partial push-through* (HK05, BKS01) on each individual data source. However, as the number of dimensions increases the pruning capacity of the push-through principle is greatly reduced, sometimes up to the size of the entire source (CDK06). Since the skyline *partial push-through* is itself blocking, the local pruning employed in (JEHH07, SWLT08) may be computationally intensive without yielding a single progressively generated partial result. In addition, (JEHH07, JMP$^+$10, SWLT08) are unable to look ahead into the output space to make decisions that can further optimize progressive result generation.

1.4.3 Contract-Driven Multi-Query Processing

The Contract-MQP problem is especially difficult for skyline-over-join workloads for the following reasons: The set-based skyline over join operation is blocking – in the worst case all the join tuples to be processed by the skyline operator may need to be generated to return a single progressive skyline result (CDK06). Therefore, the skyline operators in

the *shared query plan approach* may be blocking the progressiveness of other operators of the plan. In our case this may even be other queries in the integrated query plan. To increase progressiveness of a single query Q_i, we must fully generate large portions of the Q_i's intermediate tuples servicing another query Q_j in the workload. This defeats the chief objective of Contract-MQP.

Although existing *shared query plan* approaches have been effective for *select-project-join* (HRK$^+$09, KFHJ04, DSRS01) and aggregate (HHW97, HGHS07, WOT10) queries. These techniques rely on the queries being monotonic (GÖ05). That is, they produce an append-only result stream where processing a new input tuple never incurs deletion of a previously generated result tuple. Unfortunately skyline over join queries do not exhibit this convenient property. Instead, a newly generated join result can potentially dominate several previously generated join results – making them invalid. Moreover, these techniques assume all queries to have equal importance, thereby ignoring that some queries may have highly diverging and possibly conflicting QoS requirements.

1.4.4 Proximity-Driven Cardinality Assurance

As motivated in Chapter 1.2.4 to provide a better user experience databases must offer the capability of automatically yet appropriately refine the original query. The database can leverage information about the underlying data, quickly evaluate alternate refined queries and recommend the best set of queries to the user. The user can then choose the most appropriate query based on domain knowledge. For automatic query refinement to be meaningful, it must satisfy two criteria: (1) **Proximity**: the refined queries must minimize changes to the original query because the user will prefer queries that preserve semantics and differ as little as possible from their original intent. (2) **Cardinality Assurance**: the refined queries must minimize the difference between the actual and expected cardinality. The approach must be efficient and require minimal user effort. We refer

to the above problem of automatic query refinement as **Proximity-driven Cardinality Assurance (PCA)**.

PCA is a challenging problem for several reasons. First, the sub-problem of ensuring query cardinality is in itself NP-Hard (BCT06). Second, the amount of refinement needed to meet the cardinality constraint is difficult to predict apriori since it closely depends on factors such as data distribution and implicit correlations between attributes. Third, the number of possible refined queries is exponential in the number of predicates, making an exhaustive search impractical. Moreover, executing each refined query independently to calculate its cardinality is prohibitively resource-intensive.

1.5 Proposed Solutions

In this dissertation, we thoroughly investigate novel techniques to address the challenges listed in Chapter 1.4 in context of skyline-aware query evaluation. To demonstrate the generality of the solutions proposed in this work, we apply the foundational principle of this dissertation to an orthogonal research question – enabling users to acquire the desired information from a complex database via recommendation queries. The main contributions of this dissertation work include the following:

1.5.1 Skyline and Mapping Aware Query Evaluation

As the first objective in this dissertation, we provide key DBMS technologies that enable the skyline operation to be treated as a first-class citizen of query processing, namely: (1) extending the canonical relational operators to include operators that are skyline and mapping aware, (2) providing equivalence rules that transform the traditional query plans into plans that incorporate these skyline sensitive operators, (3) proposing query processing techniques tuned to exploit both skyline and mapping knowledge.

As a foundation, we propose new relational operators, namely *SkyMap*, *SkyJoin*, and *SkyMapJoin* to exploit the properties of the skyline operation. Accompanying this, we provide a set of equivalence rules to transform the traditional plans that involve skyline operations into plans that incorporate the new skyline and/or mapping aware operators.

In this work, we design a query execution framework *SKIN* (SKyline INside Join) that is able to: (1) leverage the skyline knowledge at various steps of query evaluation, (2) minimize the number of join results generated, (3) also reduce the number of skyline comparisons over this reduced set of join results, and (4) exhibit competitive performance for all distributions, including the skyline-unfriendly anti-correlated data sets. The **optimization-mantra** employed in *SKIN* is to "*avoid joining objects that will not result in a skyline result and avoid evaluating skylines for objects that do not join.*"

Instead of performing the join evaluation at the individual object level, we form a higher-level abstraction of the multi-dimensional data space using grid partitioning. Thereafter, we exploit the insight that skyline and mapping operations can be performed at this coarser granularity instead of directly on individual objects in the data space. The abstract-level execution principle holds true in the output space thereby reducing the total number of skyline comparisons necessary for query evaluation. *SKIN* is shown in our experimental evaluation to achieve nearly 1-2 orders of magnitude reduction in the number of join results, in comparison to state-of-the-art (KLTV06, JMP$^+$10, SWLT08, BKS01) techniques. In addition, *SKIN* successfully reduces the total number of comparisons (in several cases) by orders of magnitude against state-of-the-art approaches. To provide a complete study of the efficient handling of skyline and mapping aware query evaluation, we also explore the principles of skyline *partial push-through* (see Chapter 4 for details).

This part of the dissertation work contributes to research in the efficient processing of skyline-over-join queries in the following ways:

1. As a foundation, we introduce new skyline-aware algebra operators namely, *SkyMap*,

SkyJoin and *SkyMapJoin*, to facilitate query re-writes (see Chapter 2).

2. We propose *SKIN* in Chapter 3 as a robust methodology to process SkyMapJoin queries. To our best knowledge, *SKIN* is the first algorithm to efficiently exploit the insight that the join, skyline, and mapping operations cannot only be performed at different levels of data abstraction, but also be simultaneously performed in both input and output spaces.

3. Existing state-of-the-art techniques (JEHH07, SWLT08) do not test their approach over anti-correlated data sets, rather they restrict themselves to only the skyline friendly distributions such as independent and correlated datasets. In Chapter 5, we provide a comprehensive experimental evaluation of the existing techniques over all distributions commonly used in skyline literature as a stress test (BKS01).

4. Our performance analysis demonstrates the superiority of our proposed *SKIN* approach for many cases (such as anti-correlated and some independent data sets) by 1-2 orders of magnitude faster over state-of-the-art methods (as shown in Chapter 4). For the skyline friendly data sets such as correlated data, *SKIN* has similar performance as state-of-the-art techniques (KLTV06, JMP$^+$10, JEHH07, SWLT08). In addition, we report that *SKIN* on an average produces 50% fewer join results in comparison to state-of-the-art techniques and requires 1-2 orders of magnitude fewer skyline comparisons to produce the final result.

1.5.2 Progressive Result Generation for MCDS Queries

Next we propose a progressive query evaluation framework **ProgXe**. By transforming the execution of queries involving skyline-over-join to be *non-blocking*, *ProgXe* is able to be progressively generating results early and often. *ProgXe*, by exploiting the underlying

principle of *SKIN* – by performing query processing (join, mapping and skyline) at multiple levels of data abstraction, establishes the interaction between the input and output space to progressively output result early and often. This knowledge enables us to identify and reason with 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 maximizes the rate at which results are reported by translating the optimization of tuple-level processing into a job-sequencing problem.

Our contributions in the area of progressive result generation are:

1. We design a pipelined execution framework *ProgXe* that represents the foundation of our *progressive result generation* approach for skyline queries.

2. 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.

3. 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.

4. Our experimental analysis of *ProgXe* demonstrates the superiority of our proposed techniques over state-of-the-art techniques across a wide variety of data sets.

1.5.3 Contract-Driven Processing of Multiple Multi-Criteria Decision Support Queries

As elaborated in Chapter 1.4, when processing a workload of concurrent skyline-over-join queries each augmented with its own QoS requirement, it is not sufficient to complete one query fully before turning our attention to the second query. To the best of our knowledge we are the first to address the challenging problem of *Contract-Driven Multi-Query Processing* (Contract-MQP) where for a given workload of skyline-over-join queries and their associated *QoS contracts* we develop an execution strategy that maximizes the overall satisfaction of the workload.

In this dissertation, to address the *Contract-MQP* problem we propose the **C***ontract-***A***ware* **Q***uery* **E***xecution* framework - **CAQE** (pronounced *cake*). *CAQE* takes as input a set of skyline-over-join queries ($S_{\mathbb{Q}}$) and the associated set of contracts ($S_{\mathbb{C}}$). The core principle exploited in this work is: "*different portions of the input contribute to different and often multiple queries.*"

In this effort, we first designed a uniform model to express a progressiveness contract, and a metric to continuously measure the degree to which the contracts are being met. By exploiting *SKIN*'s principles of query processing at different data abstractions we expose and then exploit opportunities for fine-grained sharing among complex queries. We employ a novel contract-aware progressiveness estimation model to maximize the overall satisfaction of the workload of queries. The adaptive execution strategy employed in *CAQE* continuously monitors the run-time satisfaction of the queries and takes corrective steps when necessary to maximally satisfy the contracts. Our experimental evaluation demonstrates the effectiveness of *CAQE* in meeting these contracts compared to state-of-the-art techniques. To the best of our knowledge, this is the first work to define the *Contract-MQP* problem that handles multiple skyline-over-join queries.

1. We developed a *uniform model* to express a rich set of *contracts* to represent progressiveness, and a metric to continuously measure the degree to which the contracts are met.

2. Design the *min-max-cuboid* shared query plan, that facilitates sharing among plans for skyline-over-join queries, is guaranteed to contain the minimal subset of subspaces needed while maximizing sharing of skyline computations.

3. We propose an intuitive *contract-driven optimization* that applies a cost benefit model to determine the order in which the output regions are to be processed; such that the satisfaction of the conflicting contracts of workload queries are maximized.

4. We build an adaptive *contract-aware execution* methodology that continuously monitors the run-time satisfaction of the queries and aggressively takes corrective steps to maximally meet contracts.

5. Our experimental study on *CAQE* over benchmark datasets demonstrates that *CAQE* consistently outperforms existing techniques. More specifically, in many cases *CAQE* is $>$ 2x better in satisfying query contracts while generating 20x fewer join results and conducting 17x fewer skyline comparisons.

1.5.4 Cardinality Assurance Via Proximity-driven Refinement

To demonstrate the generality of the solutions proposed in this work, we apply the foundational principle exploited in this dissertation of processing queries at different level of data abstraction to an orthogonal research question namely, *Proximity-driven Cardinality Assurance (PCA)*. PCA is an unexplored area of auto-refining queries to meet cardinality and proximity constraints, while concurrently providing seamless support for select and join refinement. In this dissertation, to address this open problem we propose **C**ardinality

Assurance via **P**roximity-driven **R**ef**I**nement (**CAPRI**), the first of its kind.

Given a conjunctive Select-Project-Join (SPJ) query Q, desired cardinality $\mathcal{C}$, cardinality threshold δ, and refinement threshold γ, *CAPRI* produces a set of refined queries meeting the dual constraints of cardinality and proximity. In *CAPRI* we adopt the strategy of*Expand and Explore* to iteratively $expand$ the original query and to $explore$ refined queries with respect to cardinality. The $expand$ phase ensures that refined queries produced satisfy the refinement threshold and that queries with smaller refinements are produced before those with large refinements. Thus, once *CAPRI* finds a query satisfying the cardinality constraint, it need not examine queries with larger refinements. The $explore$ phase on the other hand efficiently computes refined query cardinalities via a novel incremental execution algorithm and an efficient, predictive index structure. By performing query processing at multiple levels of abstraction, a core contribution of *SKIN*, our incremental execution algorithm can identify dependencies between refined queries so that for each query, *CAPRI* must only execute a small sub-query and then simply use our recursive model to combine results from previous queries. Our predictive index structure assists in this process by coarsely mapping potential results to queries and enabling *CAPRI* to materialize only those query results that: (1) are likely to satisfy the given refined query and (2) haven't been materialized before. Together, these two techniques perform extensive result sharing and guarantee that any query result is processed at most once, irrespective of how many queries contain it. The two approaches enable *CAPRI* to evaluate a large number of queries rapidly without performing redundant computations.

Our contributions in addressing the PCA problem can be summarized as follows:

1. We introduce and then formalize the *Proximity-driven Cardinality Assurance* problem. We prove PCA to be NP-hard via a simple reduction (Chapter 19).

2. We propose *CAPRI* to auto-generate refined queries that satisfy both proximity and

cardinality constraints. We present a novel strategy to build the refined query space and map results to it, minimize query refinement via proximity-driven exploration, and evaluate a large number of refined queries efficiently (Chapter 20).

3. We design an innovative *Incremental Query Execution* approach to exploit dependencies between refined queries and facilitate extensive result-sharing. Our recursive model is guaranteed to process a result tuple at most once throughout our search and execution process (Chapter 21).

4. We demonstrate in our experimental study of *CAPRI* on TPC-H benchmark data sets that *CAPRI* consistently outperforms existing techniques with performance gains of up to 2 orders of magnitude. In addition, the queries generated by *CAPRI* are on average 25% closer to the original query compared to current techniques (see Chapter 22 for details).

1.6 Dissertation Organization

We discuss in detail the four research topics of this dissertation, namely **skyline aware evaluation over disparate source**, **progressive query evaluation of MCDS queries**, **contract-driven processing of multiple MCDS queries** and **proximity-driven cardinality assurance**, in Part I (Chapters 2-5), Part II (Chapters 6-10), Part III (Chapters 11-18) and Part IV (19-24) respectively. The discussions of each of the four research topics include the problem formulation and analysis, description of the proposed solution, experimental evaluation, and lastly discussions of related work. Chapter 25 concludes this dissertation and Chapter 26 discusses possible future work.

Part I

Skyline and Mapping Aware Query Evaluation

2

Skyline Aware Relational Algebra

Traditional query processing techniques (KLTV06) view skyline evaluation as a complex and extremely expensive filter operation. Thus rather than handling skyline computations only as an afterthought, in this chapter propose skyline-aware operators that exploit the principle of pushing the skyline operations *through* or *inside* other algebra operators.

In this effort, we extend the canonical relational model by introducing skyline and mapping aware operators. This extended relational model provides the foundation for research in developing efficient methodologies for evaluating these skyline aware operators. To transform a traditional query plan that has skyline operations on top of Select-Project-Join operations into an equivalent query plan with skyline sensitive operators, we provide a set of equivalence rules. In Chapter 3 we design an efficient query processing strategy for each of the proposed skyline aware operators.

2.1 Preliminaries

In this section, we review the *preference model* (Kie02) and the *algebra model* used to represent an SMJ (Skyline-Map-Join) query such as $Q1$. Each d-dimensional object is de-

fined 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 $r_i[a_k]$. $Dom(a_k)$ is domain of the attribute a_k and $Dom(A) = Dom(a_1) \times \ldots \times Dom(a_d)$.

2.1.1 Mapping Functions and Map Operator

The *map operator* (μ) is defined based on a set of k mapping functions. For each input object r_i the mapping function f_j, in a set of k mapping functions $\mathcal{F}$, takes as input a set of distinct attributes $B_j \subseteq A$ and returns a newly computed attribute x_j. That is, $f_j : Dom(B_j) \rightarrow Dom(x_j)$ and $\mathcal{F} = \{f_1, f_2, \ldots f_k\}$.

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

2.1.2 Preference Model and Skyline Operator

For a d-dimensional data set R, we use a_k ($1 \leq k \leq d$) to represent each dimension and $\mathcal{D} = \{a_1, \ldots, a_d\}$ the set of all d dimensions, called the **full-space**. For a tuple $\tau_i \in R$, the value of the attribute a_k can be accessed as $\tau_i[a_k]$. Given a set of attributes $\mathcal{V} \subset \mathcal{D}$, the preference P over the set of objects R is defined as $P := (\mathcal{V}, \succ)$ where $\succ$ is a *strict partial order* on the domain of $\mathcal{V}$. Here, $\mathcal{V}$ is termed as **subspace**. Without loss of generality, we assume that $\forall a_k : \tau_i[a_k] \geq 0$, and that smaller values are preferred.

Definition 2.1 (Full Space Dominance) *For a set R of d- dimensional tuples, a tuple $\tau_i \in R$* **dominates** *tuple $\tau_j \in R$ (denoted as $\tau_i \prec \tau_j$), iff* $(\forall(a_k \in \mathcal{D})\ (\tau_i[a_k] \leq \tau_j[a_k]) \wedge \exists(a_l \in \mathcal{D})\ (\tau_i[a_l] < \tau_j[a_l]))$.

Definition 2.2 (Subspace Dominance) *For a set R of d- dimensional tuples, and a set of attribute dimensions $\mathcal{V} \subseteq \mathcal{D}$ tuple τ_i* **dominates** *by a tuple τ_j* **in subspace** *$\mathcal{V}$ iff* $(\forall a_k \in \mathcal{V}$

$(\tau_i[a_k] \leq \tau_j[a_k]) \wedge \exists a_l \in \mathcal{V}\ (\tau_i[a_l] < \tau_j[a_l]))$ *and is denoted as* $\tau_i \prec_{\mathcal{V}} \tau_j$.

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

2.2 Extended Algebra Model

To facilitate the pushing of the skyline operation into the map and/or the join, we introduce three new operators, namely *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 object $r_i \in R$ into a k-dimensional object r_i' defined by the set of attributes X, and then (2) generate the skyline of transformed objects by the preference $P = (E, \succ_P)$, where $E \subseteq X$.

SkyJoin ($\widehat{\bowtie}_{[\mathcal{C},P]}$) combines objects from its input data sets based on the conditions in $\mathcal{C}$ and returns a set of non-dominated combined-objects based on the preference P. If $\mathcal{C} = \phi$ then the operator returns the set of non-dominated Cartesian product results.

SkyMapJoin ($\widehat{\Psi}_{[\mathcal{C},\mathcal{F},X,P]}$) performs the following operations in order: (1) combine objects from the input data sets based on the conditions in $\mathcal{C}$, (2) apply the set of mapping functions $\mathcal{F}$ to transform each combined-object to generate a transformed combined-object with attributes X, and (3) generate the skyline of combined-objects by the preference $P = (E, \succ_P)$, where $E \subseteq X$.

2.3 Query Equivalence Rules

In this work, we provide the foundation of alternate *SkyMapJoin* query plan solutions by describing the algebra and its equivalence rules. We now briefly review equivalence rules that enable us to push the skyline and mapping operation into the join operator.

The principle of pushing skylines through a join have been exploited in related work (JMP+10). However these existing techniques do not consider the multi-relational skyline operators with mapping operations (i.e., *SkyMapJoin* queries) which is the focus of our work.

To draw a parallel to *Select* in *Select-Project-Join* (SPJ) queries, there are scenarios when the mapping and skyline functionality can be *pushed inside* as well as sometimes *pushed-through* joins (BKS01, HK05). Below, we present equivalence rules to handle various mapping scenarios with Cartesian product (L1) as well as the join operations (L2). Rules L3-L7 are adaptations of the principle of *skyline push through* (HK05). Rules L7-L10 incorporate our proposed operators. Rules L10.a and L10.b aid in pushing partial skylines through the *SkyMapJoin* operator.

1. **L1**: Pushing $\mu_{[\mathcal{F},X]}$ through $\times$

   (a) $\mu_{[\mathcal{F},X]}(R \times T) \equiv \pi^X(\mu_{[\mathcal{F},X]}(R) \times T)$ iff $\forall(f_i \in \mathcal{F})(B_i \subseteq attr(R))$, where B_i as defined in Chapter 2.1.1 as the set of distinct attributes consumed by the function f_i.

   (b) $\mu_{[\mathcal{F},X]}(R \times T) \equiv \pi^X(\mu_{[\mathcal{F}_R,X_R]}(R) \times \mu_{[\mathcal{F}_T,X_T]}(T))$ iff

      i. $\mathcal{F} = \mathcal{F}_R \cup \mathcal{F}_T$, that is, $X = X_R \cup X_T$

      ii. $\forall(f_i \in \mathcal{F}_R)(B_i \subseteq attr(R))$

      iii. $\forall(f_i \in \mathcal{F}_T)(B_i \subseteq attr(T))$

   (c) $\mu_{[\mathcal{F},X]}(R \times T) \equiv \mu_{[\mathcal{F}_{RT},X]}(\mu_{[\mathcal{F}_R,X_R \cup attr(R)]}(R) \times \mu_{[\mathcal{F}_T,X_T \cup attr(T)]}(T))$ iff

      i. $\mathcal{F} = \mathcal{F}_R \cup \mathcal{F}_T \cup \mathcal{F}_{RT}$ i.e. $X = X_R \cup X_T \cup X_{RT}$

      ii. $\forall(f_i \in \mathcal{F}_R)(B_i \subseteq attr(R))$

      iii. $\forall(f_i \in \mathcal{F}_T)(B_i \subseteq attr(T))$

      iv. $\forall(f_i \in \mathcal{F}_{RT})(B_i \cap attr(R) \neq \phi) \wedge (B_i \cap attr(T) \neq \phi)$

2. **L2**: Pushing $\mu_{[\mathcal{F},X]}$ through $\bowtie$:

   Given a query plan $\mu_{[\mathcal{F},X]}(R \bowtie_{R.C=T.C} T)$ where $C = attr(R) \cap attr(T)$. If $\forall(f_i \in \mathcal{F})(B_i \cap C = \phi)$, then rules in $L1$ can be directly extended to handle joins.

3. **L3**: Pushing $\mathcal{S}_P$ through $\times$:

   (a) $\mathcal{S}_P(R \times T) \equiv \mathcal{S}_P(R) \times T$ iff $E \subseteq attr(R)$

   (b) $\mathcal{S}_P(R \times T) \equiv \mathcal{S}_P(\mathcal{S}_P(R) \times \mathcal{S}_P(T))$ iff $E \subseteq attr(R) = attr(T)$

4. **L4**: Pushing $\mathcal{S}_P$ through $\bowtie$: $\mathcal{S}_P(R \bowtie_{R.C=T.C} T) \equiv \mathcal{S}_P(\mathcal{S}_{P,GroupBy\ C}(R) \bowtie_{R.C=T.C} T)$ where $C = attr(R) \cap attr(T)$, $\mathcal{S}_{P,GroupBy\ C}(R)$ = $\{\ r_i \in R | \nexists(r_j \in R)$ s.t. $(r_j[C] = r_i[C]) \wedge (r_j \succ_P r_i)\}$

5. **L5:** Split and push $\mathcal{S}_P$ through $\times$:

   Let $P_R = (E_R, \succ_P)$ where $E_R \subseteq attr(R)$, $P_T = (E_T, \succ_P)$, $E_T \subseteq attr(T)$ and $P = \{P_R, P_T\}$ then $\mathcal{S}_P(R \times T) \equiv \mathcal{S}_{P_R}(R) \times \mathcal{S}_{P_T}(T)$

6. **L6:** Split and push $\mathcal{S}_P$ through $\bowtie$

   $\mathcal{S}_P(R \bowtie_{R.C=S.C} T) \equiv \mathcal{S}_P(\mathcal{S}_{[P_R,\ GroupBy\ C]}(R) \bowtie_{R.C=S.C} \mathcal{S}_{[P_T,\ GroupBy\ C]}(T))$ where, $\mathcal{S}_{[P_R,\ GroupBy\ C]}(R)$ = $\{\ r_i \in R\ | \nexists(r_j \in R)\ (\forall(a_k \in C)(r_j[a_k] = r_i[a_k])) \wedge (r_j \succ_{P_R} r_i)\ \}$

7. **L7:** Merging $\mathcal{S}_P$ with $\bowtie$: $\mathcal{S}_P(R \bowtie_{\mathcal{C}} T) \equiv R \widehat{\bowtie}_{[\mathcal{C},P]} T$

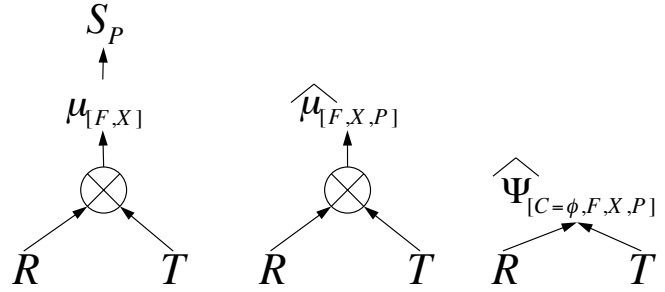
8. **L8:** Merging $\mathcal{S}_P$ with $\mu_{[\mathcal{F},X]}$: $\mathcal{S}_P(\mu_{[\mathcal{F},X]}(R)) \equiv \widehat{\mu}_{[\mathcal{F},X,P]}(R)$

9. **L9:** Merging $\widehat{\mu}_{[\mathcal{F},X,P]}$ with $\bowtie$: $\widehat{\mu}_{[\mathcal{F},X,P]}(R \bowtie_{\mathcal{C}} T) \equiv R\ \widehat{\Psi}_{[\mathcal{C},\mathcal{F},X,P]}\ T$

10. **L10:** Push down $\mathcal{S}_P$ in $\widehat{\Psi}$: Let $\mathcal{F}$ be a set of k monotonic increasing functions and $\mathcal{F}$ be proper. For each $f_i \in \mathcal{F}$, let $B_i^R \subseteq B_i$ s.t. $B_i^R \subseteq attr(R)$ and $B^R = B_1^R \bigcup \ldots \bigcup B_k^R$. Given $P = (E, \succ_P)$, let $P_R = (E_R, \succ_{P_R})$ where $E_R \subseteq E$ s.t. $E_R \subseteq B^R$. Additionally, $\mathcal{C} : R.C = T.C$ where $C \subseteq attr(R) \cap attr(T)$.

(a) $R\ \widehat{\Psi}_{[\mathcal{C},\mathcal{F},X,P]}\ T \equiv {}_{[\mathcal{C}]}\mathcal{G}_{[P_R]}(R)\ \widehat{\Psi}_{[\mathcal{C},\mathcal{F},X,P]}\ {}_{[\mathcal{C}]}\mathcal{G}_{[P_T]}(T)$

(b) $R\ \widehat{\Psi}_{[\mathcal{C}=\phi,\mathcal{F},X,P]}\ T \equiv (\mathcal{S}_{P_R}(R))\ \widehat{\Psi}_{[\mathcal{C}=\phi,\mathcal{F},X,P]}\ (\mathcal{S}_{P_T}(T))$



(a) Traditional Query Plan (b) Using ***SkyMap*** (c) Using ***SkyMapJoin***

P1 = min(tdistance); P2 = min(tcost)	*F = { f_1 ,f_2 ,f_3 ,f_4} f_1: :R.id, f_2 :T.id,*
P = {P1, P2}	*f_3: (2 * R.distance+T.distance) as tdistance,*
X = {R.id, S.id, tdistance ,tcost}	*f_4: (R.price+T.price) as tcost*

Figure 2.1: Multiple Equivalent Plans for *SkyMapJoin* query **Q2**

Example 2.1 *Figure 2.1.a depicts the traditional query plan for $Q2$ using canonical algebra operators where the results of the Cartesian product are fed to the map operator. For each Rome and Paris hotel pair, the map operator calculates $tCost$ and $tDistance$. The transformed combined-objects are then given as inputs to the skyline operator to generate the final result. Figures 2.1.b and 2.1.c represent the equivalent SkyMap- and SkyMapJoin-based query plans of $Q2$ generated by pushing first the skyline operation into the map operator, and then pushing both into the join. In other words, by applying the equivalence rules L8 and L9 listed above.*

3

SKIN: The Proposed Approach

In this chapter, we propose **SKIN** *(SKyline INside Join)* – an efficient methodology to evaluate SkyMapJoin queries. Figure 3.1 depicts the overall process of our approach called *SKIN*. For each input data source, we form an m-dimensional abstraction where m is the number of skyline dimensions in the combined (join) object. Below we elaborate on the core principles exploited in **SKIN** for evaluating the *SkyMapJoin* operators. The *SkyMap* operation is straightforward and thus omitted for conciseness. The *SkyJoin* operator can be viewed as a special case of the *SkyMapJoin* operator where each of the k mapping functions is a trivial projection.

The first phase in SKIN called **region-level elimination** targets to avoid the generation of combined objects altogether. Given a pair of input partitions from R and T, we determine: (1) whether or not the join operation between the objects in the input partition pairs will result in at least one combined-object (see details in Chapters 3.1–3.5), and (2) the region in the mapped output space into which the future combined-objects will fall during the actual object-level evaluation. Next, we identify output regions that are dominated by other regions. As we will show in Chapter 3.1 (Lemma 3.1), dominated regions are guaranteed to not contribute to the final skyline. Therefore, the join evaluation that gener-

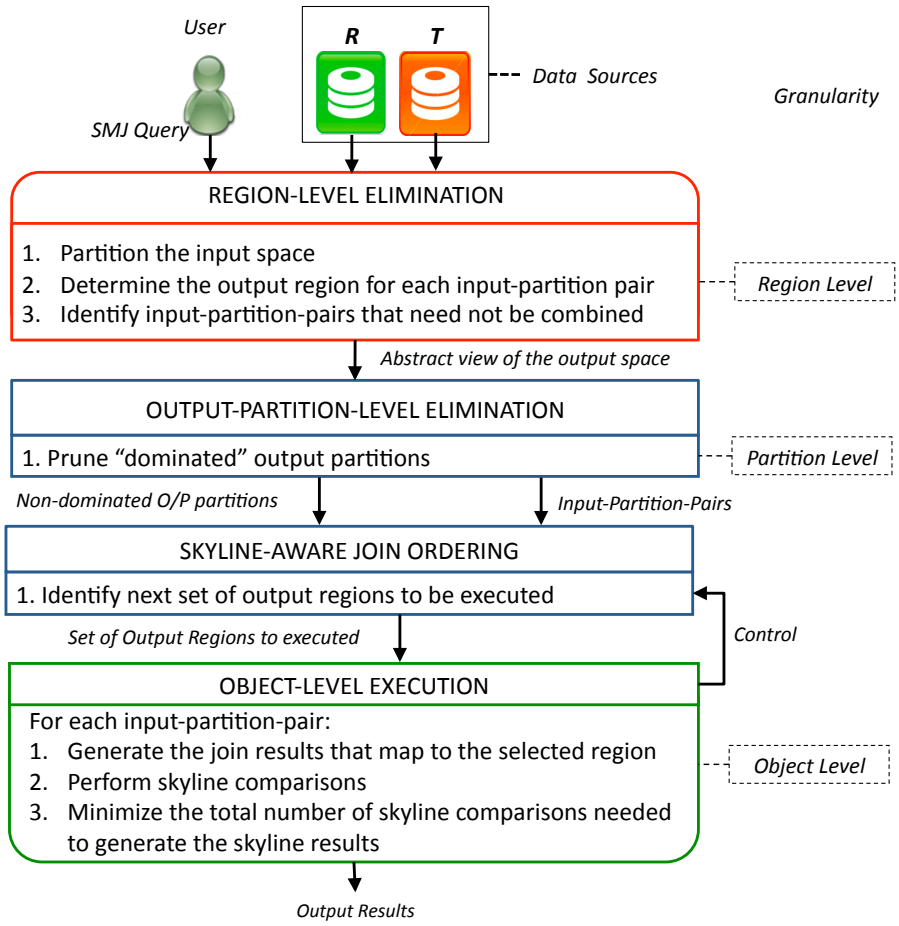


Figure 3.1: Overview of the **SKIN** Approach

ates objects that map to such dominated output regions need not be performed altogether, thereby saving on combined-object generation costs as well as dominance comparison costs (see Chapter 3.1).

Our second phase **partition-level elimination** aims to reduce dominance comparisons between combined-objects. Here, we perform query evaluation at the abstraction of partitions in the output space. In other words, we partition the output space such that each output region is composed of a set of output partitions. We observe that for some regions, a subset of their output partitions are dominated by other regions. Unlike in the above *region-level elimination* phase such partially dominated regions cannot be entirely discarded. However, we show by Lemma 3.2 that such dominated output partitions are guaranteed to not contribute to the final skyline. Therefore, combined-objects that map to these dominated output partitions can be immediately discarded without conducting any skyline comparisons (Chapter 3.2). In summary, *region-* and *partition-level elimination* phases eliminate many output regions and partitions respectively without any object-level access. Next, we sequence the execution of output regions to exploit the skyline characteristics of the output regions. In other words, we process output regions that are closer to the origin before those that can potentially be dominated by a future generated output tuple.

Finally, the fourth phase, called **object-level execution**, further reduces the total number of dominance comparisons needed to generate the final skyline. For each generated combined-object $r_f t_g$ we minimize the number of comparisons by: (1) eliminating all output partitions dominated by $r_f t_g$, and (2) restricting the object-level skyline comparisons to only a small subset of partitions containing combined-objects, namely those in output partitions that it can potentially dominate, and vice-versa (see Chapter 3.4). This third phase piggybacks on the former steps by reusing the partition information produced by them. We present the details; both underlying theory as well as concrete algorithms

Notations: • Each input partition in R is denoted as I_i^R • Each output partition is denoted as O_i • $\mathcal{I}^R$ is a set of all input partitions in R • $\mathcal{O}$ is a set of all output partitions • $[I_i^R, I_j^T]$ input-partition pairs whose combined-objects map to region $\mathcal{R}_{i,j}$ • $\mathfrak{R}$ is a set of all output regions called as *Region Collection*
• **LOWER(X):** Returns the *lower-bound* point of the region or partition X
• **UPPER(X):** Returns the *upper-bound* point of the region or partition X
• **MAP_OBJECT(r_ct_d, $\mathcal{F}$, δ):** Returns the output partition to which r_ct_d maps to.
• **MAP_REGION($\mathcal{R}_{i,j}$, $\mathcal{F}$, δ):** Returns a set of output partitions that $\mathcal{R}_{i,j}$ maps to.
• **MARK(O_i):** Marks the partition O_i as "*non-contributing*"
• **IS_MARKED(O_i):** Return *true* if output partition O_i is marked, *false* otherwise.

Table 3.1: Notations Used In This Work

for each of the four phases in Chapters 3.1, 3.2, 3.3 and 3.4 respectively.

3.1 Phase I: Region-Level Elimination

The first phase of our *SKIN* methodology, named **region-level optimization**, avoids the generation of many combined-objects. To easily highlight the core areas of optimization we first consider the motivating query $Q2$ where the join condition $\mathcal{C}$=ϕ. In Chapter 3.5 we extend the core approach to handle general join predicates as in query Q1.

We first partition each of the input data sets. For the remainder of this elaboration we employ an m-dimensional grid for partitioning the data space, with m being the number of skyline dimensions In our running example as in Figure 3.2.b the input data set T is partitioned into a 2-D grid. Each partition is uniquely identified by its bottom-left coordinates, the *lower bound* retrieved by the function LOWER(I_i^T). We define $\mathcal{I}^T$ as the **set of all non-empty input partitions** for T. In this example, $\mathcal{I}^T$=$\{I_1^T$[(1,5)(2,6)], I_2^T[(3,1)(4,2)], I_3^T[(5,0)(6,1)]$\}$.

In *region-level elimination*, for each non-empty partition, $I_i^R \in \mathcal{I}^R$ and each non-

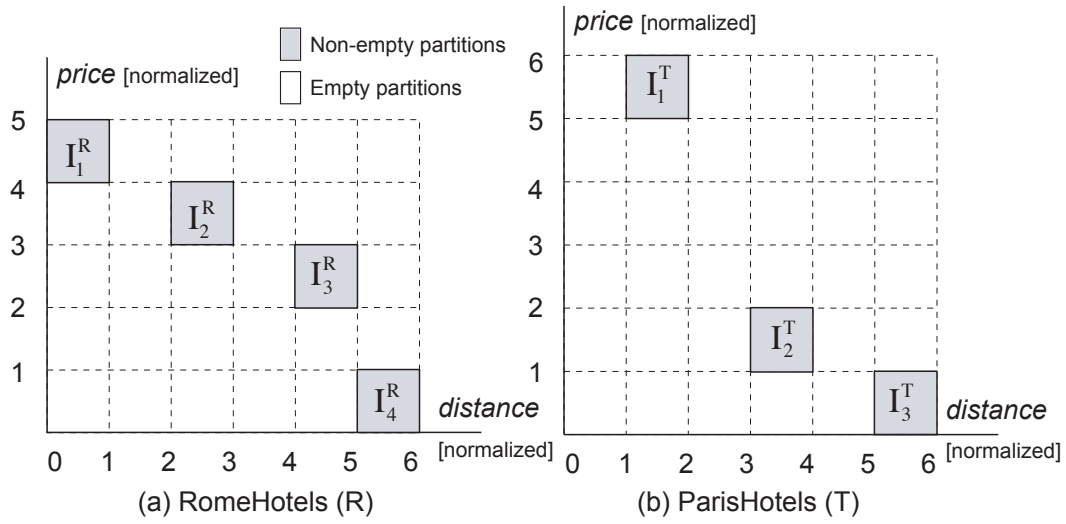


Figure 3.2: Partitioning Of Input Datasets

empty partition $I_j^T \in \mathcal{I}^T$, we determine the region of the output mapped space to which their join results will map to. This is achieved by applying the set of k mapping functions $\mathcal{F}$ to *lower-* and *upper-* bounds of the two input partitions respectively. The region corresponding to an input-partition-pair $[I_i^R, I_j^T]$ is called an **output region** (denoted as $\mathcal{R}_{i,j}$).

To elaborate, for query $Q2$ the mapping functions f_1 and f_2 compute two attributes, namely (f_1:$R.price+T.price$) computes *tprice* and ($2*R.distance+T.distance$:f_2) *tdistance*. The objects that map to the partition $I_1^R[(0,4)(1,5)]$ in Figure 3.2.a when combined with objects in partition $I_2^T[(3,1)(4,2)]$ in Figure 3.2.b result in the combined-objects that are guaranteed to fall into the region bounded by the points $b(3,5)$ and $B(6,7)$ in Figure 3.3.

We observe that: (1) the mapping and skyline functionality can now be applied at this higher level of abstraction, (2) output regions can always be determined *a priori* without any object-level data access, and (3) as long as both of the input partitions are non-empty, the corresponding output region is guaranteed to be populated. The set of all populated regions is called the **Region Collection** ($\mathfrak{R}$).

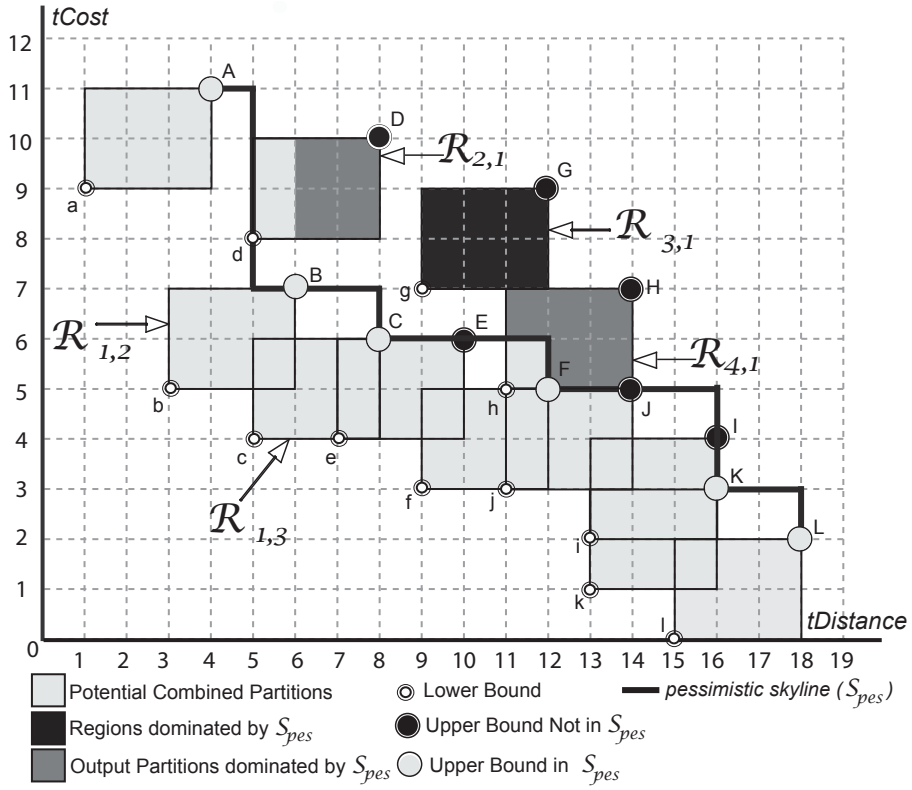


Figure 3.3: Pessimistic Skyline, S_{pes}

In query $Q2$ the preference is to minimize all skyline-dimensions. In a pessimistic scenario for each output region $\mathcal{R}_{i,j}$, all the combined-objects that map to it would lie on the upper-bound point of $\mathcal{R}_{i,j}$. We introduce the notion of the **pessimistic output skyline**, denoted as $\mathcal{S}_{pes}$ to identify the dominated output regions.

Definition 3.1 *For the preference P, the region collection $\mathfrak{R}$, $U = \{UPPER(\mathcal{R}_{i,j}) \mid \forall(\mathcal{R}_{i,j} \in \mathfrak{R})\}$, i.e., the set of upper-bounds of all output regions in $\mathfrak{R}$. Pessimistic skyline $\mathcal{S}_{pes}$ is defined as the skyline over U based on P, i.e., $\mathcal{S}_{pes} = \mathcal{S}_P(U)$.*

In Figure 3.3 the region $\mathcal{R}_{1,2}$ with the upper bound $B(6,7)$ clearly dominates $G(12,9)$, the upper bound of the region $\mathcal{R}_{3,1}$. Therefore H is not in the pessimistic skyline, $\mathcal{S}_{pes}$ = { A, B, C, E, F, K, L }.

Lemma 3.1 *For a region collection $\mathfrak{R}$, its pessimistic skyline S_{pes}, preference P and an output region $\mathcal{R}_{i,j} \in \mathfrak{R}$, if $\exists s \in S_{pes}$ such that $s \succ_P$ LOWER($\mathcal{R}_{i,j}$) then no combined-object $r_f t_g \in \mathcal{R}_{i,j}$ can be contained in the output skyline.*

Proof: Proof by contradiction. Assume that $\exists(r_f t_g \in \mathcal{R}_{i,j})$, $\exists s \in S_{pes}$ and $s \succ$ LOWER($\mathcal{R}_{i,j}$), but $r_f t_g$ in the output skyline. By Definition 3.1, $\exists\, \mathcal{R}_{x,y}$ s.t. $s =$UPPER($\mathcal{R}_{x,y}$). Since $\mathcal{R}_{x,y} \neq \phi$, let $r_b t_c \in \mathcal{R}_{x,y}$. Since $s \succ$ LOWER($\mathcal{R}_{i,j}$), $r_b t_c \succ r_f t_g$ and thus $r_f t_g$ is not in the output skyline. This is a contradiction. Therefore, the output region $\mathcal{R}_{i,j}$ will not contribute to the output skyline. ■

Rule 3.1 Region Elimination. *For a region collection $\mathfrak{R}$, its pessimistic skyline S_{pes}, and preference P and a region $\mathcal{R}_{i,j} \in \mathfrak{R}$, if $\exists s \in \mathcal{S}_{pes}$ s.t., $s \succ$ LOWER($\mathcal{R}_{i,j}$), then by Lemma 3.1, the objects in the input partitions I_i^R need never be combined with those in I_j^T since the resulting combined-objects are guaranteed to not be in the final skyline.*

For example, in Figure 3.3 the output region $\mathcal{R}_{3,1}$ with the lower-bound $g(9,7)$ is dominated by the pessimistic skyline points $B(6,7)$ as well as $C(8,6)$. By Lemma 3.1

the combined-objects generated from the input partition pair $[I_3^R, I_1^T]$ are guaranteed to not contribute to the output skyline, thus need not be generated. Algorithm 1 is the pseudo-code for the region-level elimination.

To summarize, the properties of region-level elimination are:

1. Only the regions that are guaranteed to be populated are considered for join evaluation.

2. Dominated regions are guaranteed to not contribute to the final results and therefore need not be considered for join evaluation.

Algorithm 1 Region-Level Elimination

Input: $\mathcal{F}$, P, $\mathcal{I}^R$, $\mathcal{I}^T$
Output: $\mathfrak{R}$ {Region Collection}
1: $\mathfrak{R} = \phi$; $U = \phi$
2: **for** each partition $I_i^R \in \mathcal{I}^R$ **do**
3: **for** each partition $I_j^T \in \mathcal{I}^T$ **do**
4: $\mathcal{R}_{i,j} \leftarrow$ Associated output region for $[I_i^R, I_j^T]$
5: Add UPPER($\mathcal{R}_{i,j}$) to U; Add $\mathcal{R}_{i,j}$ to $\mathfrak{R}$
6: $\mathcal{S}_{pes} = \mathcal{S}_P(U)$
7: **for** each output region $\mathcal{R}_{i,j} \in \mathfrak{R}$ **do**
8: **if** $\exists(s \in \mathcal{S}_{pes})$ $(s \succ_P$ LOWER$(\mathcal{R}_{i,j}))$ **then**
9: Remove $\mathcal{R}_{i,j}$ from $\mathfrak{R}$ {By Lemma 3.1, Rule 3.1}
10: **return** $\mathfrak{R}$

Time Complexity. The total number of input partitions for the input data sets R and T is denoted by n^R and n^T respectively. If $n^R = n^T = n$, then the time complexity to determine all n^2 output regions is $\mathcal{O}(n^2)$. The time complexity of generating the pessimistic skyline is in the worst case $\mathcal{O}(n^4)$ based on the Block-Nested-Loop (BNL) skyline algorithm (BKS01). Therefore, region-level elimination has the time complexity of $\mathcal{O}(n^2+n^4) \approx \mathcal{O}(n^4)$. This optimization is beneficial because it: (1) is at the granularity of output regions and does not require any object-level data access, (2) has the potential

to eliminate $n^2 - 1$ out of n^2 output regions in the best case scenario, and (3) has a significantly cheaper time complexity than $\mathcal{O}(N^4)$ for popular skyline algorithms (BKS01), where $|R|$=$|T|$=N. Since typically, $n << N$, $\mathcal{O}(n^4) << \mathcal{O}(N^4)$.

3.2 Phase II: Output-Partition Level Elimination

The **partition-level elimination** phase aims to reduce dominance comparisons needed to produce the final skyline. Each region $\mathcal{R}_{i,j}$ is mapped to one or possibly several output partitions. In Figure 3.3, $\mathcal{R}_{1,1}$ maps to the set of output partitions $\{O$[(1,9)(2,10)], O[(1,10)(2,11)], O[(2,9)(3,10)], O[(2,10)(3,11)], O[(3,9)(4,10)], O[(3,10)(4,11)]$\}$. Different regions may map to common output partitions. For example, $\mathcal{R}_{1,2}$ and $\mathcal{R}_{1,3}$ in Figure 3.3 share the output partition $O[(5,5)(6,6)]$. The *set of all output partitions* in the mapped output space is denoted as $\mathcal{O}$.

We observe that for some regions, only a subset of the output partitions they contain are dominated by the pessimistic skyline $\mathcal{S}_{pes}$. In Figure 3.3 for $\mathcal{R}_{2,1}$ the output partitions: O[(6,8)(7,9)], O[(6,9)(7,10)], O[(7,8)(8,9)] and O[(7.9)(7,10)] are dominated by point $B(6,7) \in \mathcal{S}_{pes}$. Prior to the actual generation of combined-objects that map to $\mathcal{R}_{2,1}$ we cannot easily determine which of the output partitions for $\mathcal{R}_{2,1}$ the combined-objects will fall into. For this reason, unfortunately we cannot entirely discard the partially dominated output region $\mathcal{R}_{2,2}$ as in Chapter 3.1. Therefore, we have to first generate the combined-objects that map to such partially dominated regions. As we show next, we can however exploit this fact of partially dominated output regions. The intuition here is that for a point B to be in $\mathcal{S}_{pes}$ there must exist an output region, here $\mathcal{R}_{1,2}$, such that UPPER($\mathcal{R}_{1,2}$) = B. All output regions are guaranteed to be populated i.e., $\mathcal{R}_{1,2} \neq \phi$ and $\exists r_b t_c \in \mathcal{R}_{1,2}$ such that $r_b t_c$ dominates a subset of output partitions mapped to $\mathcal{R}_{2,1}$. Therefore, combined-objects that map to output partitions: O[(6,8)(7,9)], O[(6,9)(7,10)], O[(7,8)(8,9)] and

O[(7.9)(7,10)] are guaranteed to not contribute to the final skyline.

Lemma 3.2 *Given a region collection $\mathfrak{R}$, its pessimistic skyline S_{pes}, preference P and an output partition $O_l \in \mathcal{O}$ s.t., $\exists(s \in \mathcal{S}_{pes})$ ($s \succ_P$ LOWER(O_l)), then no combined-object $r_f t_g \in O_l$ can be contained in the output skyline.*

We omit the proof for By Lemma 3.2 for conciseness. Combined objects that map to dominated output partitions during actual object-level evaluation can safely discard thereby avoid performing any skyline comparisons on such combined-objects. In addition, we mark all such dominated output partitions as **non-contributing**. Algorithm 2 is the pseudo-code for the partition-level elimination. For each region $\mathcal{R}_{i,j}$, it determines its corresponding partitions by the function MAP_REGION($\mathcal{R}_{i,j}, \mathcal{F}, \delta$) (Line: 2). For each output partition O_l we determine if it is dominated by a pessimistic skyline point (Line: 4-5). All dominated output partitions are marked as "non-contributing" by the function MARK(O_l) (Line: 6).

Algorithm 2 Output Partition-Level Elimination

Input: $\mathfrak{R}$ {Region Collection}, S_{pes} {Pessimistic Skyline}
Output: $\mathcal{O}$ {Set of output partitions}
1: **for** each output region $\mathcal{R}_{i,j} \in \mathfrak{R}$ **do**
2: **for** each o/p partition $O_l \in$ MAP_REGION($\mathcal{R}_{i,j}, \mathcal{F}, \delta$) **do**
3: Add O_l to $\mathcal{O}$
4: **for** each output partition $O_l \in \mathcal{O}$ **do**
5: **if** $\exists(s \in \mathcal{S}_{pes})$ ($s \succ$ LOWER(O_l)) **then** MARK(O_l)
6: **return** $\mathcal{O}$

Time Complexity. This phase eliminates the skyline comparisons for combined-objects that map to any dominated output partitions. In the worst case scenario, the region-level elimination is unsuccessful in eliminating any output region. Then the pessimistic skyline has n^2 points that correspond to the upper bounds of all n^2 output regions. The total number of output partitions is denoted as n_o. The time complexity of the partition-level elimination is $\mathcal{O}(n_o^2)$. This overhead is small because, (1) it is at the granularity of output

partitions and does not require any object-level data access, and (2) in a typical database $n_o^2 << N^2$.

3.3 Phase III: Skyline-Aware Join Ordering

In this section, we present an optimization technique that orders the execution of output regions to minimize the total execution time. This is accomplished by analyzing the dependencies among the output regions in the multi-dimensional abstract space. In this effort, we first introduce the concept of an optimistic skyline. Without loss of generality, we assume that the preference model is to minimize all skyline-dimensions. Thus in the optimistic scenario for each output region $\mathcal{R}_{i,j}$ all future join results would lie on (or near) the lower bound point of $\mathcal{R}_{i,j}$.

Definition 3.2 *For the region collection* $\mathfrak{R}$, $L = \{LOWER(\mathcal{R}_{i,j}) \mid \forall(\mathcal{R}_{i,j} \in \mathfrak{R})\}$, *i.e., the set of lower-bounds of all output regions in* $\mathfrak{R}$. *The* **optimistic skyline** $\mathcal{S}_{opt}$ *is defined as the skyline over* L *based on P, i.e.,* $\mathcal{S}_{opt} = \mathcal{S}_P(L)$.

The main intuition here is to evaluate all output regions that intersect the optimistic skyline. Output regions on this optimistic skyline have a greater chance of dominating those region(s) not on the optimistic skyline, thereby potentially avoiding their join evaluation. Once all the output regions on the optimistic skyline have been processed, if any non-dominated output regions still exist, we repeat the above technique. In other words, we evaluate output regions on the updated optimistic skyline generated over the remaining non-dominated output regions.

In Figure 3.4, point $c(5, 4)$ is the lower-bound corners of the output region $\mathcal{R}_{1,3}$ and dominates $e(7, 4)$, the lower-bound corner of region $\mathcal{R}_{2,2}$. Therefore, $c \in \mathcal{S}_{opt}$, while $e \notin \mathcal{S}_{opt}$. We define the optimistic skyline as the skyline of all the lower-bound corners.

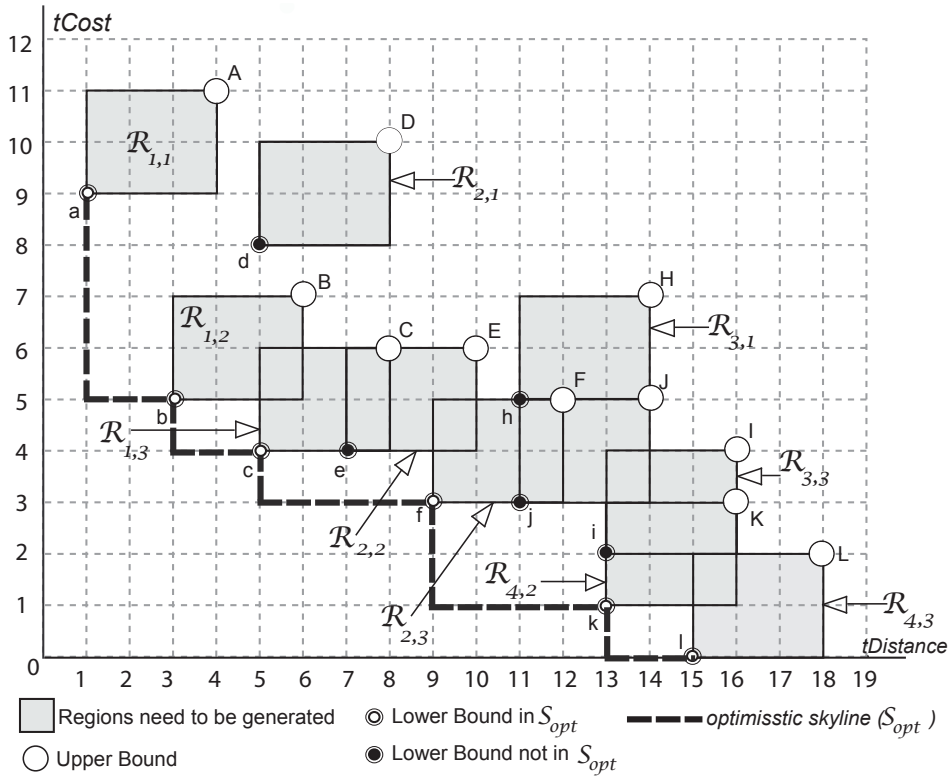


Figure 3.4: Generating Optimistic Skyline $\mathcal{S}_{opt}$

In Figure 3.3, S_{opt} = { a, b, c, f, k, l}. Current state-of-the-art skyline algorithms (BKS01, CGGL03, KS00, BCP06) can be utilized to find this optimistic skyline.

Next, we consider for join and skyline evaluation the subset of output regions whose lower bounds are on the optimistic skyline namely, $\mathcal{S}_1^O$. In Figure 3.4, this is found to be { $\mathcal{R}_{1,1}$, $\mathcal{R}_{1,2}$, $\mathcal{R}_{1,3}$, $\mathcal{R}_{2,3}$, $\mathcal{R}_{4,2}$, $\mathcal{R}_{4,3}$}. We can be obtained by the function $OPT\_REGION(\mathfrak{R})$. Then we pipeline the execution of all regions in $\mathcal{S}_1^O$. Next, we investigate the remaining non-dominated output region that have not yet been considered for tuple-level processing to determine the output regions interesting the optimistic skyline. We denote this set as $\mathcal{S}_2^O$. Thus optimistic skyline-based ordering is an ordered subset of output regions in the mapped output space, $\mathfrak{R}$, denoted as $\mathcal{S}^O$ = ($\mathcal{S}_1^O$, ... $\mathcal{S}_n^O$), where $\mathcal{S}_i^O$ is the i^{th} subset of output regions to be considered for object-level join evaluation.

Definition 3.3 *Given the region collection $\mathfrak{R}$, the optimistic skyline-aware join order is*

defined as a set $\mathcal{S}^O = \{\mathcal{S}_1^O, \mathcal{S}_2^O, \ldots \mathcal{S}_n^O\}$, *where* $\mathcal{S}_i^O = OPT\_REGION(\ \mathfrak{R} \setminus \bigcup_{j=1}^{i-1} \mathcal{S}_j^O)$.

To illustrate, in Figure 3.4, $\mathcal{S}_1^O$ = { $\mathcal{R}_{1,1}$, $\mathcal{R}_{1,2}$, $\mathcal{R}_{1,3}$, $\mathcal{R}_{2,3}$, $\mathcal{R}_{4,2}$, $\mathcal{R}_{4,3}$} is the first subset of regions to be considered for the object-level execution phase of *SKIN*. Once all regions in $\mathcal{S}_1^O$ have been processed we then move to processing the next subset of regions, $\mathcal{S}_2^O$ = $\{\mathcal{R}_{2,1}$, $\mathcal{R}_{2,2}$, $\mathcal{R}_{3,2}$, $\mathcal{R}_{3,3}$ }. Finally, $\mathcal{S}_3^O$ =$\{\mathcal{R}_{4,1}\}$ is considered for object-level execution.

Algorithm 3 Skyline-Aware Join Ordering

Input: $\mathcal{X}$ {Set of regions yet to be considered for object-level execution}
Output: $\mathcal{S}_{curr}^O$ {Skyline-Aware Join Order}

```
1:  Initialize S^O_curr=φ; Remainder = φ
2:  for each partition R_{i,j} ∈ X do
3:     if S^O_curr=φ then
4:        Add R_{i,j} to S^O_curr
5:     for each R_{x,y} ∈ S^O_index do
6:        if LOWER(R_{x,y}) ≻ LOWER(R_{i,j}) then
7:           Add R_{i,j} to Remainder; Goto 2;
8:        else
9:           if LOWER(R_{i,j}) ≻ LOWER(R_{x,y}) then
10:             Remove R_{x,y} from S^O_curr
11:             Add R_{x,y} to Remainder
12:    Add R_{i,j} to S^O_curr
13: X ← Remainder
14: return S^O_curr
```

Next, we present Algorithm 3 for finding the optimistic skyline-based ordering $\mathcal{S}_{curr}^O$. For each output $\mathcal{R}_{i,j}$, we first identify if it belongs to $\mathcal{S}_{curr}^O$ (starting with $curr = 1$) by comparing $\mathcal{R}_{i,j}$ against all the non-dominated regions yet to be considered for object-level execution (Line: 2–4). If there exists an $\mathcal{R}_{i,j} \in \mathcal{S}_{curr}^O$ such that the lower-bound corner of $\mathcal{R}_{x,y}$ dominates the corresponding lower-bound of $\mathcal{R}_{i,j}$, then $\mathcal{R}_{i,j}$ does not fall on the current optimistic skyline $\mathcal{S}_{curr}^O$ (Line: 9-11).

Time Complexity: For n output regions, in the best case scenario the skyline-aware join ordering is triggered only once. Hence the time complexity of $\mathcal{O}(n^2)$. In the worst case scenario $|\mathcal{S}^O| = n$, that is, in each iteration we have only one region to be sent for tuple-

level processing. In such a scenario, the skyline-aware join ordering needs to be triggered n times. Therefore the ordering step of SKIN has the time complexity of $\approx \mathcal{O}(n^3)$.

3.4 Phase IV: Object-Level Execution

We now introduce the processing logic for the actual generation of combined-objects and skyline evaluation assuming the above pruning steps have been carried out.

For each remaining pair of input partitions $[I_i^R, I_j^T]$ the object-level execution involves three steps. First, each object r_f in partition I_i^R is combined with each object t_g in partition I_j^T to generate the combined object $r_f t_g$. Second, the newly generated combined object $r_f t_g$ is then mapped to its corresponding output partition O_h. Finally, $r_f t_g$ is compared against other existing combined-objects to generate the output skyline.

We optimize the skyline comparison step as follows: first, if the newly generated combined object $r_f t_g$ that maps to an output partition that is marked as "non-contributing" can be safely discard without further processing. If $r_f t_g$ maps to an output partition is not marked "non-contributing" we cannot discard $r_f t_g$ without having to perform skyline comparisons. Second, in such scenarios we next mark all partitions dominated by $r_f t_g$ as being "non-contributing" by following the principle stated in Lemma 3.2. Please note that this sub-task of marking dominated partitions is done only for the first combined object that falls in partitions O_h.

In Figure 3.5, the combined-object $r_f t_g \in O_h$ clearly dominates the output partitions in the top-right corner, i.e., every $O_q \in \mathcal{O}$ where $r_f t_g \succ_P$ LOWER(O_q) holds. Third, we minimize the total number of combined-object comparisons during skyline computation. We exploit the knowledge gained from the output space by ordering the sequence in which the input partition pairs $[I_i^R, I_j^T]$ are considered for object-level execution based on its nearness to origin given that we assume that the preference is to minimize across all

dimensions.

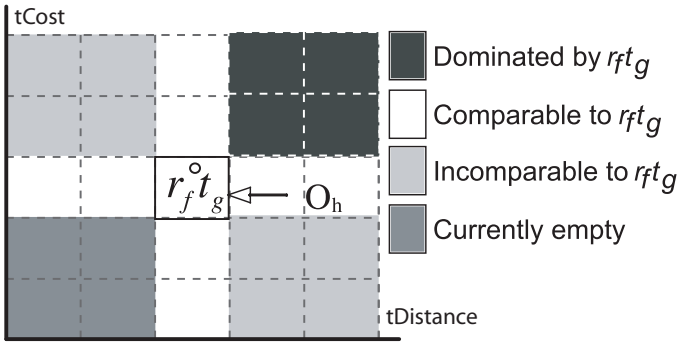


Figure 3.5: Object-Level Comparison Criteria

In Figure 3.5 we observe that (1) combined-objects that map to output partitions in the top-left 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 guaranteed to be empty, else O_h would have been be marked as "non-contributing" in an earlier iteration. (3) $r_f t_g \in O_h$ can be only dominated by combined objects that map to the slice of partitions that either have the same $price$ or $distance$ as O_h.

Rule 3.2 Comparable Partitions. *Given a newly generated combined-object $r_f t_g$, let MAP_OBJECT($r_f t_g, \mathcal{F}, \delta$) = O_h, and let A = LOWER(O_h). Then, $r_f t_g$ needs to only be compared against combined-objects in output partition $O_q \in \mathcal{O}$ with B = LOWER(O_q), where $\exists(1 \leq v \leq m)(A[l_v] = B[l_v])$.*

Algorithm 4 lists the pseudo-code for the object-level execution. For each generated combined-object $r_f t_g$, we first identify the partition O_h to which it maps by the function MAP_OBJECT (Line: 4). If O_h is marked "non-contributing" we immediately discard $r_f t_g$ without any further processing (Line: 5-6). Otherwise, we mark all partitions $O_q \in \mathcal{O}$ which are dominated by O_h as "non-contributing" Line (9-11). Next, we begin the comparisons of $r_f t_g$ by first comparing it against other existing combined-objects that are mapped to the same O_h (Line: 12). If $r_f t_g$ is dominated in O_h we stop and move to the

Algorithm 4 Object-Level Execution

Input: $\mathfrak{I}^R$, $\mathfrak{I}^T$, $\mathcal{F}$, $\mathfrak{R}$, P
Output: Set of non-dominated combined-objects
1: **for** each output region $\mathcal{R}_{i,j} \in \mathfrak{R}$ **do**
2: **for** each object $r_f \in I_i^R$ and $t_g \in I_j^T$ **do**
3: Generate combined object $r_f t_g$
4: $O_h \leftarrow$ MAP_OBJECT($r_f t_g$, $\mathcal{F}$, δ)
5: **if** IS_MARKED(O_h) **then**
6: discard $r_f t_g$
7: **else**
8: **for** each $O_q \in \mathcal{O}$ **do**
9: **if** $r_f t_g \succ_P$LOWER(O_q) **then** MARK(O_q)
10: Call UpdatePartition(O_h, $r_f t_g$, P)
11: **for** each O_q satisfying Rule 3.2 AND $r_f t_g$ not dominated **do**
12: Call UpdatePartition(O_q, $r_f t_g$, P)
13: Add $r_f t_g$ to O_h if $r_f t_g$ is still not dominated
14: **return** all combined objects mapped to all output partitions
15: **procedure** UpdatePartition(O_q, $r_f t_g$, P)
16: **for** each $r_x t_y \in O_q$ **do**
17: **if** $r_x t_y \succ_P r_f t_g$ **then** discard $r_f t_g$
18: **else if** $r_f t_g \succ_P r_x t_y$ **then** remove $r_x t_y$ from O_q

generation of the next combined-object. Otherwise, we compare $r_f t_g$ against combined-objects that are mapped to "*comparable partition*" as defined by Rule 3.2 (Line 13-14). After all these comparisons if $r_f t_g$ remains not dominated then and only then we insert $r_f t_g$ into O_h.

Optimization Benefits. Let each dimension in the output space be partitioned into k partitions. Then the d-dimensional grid has k^d output partitions. For any skyline algorithm in the worst case scenario all objects are in the final skyline. Therefore, for a naïve approach in the worst case scenario, each newly generated combined-object will be compared against objects in all k^d partitions. Instead, in our approach for each newly generated combined-object, in the worst case, we only perform dominance comparisons against objects that are mapped to a smaller set of $[k^d - (k-1)^d]$ partitions.

3.5 Handling Join Predicates

We now illustrate the full solution of our approach, namely we now to also handle join predicates in queries such as $Q1$ (in Figure 1.4) where suppliers and transporters have to be from the same country. Unlike the scenario in Chapter 3.1 where we utilize all input partition pairs, we now only need to consider those input partition pairs that are guaranteed to have at least one tuple each with matching attributes values. That is, they indeed join and populate the corresponding region. Clearly, otherwise no input objects will find matching join partners, and thus the region will be empty. Intuitively, for this determination we perform join evaluation at a higher-level abstraction instead of join over actual objects. In our example as in Figure 3.6, the input partition I_2^R shares with input partition I_2^T the domain values {Brazil, China, Mexico}, while both input partitions I_1^T and I_3^T share the attribute value {China}. Therefore, objects in I_2^R are guaranteed to find at least one join pair in each of the partitions I_1^T, I_2^R and I_3^T. Conversely, in Figure 3.6, we do not consider the pairs [I_3^R,I_1^T], [I_3^R,I_2^T] and [I_3^R,I_3^T], since there is no partition in T that shares any domain value with I_3^R, in this case *Indonesia*.

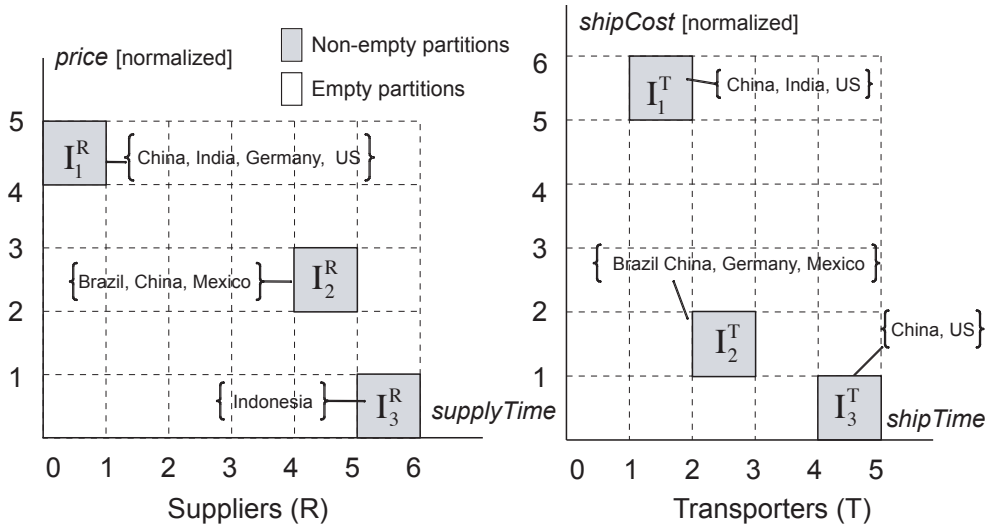


Figure 3.6: Partitioning For Suppliers (R) and Transporters (T)

To determine if an input-partition shares at least a single domain value, we maintain a signature for each partition to capture the domain values of its member objects. This signature could either be a Bloom Filter, bit vector, etc. In Figure 3.6 for each partition in R (or T) we maintain a list of countries to which the suppliers (or transporters) belong to, e.g., I_3^T has suppliers from *China* and *US*.

For finite domains, to efficiently maintain the occurrence of join values within a partition we use a bit vector with size equal to the cardinality of the domain of the join attribute. In the motivating example as in Figure 3.6, $Dom(country)$= {Brazil, China, Germany, India, Indonesia, Mexico, USA}, and input partition I_1^R has a bit vector $B_1^R(0111001)$. For our bit vector method, to determine if the input partition pair [I_i^R, I_j^T] will generate even a single join result, a simple bit-wise AND operation between B_i^R and B_j^T is sufficient. If the resulting bit vector $B_i^R \wedge B_j^T$ is greater than 0, the region $\mathcal{R}_{i,j}$ is guaranteed to be populated. Otherwise, the output region $\mathcal{R}_{i,j}$ is guaranteed to be empty. Then we simply do not perform computation on such empty region and move on to the next input partition pair. In our running example, $B_1^R(0111001) \wedge B_1^T(0101001)$=$(0101001) > 0$ and therefore we generate region $\mathcal{R}_{1,1}$. Conversely, $B_3^R(0000100) \wedge B_1^T(0101001)$ = (0000000) and we do not generate region $\mathcal{R}_{3,1}$.

Region- and Output Partition-Level Elimination:. When an output region or output partition is populated by at least one member, we proceed with the techniques described in Chapters 3.1-3.2 without any modification. This is because both output region- and output partition-level reasoning are independent of domain values and only concern themselves itself exclusively with dominance properties of output regions or partitions that are known to contain some join pairs.

Object-Level Execution. We only consider output regions that are guaranteed to be populated and generated in the previous steps. Next, we generate combined objects that satisfy the join condition. The strategy presented in Chapter 3.4 can thus be used as is to

generate the final skyline.

For very large domain values when the bit-vector based join evaluation may become rather expensive. We propose two alternative solution strategies that can applied as explained next. One, we could interleave region elimination during tuple level processing and, two, we could deploy sampling. In the case of interleaving we start by assuming that we have no knowledge of any region being guaranteed to be populated. Then, we iteratively trigger the region and partition-level elimination when a previously empty output region or output partition becomes populated. Alternatively, for the sampling based approach we sample each input data source (table) in the given SMJ join query. Based on the obtained sample set of input tuples (from both tables in our running example) we generate join tuples. These join tuples are then mapped and further used to prune dominated output regions.

SKIN handles both numeric and non-numeric join attributes as illustrated in our motivating example in Figure 3.6 where the join condition is on *country* – a non-numeric attribute. Since we use a bit-vector to encode join attribute values, both numeric and non-numeric domain values can be supported.

4

Experimental Evaluation of SKIN

In this section, we verify the effectiveness and efficiency of our proposed SKIN approach to handle skyline queries over disparate sources.

4.1 Experimental Setup

4.1.1 Proposed Techniques

The principle of skyline *partial push-through* (BKS01, HK05) is complimentary to the core approach presented in Chapter 3. Here, we incorporate this principle by pruning each individual data source by first applying Algorithm 5 on them separately.

Algorithm 5 Skyline Partial Push-Through

Input: Input Set R; Skyline dimensions $\{a_1, \ldots a_d\}$; Join attribute a_{d+1}
Output: Set of non-dominated objects with-in the same join attribute value a_{d+1}
1: Group objects in input set R into groups $\mathcal{G}_i$ by the join attribute a_{d+1}
2: **for** each group $\mathcal{G}_i$ **do**
3: LR_i = Generate local skyline on attributes $\{a_1, \ldots a_d\}$
4: **return** $LR_o \cup LR_1 \cup \ldots LR_k$

4.1.2 Competitor Techniques

In this dissertation we compare the performance of our proposed technique with the following state-of-the-art techniques. First, *JFSL* with the improvement of incrementally maintaining the skyline of join results and using a hash-based join implementation (KLTV06). Second, an optimized *JFSL*$^+$ that uses the principle of skyline partial push-through to prune each individual data source. Third, *Skyline-Sort-Merge-Join* (*SSMJ*) technique proposed by (JEHH07). *SSMJ* maintains for each data source two active lists of objects: (1) those objects that are in the set-level skyline generated by ignoring the join condition, and (2) the objects that are in the group-level skyline for each join attribute value (as in Algorithm 5). Next, these lists are given for join evaluation, set-level lists first followed by group-level lists. The skyline is then computed over the join results to return the final query results. Fourth, we deploy *SAJ* (KLTV06) which extended the popular Fagin technique (FLN01) following the JFSL paradigm.

4.1.3 Experimental Platform

All experiments are conducted on a Linux machine with AMD 2.6GHz Dual Core CPUs and 4GB memory. All algorithms are implemented in Java 1.5.0_16.

4.1.4 Evaluation Metrics

For each algorithm we measure: (1) the total execution time, (2) the total number of intermediate combined-objects generated, and (3) the total number of domination comparisons required to generate the final skyline. In addition, we measure the time taken by each phase of our approach.

4.1.5 Stress Test Data

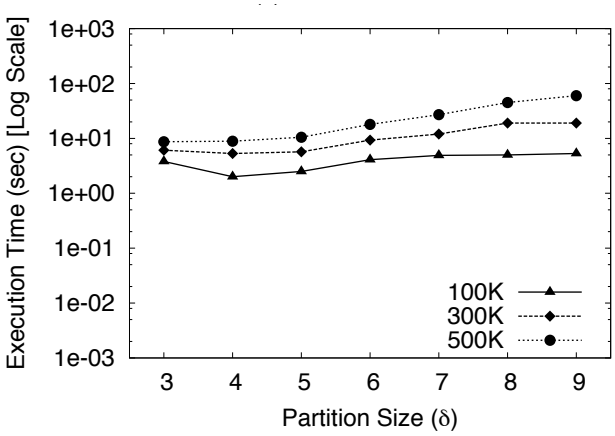
We have conducted our experiments using data sets that are the *de-facto* standard for stress testing skyline algorithms in the literature (BKS01). The data sets contain three extreme attribute correlations, namely *independent*, *correlated*, or *anti-correlated*. For each data set R (and T), we vary 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 in Chapter 1.2. In Chapter 4.3.4 we analyze the performances of the different techniques by varying the mapping function applied over the join results. We set $|R| = |T| = N$.

Real Data Sets. In addition, we also conducted our experiments using two real data sets, namely *NASDAQ* data[1] and *Household* data[2]. The *NASDAQ* data set contains 219K 6-dimensional tuples, where each tuple represents the information about daily trading of NASDAQ stocks. The attributes for each tuple comprises of open price, high price, low price, close price, volume, and adjusted close price. The *household* data set contains 127k 6-dimensional tuples, where each tuple represents the amount spent annually by an American family on gas, electricity, water, heating, insurance, and property tax.

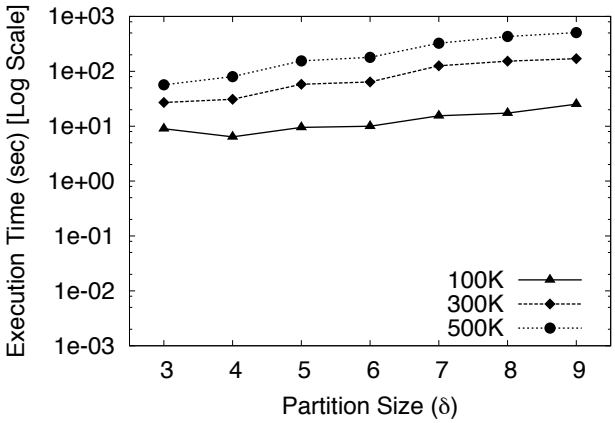
4.2 Experimental Analysis of SKIN

Purpose. We first study the robustness of our approach by varying: (1) partition sizes δ, (2) data distributions, (3) cardinality N , and (4) dimensions d. For a dimension d, data distribution and cardinality N, we measure the execution time for each partition size δ.

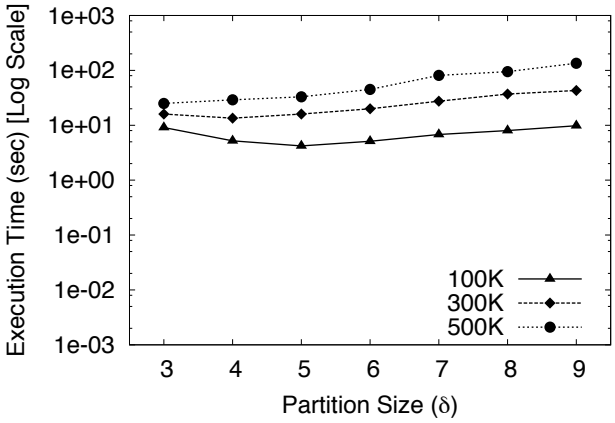
[1] available at http://davis.wpi.edu/xmdv/datasets/nasdaqg.html
[2] available at www.ipums.org



(a) Correlated Distribution



(b) Independent Distribution



(c) Anti-Correlated Distribution

Figure 4.1: Effects of Partition Size (δ) On SKIN's Performance ($d = 2$; $\sigma = 0.1$)

Partition Size (δ). The effectiveness of SKIN depends on the ratio between the object-level vs. the partition-level granularity. (DTB09, TB10) proposed a tool to automate the task of identifying good settings for database configuration parameters. In this work, we follow the principle proposed in (TB10) to test our proposed system based on a set of training work loads generated by the de-facto standard for testing skyline algorithms (BKS01). For our training data set, for each d, we generated data sets with several key distributions (correlated, anti-correlated and independent) of sizes 100K, 300K, and 500K. That is, for each d we had 9 training data sets. Figures 4.1.a–c show the execution times of SKIN for dimension d=2, and for the data sizes N= 100K, 300K and 500K. Smaller partition sizes δ will result in many sparsely populated input partitions, and therefore the overhead costs of region-level elimination will out-weigh its benefits. Alternatively, as δ is increased the execution costs of region-level eliminations are reduced. For large δ however may only marginally reduce the number of combined objects generated and thus would increase the number of combined objects to be compared against the output space. This insight is confirmed in Figures 4.1.a–c for $\delta \geq 6$ for all distributions. In Figures 4.1.a–c and for all N, we observe that $\delta = 4$ for all three distributions has the smallest execution costs compared to the other partition sizes. Similarly, for the dimension d=3 and then d=4 we observe similar trends for picking δ =12 or δ=17 respectively for all distributions. Based on these encouraging findings of stability across distributions, an optimizer could choose a delta for given a data set, distribution and number of dimensions in its setup time.

Execution Time Analysis of SKIN Steps. In Figure 4.2 we present the CPU processing time incurred by the four steps in our proposed *SKIN* methodology.

Cardinality (N). In Figure 4.2 for any given data distribution, we observe that the execution time for *output region elimination*, *output partition elimination* and *skyline aware*

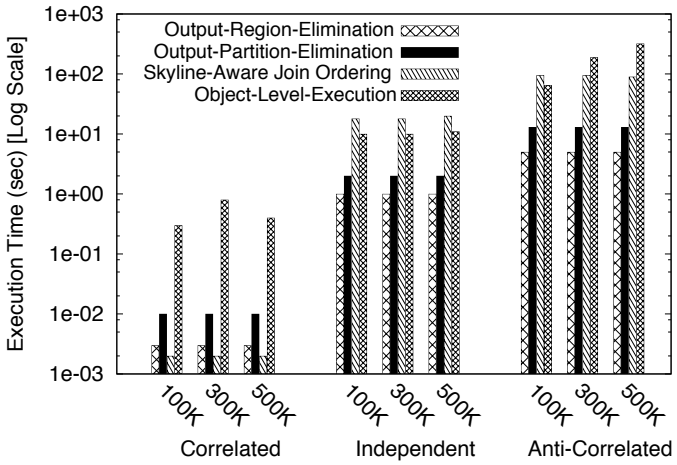


Figure 4.2: Execution Time for the Different Phases in SKIN; d=4, σ=0.01

join ordering remains constant across varying N. This is due to the fact these steps perform query evaluation at the granularity of output regions and output partitions, and stay unchanged once a given δ selection has been made. This is indeed the salient feature of our approach, namely to conduct much of the reasoning and dominance elimination early on at the cheaper partition level so that the more compute-intensive object-level effort is minimized. In contrast, the *object level execution* is affected by the change in N. More specifically, in Figure 4.2 for anti-correlated data distribution, as N increases from 100K to 500K the time needed to perform object-level execution is 65, 190 and 320 seconds respectively. This is consistent with the findings reported in Figure 4.1.c for the best δ value. It is important to note that the effects of N on *object level execution* is greatly reduced by (1) performing the dominance comparison to tuples mapped to a smaller subset of output partitions (see Chapter 3.2), and (2) the effective pruning accomplished by the first three optimization steps of SKIN. In Figure 4.2, for independent and correlated distributions the first three optimization steps of SKIN are so effective that the object-level execution step takes less than 1 and 10 seconds.

Distributions. Correlated distribution is specially geared for skyline operation, since a few objects can potentially dominate the remainder (BKS01). Most skyline algorithms

tend to do the best for such correlated data sets (BKS01, PTFS05, KLTV06). Figures 4.1.b, 4.3.a and 4.4.a show clearly that our approach is efficient in handling correlated data for all cardinalities and dimensions. Anti-correlated data is a much more challenging for skyline algorithms since they produce large skyline results (BKS01). It is interesting to observe in Figures 4.3.a and 4.4.a that our method however is robust and has significantly better performance that the state-of-the-art techniques in this most challenging scenario. The detailed comparison of the proposed approach against state-of-the-art techniques, depicted in Figure 4.5, confirms that for anti-correlated data our optimization phases are highly effective in reducing the number of skyline comparisons.

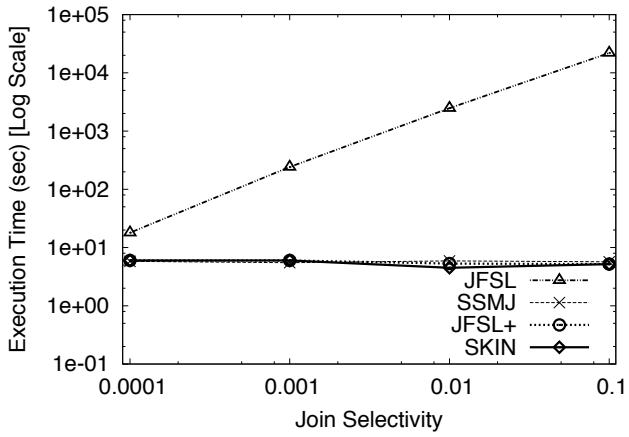
4.3 Comparisons with State-of-The-Art

Purpose. We compare SKIN against existing techniques based on three factors: (1) execution time, (2) the number of intermediate join results (combined objects) generated, and (3) the number of skyline comparisons. (KLTV06) acknowledged that *SAJ* is beneficial only for correlated data while JFSL-based techniques exhibit better performance for all distributions. Thus we focus our detailed comparative study on *JFSL*, *JFSL*$^+$ and *SSMJ*.

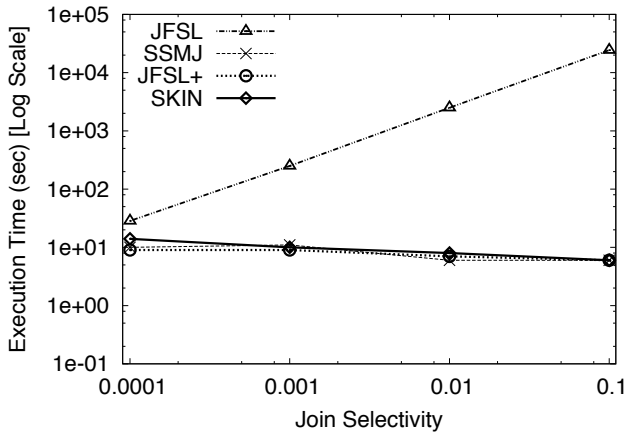
4.3.1 Execution Time

Figures 4.3 and 4.4 compare the execution times of the different techniques for d=3, and d=5 respectively. JFSL generates all possible join results which for most of cases range in the order of several million combined-objects. Due to this drawback, JFSL exhibits inferior performance as observed in Figures 4.3.a-c. For d=4 and anti-correlated data sets, JFSL ran out of memory space and failed to return results. Therefore, for $d \geq$4 we only compare our technique against *JFSL*$^+$ and *SSMJ*.

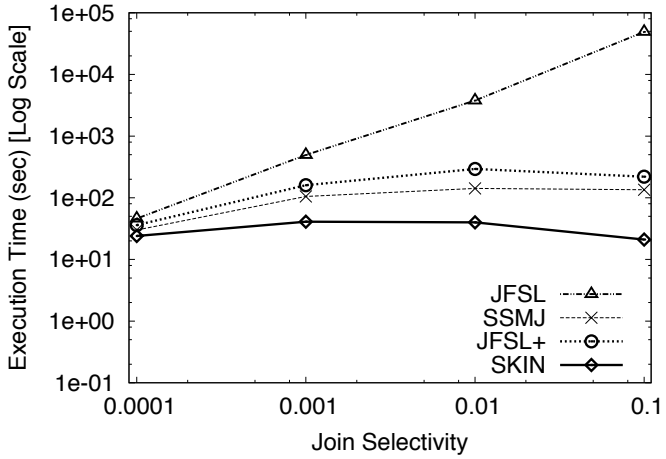
For **d=3** and for skyline friendly data distributions such as correlated and independent



(a) Correlated Distribution

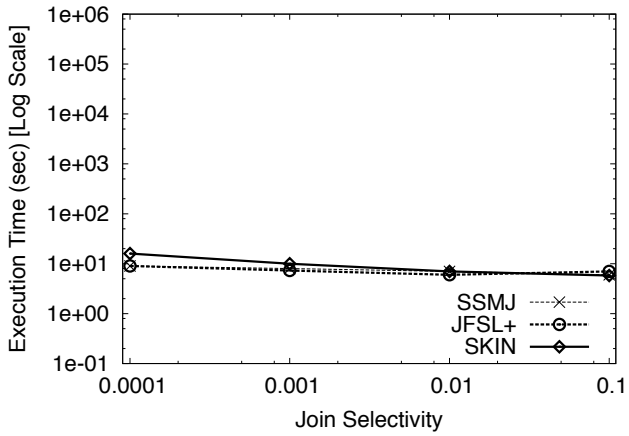


(b) Independent Distribution

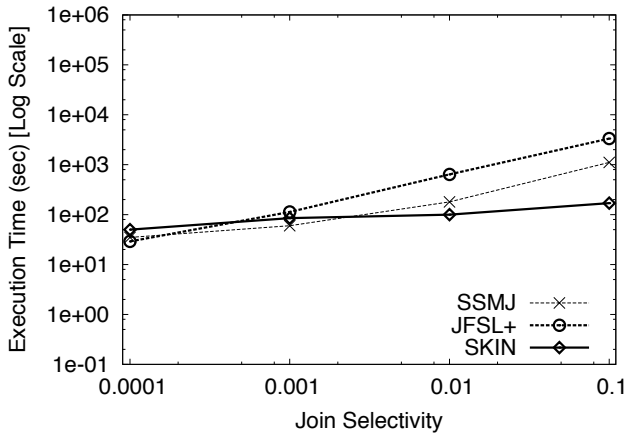


(c) Anti-Correlated Distribution

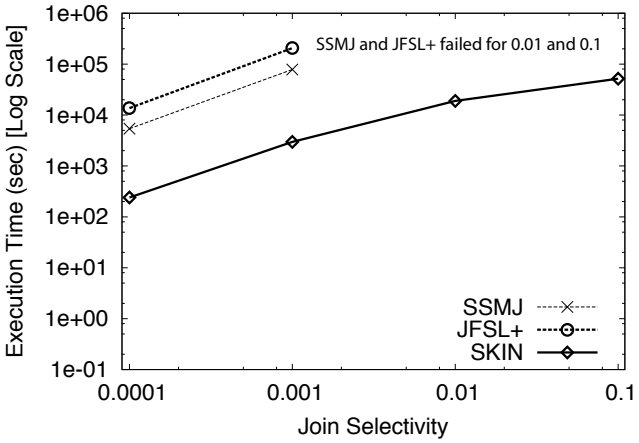
Figure 4.3: Performance Comparisons with JFSL, JFSL$^+$, and SSMJ for $d = 3$; N=500K



(a) Correlated Distribution



(b) Independent Distribution



(c) Anti-Correlated Distribution

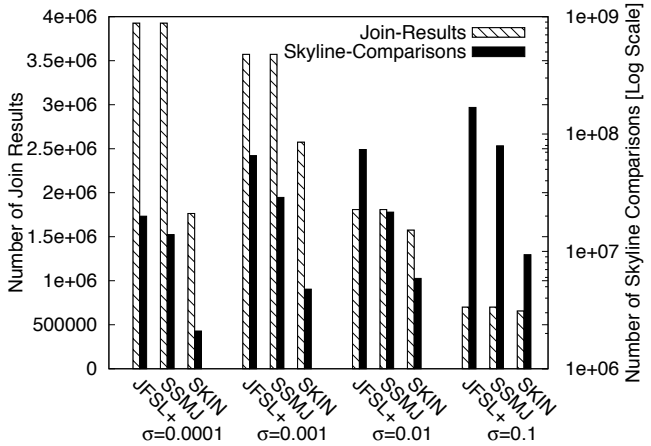
Figure 4.4: Closer Investigation with only *JFSL*$^+$, and *SSMJ* for $d = 5$; N=500K

distributions, *SKIN* has identical performance as *JFSL*$^+$ and *SSMJ* as shown in Figures 4.3.a and 4.3.b. For the tougher anti-correlated data, *SKIN* is effective and outperforms both *JFSL*$^+$ and *SSMJ* on average by 1 order of magnitude for the join selectivity of $\sigma \geq 0.001$. It is important to note that even though *JFSL*$^+$, *SSMJ*, and *SKIN* all receive the same set of pruned pre-filtered sources, and *SKIN* is able to perform better than others due to its underlying principle of query processing at different data abstractions. Therefore, from Figure 4.3 we can conclude that for d=3 *SKIN* is robust and exhibits in many cases 1 order of magnitude faster performance. Henceforth, we focus our discussion on the anti-correlated and independent data distributions.

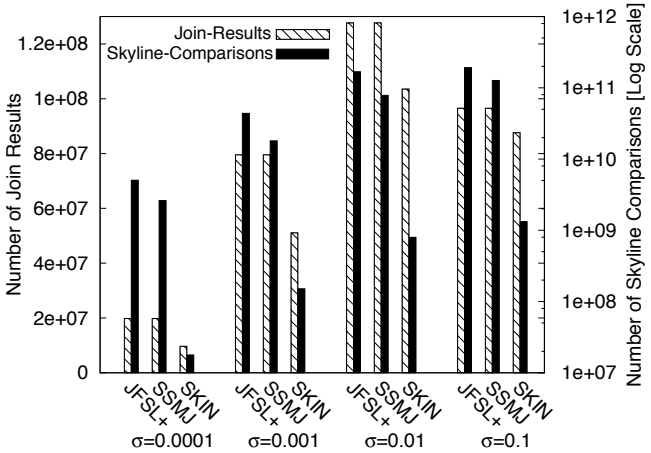
For **d=5** and independent data sets in Figure 4.4.b we observe that for the join selectivity of $\sigma = 0.0001$, *SSMJ* is 15 seconds faster than SKIN. The crossover point is $\sigma > 0.001$ when *SKIN* outperforms *SSMJ*. More specifically, by slightly less than 1 order of magnitude for $\sigma = 0.1$ and 2 folds better for $\sigma = 0.001$. For **d=5** and anti-correlated data set, in Figure 4.4 we observe the following: (1) for join selectivity $\sigma > 0.001$, both *JF-SL*$^+$ and *SSMJ* fail to return results even after several hours, (2) since *SKIN* can exploit the knowledge of the output space it can further optimize the generation of join results and minimize the number of skyline comparisons, and (3) for $\sigma = 0.0001$ and $\sigma = 0.001$ *SKIN* outperforms both *SSMJ* and *JFSL*$^+$ by a factor larger than 1 order of magnitude. Therefore, for $d = 5$ *SKIN* is robust and effective across all distributions in comparison to existing techniques.

4.3.2 Number of Join Results Generated

For correlated data sets a few tuples can dominate a large portion of the input space. Therefore the *partial push through* principle when used as a pre-filtering phase is very effective in making correlated data sets less interesting. In Figure 4.5, we compare the number of join results generated by the different algorithms on the **primary** $y - axis$ and



(a) Independent Distribution



(b) Anti-Correlated Distribution

Figure 4.5: Number of Join Results Generated, and Number of Skyline Comparisons for d=4 and N=500K

the . number of skyline (dominance) comparisons on the **secondary** $y - axis$.

Here we observe that *SKIN* produces the least number of join results across all join selectivities and for both independent as well as anti-correlated data sets. This is because *SKIN* effectively exploits the *region- and partition-level* elimination techniques proposed in Chapters 3.1 and 3.2 respectively. In Figure 4.5.a for independent data sets, *SKIN* has an average performance benefit of producing 45% fewer intermediate join results across various join selectivities when compared against both *SSMJ* and *JFSL*$^+$. More specifically, the minimum benefit is 7% lesser join results for $\sigma = 0.1$ and a maximum

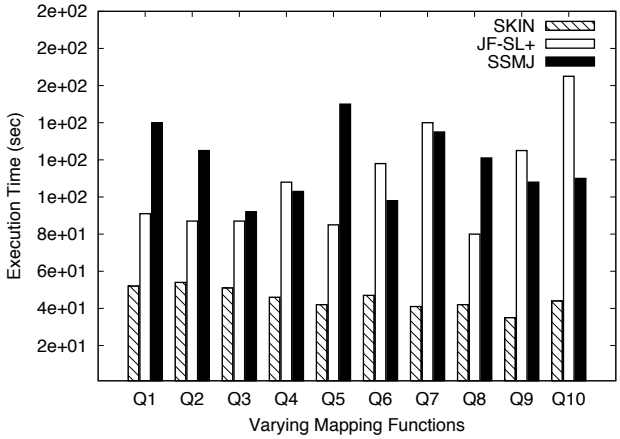
benefit of 122% for $\sigma = 0.0001$. For anti-correlated data sets, as in Figure 4.5.b, the average benefit is to produce 50% lesser intermediate join results with a minimum of 10% fewer intermediate join results for $\sigma = 0.1$ and a maximum of 105% fewer intermediate join results for $\sigma = 0.0001$.

4.3.3 Number of Skyline Comparisons Performed

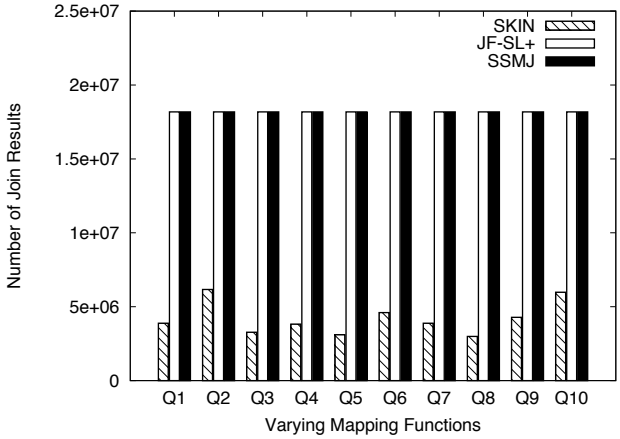
In Figure 4.5, we compare the number of skyline (dominance) comparisons on the **secondary** *y-axis* needed by the respective algorithms. For anti-correlated data, *SKIN* requires 1 and 2 orders of magnitude fewer skyline comparisons than *SSMJ* and *JFSL*$^+$ respectively to generate the final skyline. In contrast, for independent data sets *SKIN* requires 1 order of magnitude fewer skyline comparisons than both *SSMJ* and *JFSL*$^+$.

4.3.4 Differing Mapping Functions

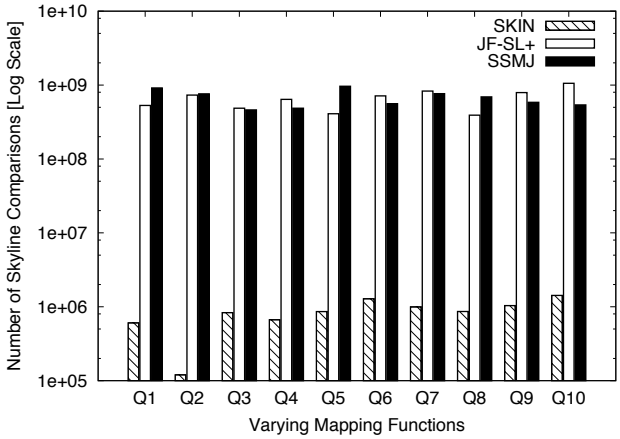
In this section, we compare the performance of the algorithms for different mapping functions. The mapping functions considered in this work are of the form $(\alpha_i * r_l[i]) + (\beta_i * t_m[i]) + \gamma[i]$ where $r_l \in R$ and $t_m \in T$. We vary the product coefficients $\alpha[i]$ and $\beta[i]$ in between [1–5] and the constant coefficient $\gamma[i]$ is chosen between [0–5]. For instance, in Figure 4.6 for Q1, the third dimension of each join tuple is produced by the mapping function $((5 * r_l[3])$ + $(3 * t_m[3]) + 4)$ where $r_l \in R$ and $t_m \in T$. In Figure 4.6.a, we compare the execution times of *JFSL+*, *SSMJ* and *SKIN*. The underlying mapping function effects the number of skyline points in the final result set. Therefore, there is a slight fluctuation in total execution costs, as in Figure 4.6.a, for each algorithm. In Figure 4.6.a when comparing execution costs, we observe for all 10 mapping functions *SKIN* performs consistently well in the range of 2-4 fold better than both *SSMJ* and *JFSL+*. The superior performance of *SKIN* even when there is a large overlap between the regions is due the



(a) Number of Join Results Generated



(b) Number of Skyline Comparisons Performed



(c) Total Execution Time

Figure 4.6: Performance Comparison for Various Mapping Functions (Anti-Correlated; $d = 3$; $N = 500K$)

effecting query processing at different levels of granularity employed by *SKIN*.

In Figure 4.6.b we observe that once *skyline partial push-through* is performed both *JFSL+* and *SSMJ* cannot further reduce the number join results that need to generated. However, by performing operations at a higher-level of granularity *SKIN* is able to still further reduce the number join results generated. More specifically, *SKIN* generates on average 78% fewer join results than its competitors. In Figure 4.6.c we compare the number of skyline comparisons performed by the algorithms where we observe that *SKIN* requires 2-3 orders of magnitude fewer skyline comparisons than *SSMJ* or *JFSL+*.

4.4 Real Data Sets

The preference is on 5 numeric dimensions obtained by adding the corresponding dimensions of the two contribution input tuples. For the *NASDAQ* data sets we use the stock-id as the join condition where there are 110 unique stock ids. In contrast for the *Household* data sets we introduce a join variable having selectivity of 0.001. The performance comparisons of the three compared techniques are reported in Table 4.1. Our proposed *SKIN* technique is shown to have better performance than both *JFSL*$^+$ and *SSMJ* for all three metrics, namely execution time, number of join results generated and number of skyline comparisons performed during query evaluations.

For *NASDAQ* data sets the distribution for the attribute values is observed to be correlated. For such correlated distributions the principle of *skyline partial push through* a small fraction of the input tuples can effectively prune a large number of input tuples. Therefore after applying skyline-push through all three techniques generate 3520 join tuples – which is indeed very small. *SKIN* is shown to conduct approximately 1 order of magnitude fewer skyline comparisons than both *JFSL*$^+$ and *SSMJ*.

Even though the principle of *skyline partial push through* is not as effective for the

Data Set	Algorithm	Time (sec)	# of Join Tuples	# of Skyline Comparisons
NASDAQ	*SKIN*	1	3520	3432
	JFSL$^+$	4	3520	28228
	SSMJ	4	3250	28228
Household	*SKIN*	9	442,953	1,312,997
	JFSL$^+$	15	835,688	52,323,765
	SSMJ	14	835,688	24,135,510

Table 4.1: Performance Comparisons over *NASDAQ* and *Household* data sets

Household data sets as it was for *NASDAQ* data sets, our proposed *SKIN* approach is shown to generate far fewer number of join results than the comparable techniques. More specifically, generating ≈2 fold fewer join results than both *JFSL*$^+$ and *SSMJ*. Lastly, *SKIN*'s object-level execution methodology enables it to perform far smaller number (> 1 order of magnitude) of skyline comparisons than the compared techniques.

4.5 Summary of Experimental Conclusions

SKIN is effective for the more difficult data distribution for skylines, namely the anti-correlated data due to the fact that we exploit the principle of abstract level processing at each step of the query evaluation process. We now elaborate in detail where these benefits occur.

1. The region-level elimination supplemented by an effective execution ordering enables *SKIN* to avoid the generation of many join tuples which are otherwise generated by compared techniques (JMP$^+$10, KLTV06) as depicted in Figures 4.5.

2. In *SKIN* we propose two additional optimization steps – output-partition level (Phase II) and object-level processing (Phase IV) described in Chpaters 3.2 and 3.4 respectively. For each newly generated join tuple $r_i t_j$, *SKIN* can (cheaply) determine the subset of (*comparable*) partitions whose tuples can potentially dominate it. In the

case of anti-correlated data sets, instead of comparing against all objects as done in existing techniques, *SKIN* reduces the comparisons to only a small set of objects that map to the comparable partitions.

3. State-of-the-art techniques (JEHH07, JMP$^+$10, KLTV06) do not conduct any abstract-level processing and instead must rely on the much more expensive pairwise object-level comparisons. Therefore, for anti-correlated data sets as reported in Figures 4.5.b and 4.6.c we show that we drastically reduce the total number of comparisons by 1-3 orders of magnitude.

The findings of our experimental analysis can be summarized as follows:

1. The *SKIN* approach is robust across all distributions, cardinality and join factors.

2. The principle of skyline *partial push-thorough* is complimentary to our work.

3. For all three data distributions, cardinalities and join selectivities, *SKIN* outperforms both *JSFL*$^+$ and *SSMJ*. More specifically, for skyline non-friendly anti-correlated data *SKIN* outperforms by 1-2 orders of magnitude in execution times. In addition, for skyline-friendly correlated data sets *SKIN* has competitive performance with respect to both *SSMJ* and *JFSL*.

4. For dimensions d=5, current techniques do not report any results even after several hours when the data set is anti-correlated and selectivity $\sigma > 0.001$. In contrast, *SKIN* is shown to perform well even for such difficult skyline non-friendly data sets.

5. For independent and anti-correlated data sets, *SKIN* produces fewer numbers of intermediate join results than compared techniques. More precisely, on average 50% fewer intermediate join results are produced.

6. *SKIN* is more effective in performing a smaller number of dominance comparisons to generate the final skyline than *JSFL*$^+$ or *SSMJ*. More specifically, *SKIN* requires 1-2 orders of magnitude fewer skyline comparisons to generate the final results.

5

Related Work for Part I

5.1 Skyline Algorithms over a Single Relation

The majority of research on skylines has focused on the efficient computation of a skyline over a single relation (BKS01, KRR02, CGGL03, PTFS03, BCP06). These algorithms can be broadly categorized as either a *non-index* or *index-based* solutions. (BKS01) proposed non-index based approaches such as *Block-Nested Loop* (BNL) and Divide and Conquer (D&C). The *block nested loop* (BNL) (BKS01) approach compares each new object against the skyline of objects considered so far. The simplicity of BNL and its ability to handle a higher dimensional skyline operator without the help of indexing and sorting makes it widely applicable. *Divide-and-Conquer* (D&C) (BKS01) divides a single data set into several memory-resident subsets. For each subset, the algorithm then evaluates a local skyline which is later merged to form the output skyline. D&C is primarily efficient when the data set and number of dimensions are small (BKS01, PTFS03).

Sort-First-Skyline (CGGL03) and *SalSa* (BCP06) based on the BNL rationale proposed algorithms that improve the performance by conducting various sorting and halting strategies. The *bitmap* algorithm proposed in (TEO01) converts each data point p into a

bit string, which encodes the number of points having a smaller coordinate than p on each dimension. When encoding a single dimension of an object, we need to know how many other objects dominate this object in that dimension and therefore this indexing is an expensive process. Additionally, this approach assumes no duplicate values. The skyline operation is then viewed as set of bitwise operations.

The *Nearest Neighbour* (NN) algorithm (KRR02) is an index-based approach that recursively divides the input region by finding the nearest neighbor to the region's origin. For higher dimensions of $d > 2$ the partitions can have some overlap thereby causing duplicate objects. (KRR02) proposes several alternative techniques such as *laisser-faire*, *merge* and *propagate* to handle duplicate elimination. Experimental evaluation in (KRR02) concludes that the performance results of *laisser-faire* and *merge* techniques make them unacceptable. To overcome these shortcomings, (PTFS05) proposed a *Branch and Bound Search* of the *R-Tree* index.

5.2 Skylines over Disparate Sources

In the context of returning meaningful results by relaxing user queries, (KLTV06) presented various strategies that follow the join-first, skyline-later (JF-SL) paradigm. This approach does not consider mapping functions. In fact, it is shown to be effective only for correlated data where the combined-object generation can be stopped early (BKS01) confirming the findings presented in (KLTV06). (JEHH07, JMP$^+$10) proposed the *Skyline Sort Merge Join* (*SSMJ*) technique and its extensions to handle skylines over join by primarily exploiting the principle of *skyline partial push-through*.

This approach suffers from the following three drawbacks: First, *SSMJ* is only beneficial when the local level pruning decisions can successfully prune a large number of objects, namely for skyline friendly data distributions such as correlated and independent

or highly selective join predicate (SWLT08). *SSMJ* proposed to first group tuples for each individual table by the join attribute(s). Next, for each join attribute domain value it generates a local skyline. Additionally, for each individual table the algorithm also maintains a table-level partial-skyline. This approach does not consider mapping functions, which is a critical component of our targeted queries. Thus it simplifies the problem because when mapping functions are considered the skyline dimensions are generated on the fly. We now present a counter example to illustrate that the dominance comparison avoidance property achieved in (JEHH07) no longer holds when mapping functions are supported as in our case. Consider the example when the relations R and T both have two objects each with the domain value 135. Specifically, r_1 (135, 0.25, 4.75) and r_4 (135 , 2.75, 3.25) in R and t_2 (135, 1.75, 5.25) and t_4 (135, 3.25, 1.75) in T, where the objects have the schema <joinAttribute, distance, price>. Since the objects in R and T both are in the group- and table-level skyline, (JEHH07) generates four skyline combined-objects namely r_1t_2, r_1t_4, r_4t_2 and r_4t_4, and outputs them as skyline results. This however is incorrect because the join result r_1t_4 (3.5, 6.5) dominates the result r_4t_2 (4.5, 8.5) and therefore is not in the final skyline. Clearly, the prior guarantees of minimizing dominance comparisons no longer hold here when mapping functions are involved.

Moreover, (JEHH07) suffers from the following disadvantages: (1) skylines and mapping functions cannot always be correctly pushed-through (KLTV06), (2) when the number of distinct variables are large the pruning capacity of the partial skyline is greatly reduced (BKS01), (3) generating group- and table-level skylines is by itself an expensive process when dealing with a large table where the number of distinct variables are few, (4) the dominance comparison avoidance property previously achieved in (JEHH07, JMP$^+$10) can no longer be ensured when handling mapping functions (as shown below), and (5) to ensure that no skyline results have been missed requires a significant amount of bookkeeping (JMP$^+$10). In our experimental study, see Chapter 4, we show that even in

data sets where *SSMJ* has good performance our proposed *SKIN*$^+$ performs equally well. Second, since existing techniques do not have any knowledge of the mapped output space *SSMJ* is unable to exploit this knowledge to reduce the number of dominance comparisons. Third, the previously held guarantee, namely objects in the set-level skyline of an individual table will clearly be in the output, no longer holds here. This is so because they do not consider mapping functions which can affect dominance characteristics.

(SWLT08, JMP$^+$10) noted that for a very low join selectivity of ≤ 0.000001, *SSMJ* is ineffective in pruning many objects both at the set- and group-level of each data source. To handle such scenarios, (SWLT08, JMP$^+$10) made modification to the core *SSMJ* approach. However, for $\sigma > 0.00001$ (when the *non-reductive* property holds for join operations) as in our work, the performance of these techniques remain identical to that of *SSMJ* (SWLT08). Following the principles proposed in (JMP$^+$10), (VDP11, KML11) recently proposed alternative sorting-based techniques.

5.3 Pushing Skyline Inside and Through Join Evaluation

To draw a parallel to *Select* in *Select-Project-Join* (SPJ) queries, there are scenarios when the skyline functionality can be *pushed inside* as well as sometimes *pushed-through* joins (BKS01, HK05). Such rewrite rules aim to facilitate processing. For instance, pushing Select inside the Cartesian product results in the theta-join operator, for which numerous efficient strategies such as index-join, sort-merge join, etc., have been devised in the literature.

In the same spirit, we observe that the skyline operation can be viewed as a complex and expensive filter operation. In our motivating query $Q1$, the attribute value of each skyline dimension is generated on the fly by the join and mapping operations. (KLTV06) noted that skylines cannot be correctly pushed-through in many scenarios. In

traditional filters increasing the number of conditions will usually increase its pruning capacity. However, adding a preference to a skyline operation will reduce their filtering capacity and increase the cardinality of its result set up to the size of the entire relation (CDK06). Therefore, the *partial push-through* of skylines can be an expensive and sometimes ineffective proposition.

(JMP+10, KML11, SWLT08) proposed techniques that rely heavily on the effectiveness of making local *partial push-through* decisions on each individual data source. By relying primarily on the principle of *partial push-through*, (JEHH07, SWLT08) are *unable to see the forest from the individual trees*. Thus by only exploiting the partial push-through they suffer from the following two limitations: (1) not being robust for a wide variety of data sets as they themselves report (SWLT08), and (2) for the generated join results techniques proposed in (JMP+10, SWLT08) cannot further optimize the skyline evaluation even when such opportunities could be found provided the knowledge of the *forest* is exploited.

Part II

Progressive Result Generation for Multi-Criteria Decision Support Queries

6

ProgXe: Progressive Execution Framework

In this section, we provide an overview of the main steps of our proposed progressive query execution framework, *ProgXe*. The *ProgXe* framework (Figure 6.1) efficiently exploits 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 (JEHH07, KLTV06, SWLT08), *ProgXe* exploits *SKIN*'s principle of processing queries at multi-levels of data abstraction to 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. Internally, the *output space look-ahead* phase of *ProgXe* exercises two *SKIN* components namely *region-level elimination* and *partition-level elimination* (described in Chapters 3.1 and 3.2 respectively) to perform the join and skyline query execution at a higher granularity of abstraction rather than at the level of individual tuples.

Next, in our **progressive driven ordering** step we investigate the output space, as

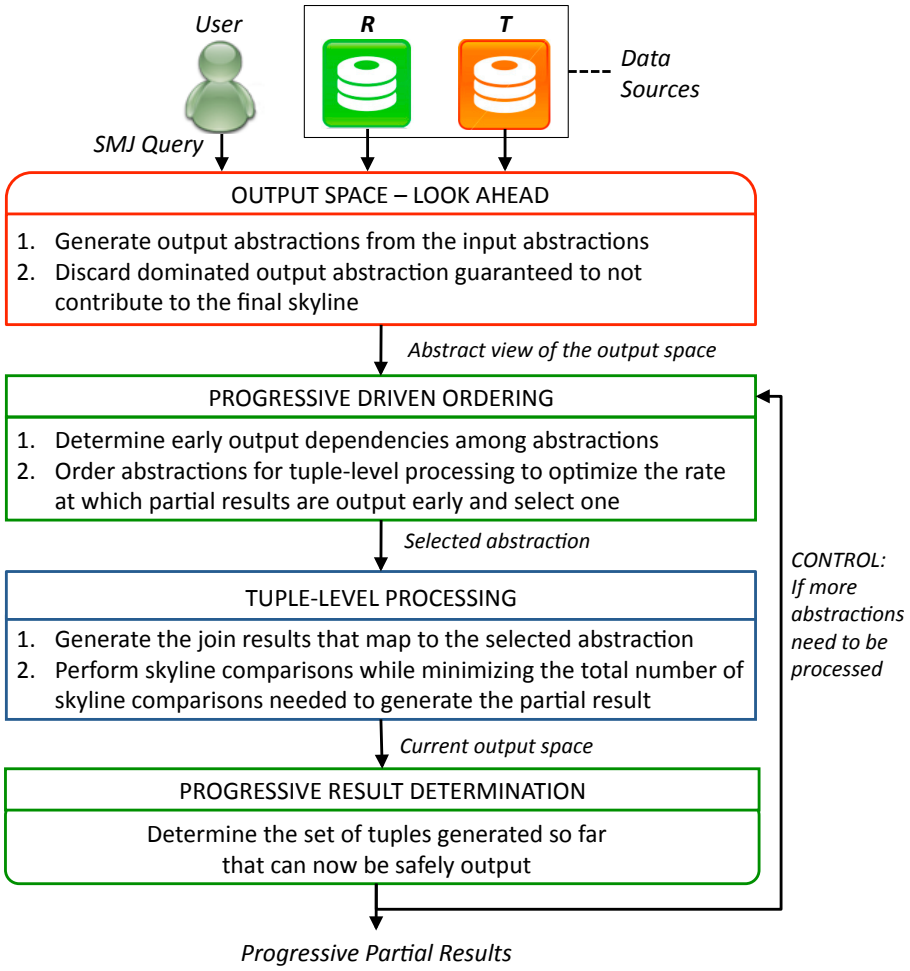


Figure 6.1: Overview of the Progressive Query Execution Framework – ProgXe

shown in Figure 6.2 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 processing to *SKIN object-level processing*.

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

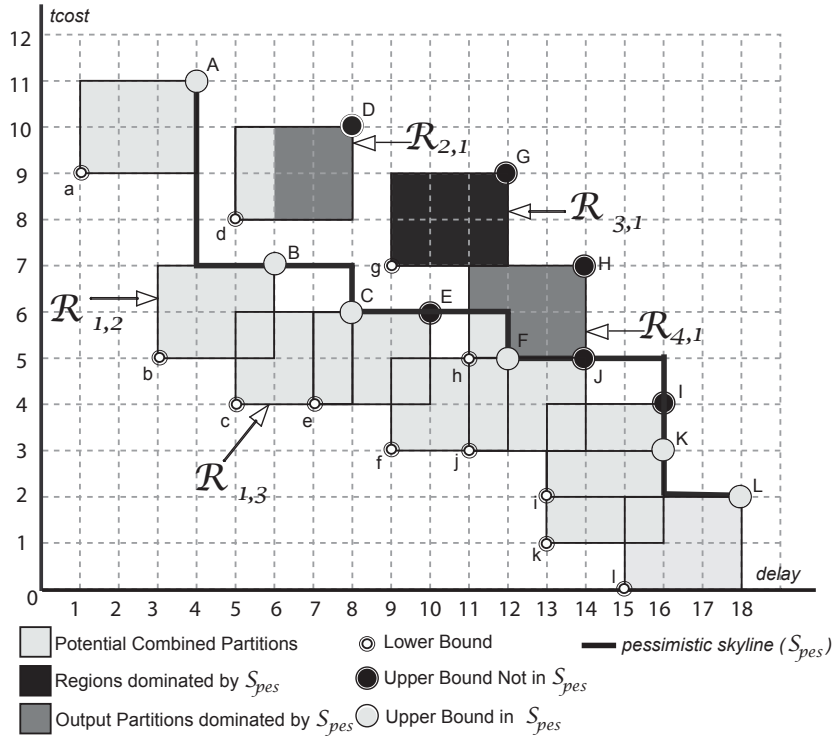


Figure 6.2: Output Space Look-Ahead: Avoid Join and/or Skyline Costs

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).

Details descriptions of the components of the *output space look-ahead* (namely, *region-level processing* and *object-level processing*) and *tuple-level processing* can be found in Chapter 6. In Chapter 7 we elaborate on the progressive benefits of ordering, followed by our *progressive driven ordering* algorithm to achieve this goal. Lastly, in Chapter 8 we present the details of our *progressive result determination* phase.

7

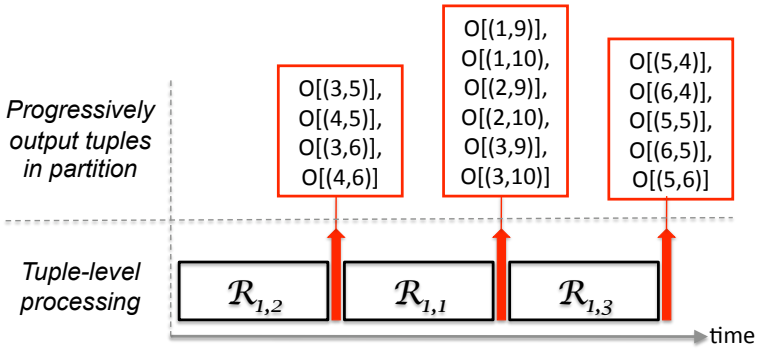
Progressive Driven Ordering

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 the 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).

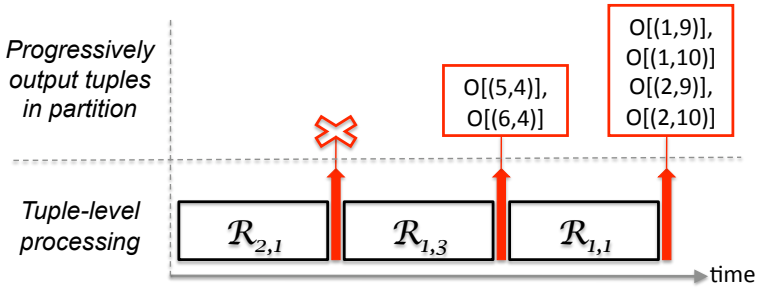
7.1 Effect of Ordering

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 7.1.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 6.2, 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.



(a) More Results Output Earlier



(b) Less Results Output Earlier

Figure 7.1: Effect 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 7.1.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 7.1.a are clearly preferable over those in Figure 7.1.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 num-

ber 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.

7.2 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, (BKST78, Buc89) 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:

$$Cardinality(\mathcal{R}_{a,b}) = ln(\sigma \cdot n_a^R \cdot n_b^T)^{d-1}/(d-1)! \tag{7.1}$$

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.

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 7.2. First, in Figure 7.2.a the lightly shaded partitions in the region $\mathcal{R}_{a,b}$ can completely eliminate *all* partitions in the region $\mathcal{R}_{e,f}$.

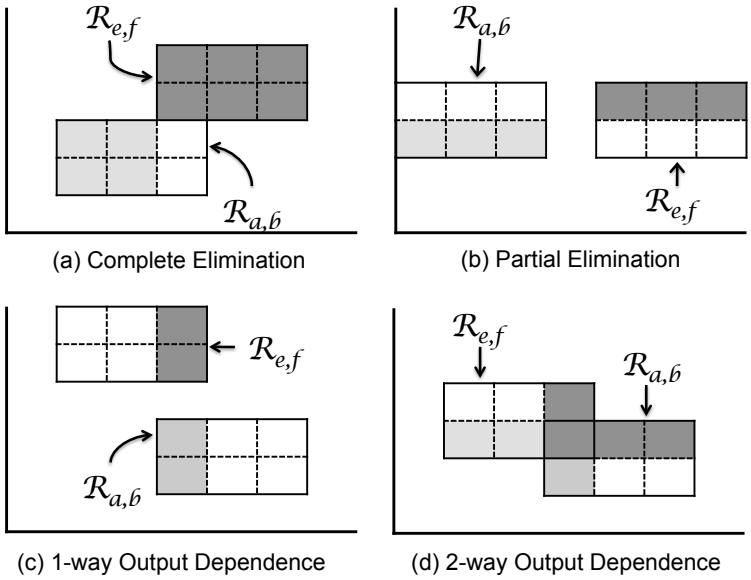


Figure 7.2: Relationships between Output Space Abstractions

Such a relationship is called **complete elimination**. Second, in Figure 7.2.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 7.2.c and 7.2.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 7.2.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 7.2.c and 7.2.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 $\mathcal{R}_{e,f}$.

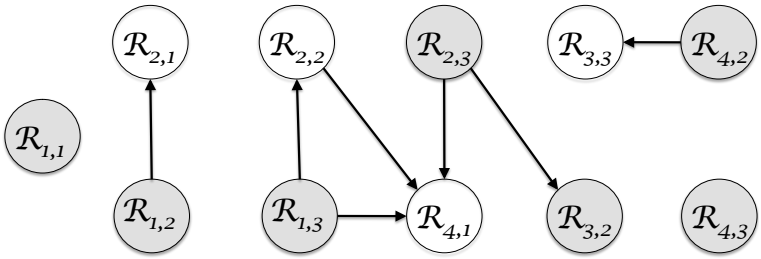


Figure 7.3: Elimination Graph (EL-Graph) (Root Nodes Depicted as Shaded Nodes)

Example 7.1 *In our running example, as depicted in Figure 6.2, 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 7.3, 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 7.3) 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 7.1) 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 7.4 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 $\mathcal{R}_{1,2}$.

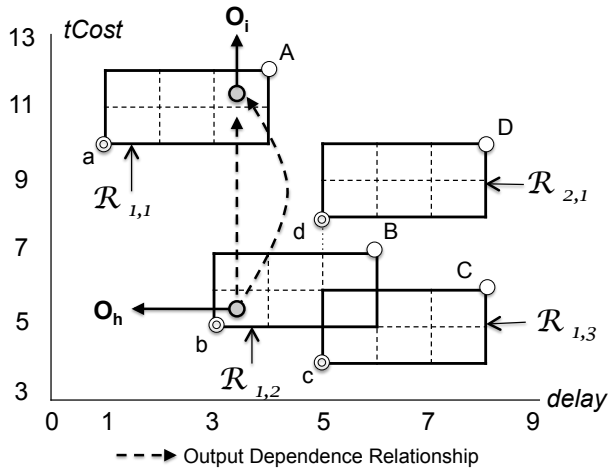


Figure 7.4: Calculating *Progressiveness Capacity*

Definition 7.1 *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 7.2 *In Figure 7.4, the progressiveness count for the output region $\mathcal{R}_{1,2}$ 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.

$$Benefit(\mathcal{R}_{a,b}) = \frac{ProgCount(\mathcal{R}_{a,b})}{PartitionCount(\mathcal{R}_{a,b})} \cdot Cardinality(\mathcal{R}_{a,b}) \tag{7.2}$$

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

7.3 Cost Model: Tuple-Level Processing

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}).

$$Cost(\mathcal{R}_{a,b}) = C_{join}(\mathcal{R}_{a,b}) + C_{map}(\mathcal{R}_{a,b}) + C_{sky}(\mathcal{R}_{a,b}) \tag{7.3}$$

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.

$$C_{join} = (|I_a^R| \cdot |I_b^T|) \tag{7.4}$$

The join results cardinality 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:

$$C_{map} = (\sigma \cdot |I_a^R| \cdot |I_b^T|) \tag{7.5}$$

Assuming independent data distribution, the average skyline execution time by (KLP75) 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 Chapter 3.4 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 *comparable partitions* for a tuple mapped an output partition (see Chapter 3.4) 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:

$$O\Bigg(\Big(CP_{avg} \cdot s_{avg}\Big) \cdot log^{\alpha}\Big(CP_{avg} \cdot s_{avg}\Big)\Bigg) \tag{7.6}$$

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 7.3 by those in Equations 7.4, 7.5 and 7.6, the amortized time for processing the output region $\mathcal{R}_{a,b}$ now is modelled as:

$$\begin{aligned} O\Bigg(&\Big(n_a^R \cdot n_b^T\Big) + \Big(\sigma \cdot n_a^R \cdot n_b^T\Big) \\ &+ (\sigma \cdot n_a^R \cdot n_b^T)\Bigg(\Big(CP_{avg} \cdot s_{avg}\Big) \cdot log^{\alpha}\Big(CP_{avg} \cdot s_{avg}\Big)\Bigg)\Bigg) \end{aligned} \tag{7.7}$$

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

7.4 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 7.2 and 7.3 as:

$$rank(\mathcal{R}_{a,b}) = \frac{Benefit(\mathcal{R}_{a,b})}{Cost(\mathcal{R}_{a,b})} \tag{7.8}$$

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 6. 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 7.8) and insert them into $PQueue$ (Line: 18). The above steps are iterated until all non-dominated regions have been considered for tuple-level processing.

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

Algorithm 6 *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, $COST(\mathcal{R}_{a,b}))$

maintaining the priority queue which is $\mathcal{O}(n \cdot log(n))$. Since typically, $n << N$ (number of tuples in one source), $\mathcal{O}(n^2) << \mathcal{O}(N^2)$.

8

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 7.2.c and 7.2.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.

8.1 Our Approach

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.

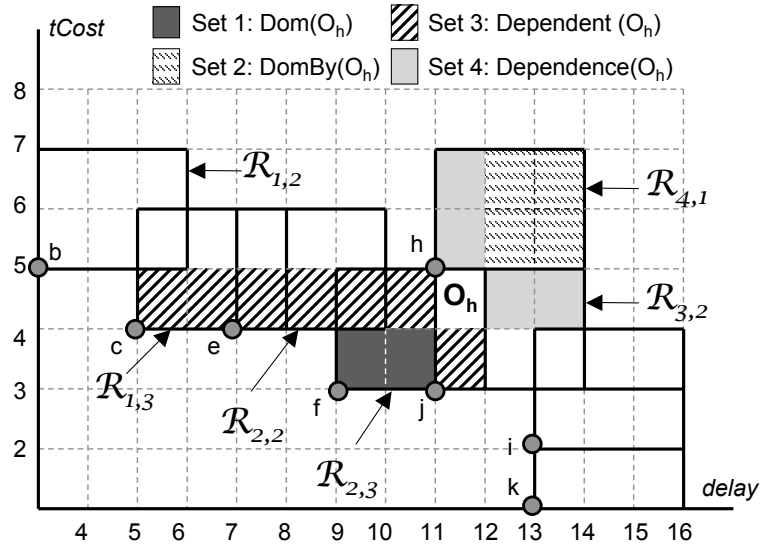


Figure 8.1: Data Maintained for O_h[(11,4)(12,5)]

To illustrate the intuition consider the running example in Figure 8.1 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 8.1 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 8.1 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 8.1 (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 8.1, we maintain a list of partitions that are guaranteed to dominate O_h if they were populated. Set 1 in Figure 8.1 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 8.1). It is denoted as $DomBy(O_h)$.

To evaluate the third condition in Principle 8.1, 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 8.1 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 8.1).

Algorithm 7 *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, $Output$);
6: Call Progressive-Output(O_h, $Output$);
7: **return** $Output$

Algorithm 8 *Progressive-Maintenance*

Input: Output Partition $\mathcal{O}_h$; $Output$ – list of output partition that can output early
1: **for** each partition $O_g \in DomBy(O_h)$ **do**
2: Remove O_h from $Dom(O_g)$
3: Call Progressive-Output(O_g);
4: **for** each partition $O_g \in Dependent(O_h)$ **do**
5: Remove O_h from $Dependence(O_g)$
6: Call Progressive-Output(O_g);

8.2 The ProgDetermine Technique: Putting It All Together

Next, we present the technique (see Algorithm 7) that utilizes the above mentioned lists to determine a set of partitions that can be output early yet safely based on Principle 8.1. First, we assume that Algorithm 7 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 (Algorithm 8) that updates the corresponding lists associated with O_h. For example, in Line: 4-6

Algorithm 9 *Progressive-Output*

Input: Output Partition $\mathcal{O}_h$; $Output$ – list of output partition that can output early
1: **if** $|Dom(O_h)| = 0) \wedge \neg IS\_MARKED(O_h)$ **then**
2: **if** $|Dependence(O_h)| = 0$ **then**
3: Add O_h to $Output$

in Algorithm 8 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. Finally, in Line: 6 in Algorithm 7 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 8.1 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, we merely decrement the corresponding count.

Time Complexity. The worst case scenario for *ProgDetermine* is when all regions the output space overlap each other. That is for each insert we need to adjust the count of $[k^d - (k-1)^d]$ partitions.

9

Experimental Evaluation of ProgXe

9.1 Experimental Setup

Alternative Techniques. State-of-the-art techniques that handle skylines over joins are: first, *JF-SL* using a hash-based join (KLTV06). Second, an optimized *JF-SL*$^+$ which uses the principle of skyline partial push-through to prune each data source. Third, *SAJ* (KLTV06) extended the popular Fagin technique (FLN01) following the JF-SL paradigm. In addition, we also compare against the popular *Skyline-Sort-Merge-Join* (*SSMJ*) technique (JEHH07, JMP$^+$10). However, as discussed in Chapter 5.2 *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. (SWLT08) noted that for low join selectivity of ≤ 0.000001, *SSMJ* is ineffective in pruning many objects.

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 *ProgXe* by varying: (1) data distribu-

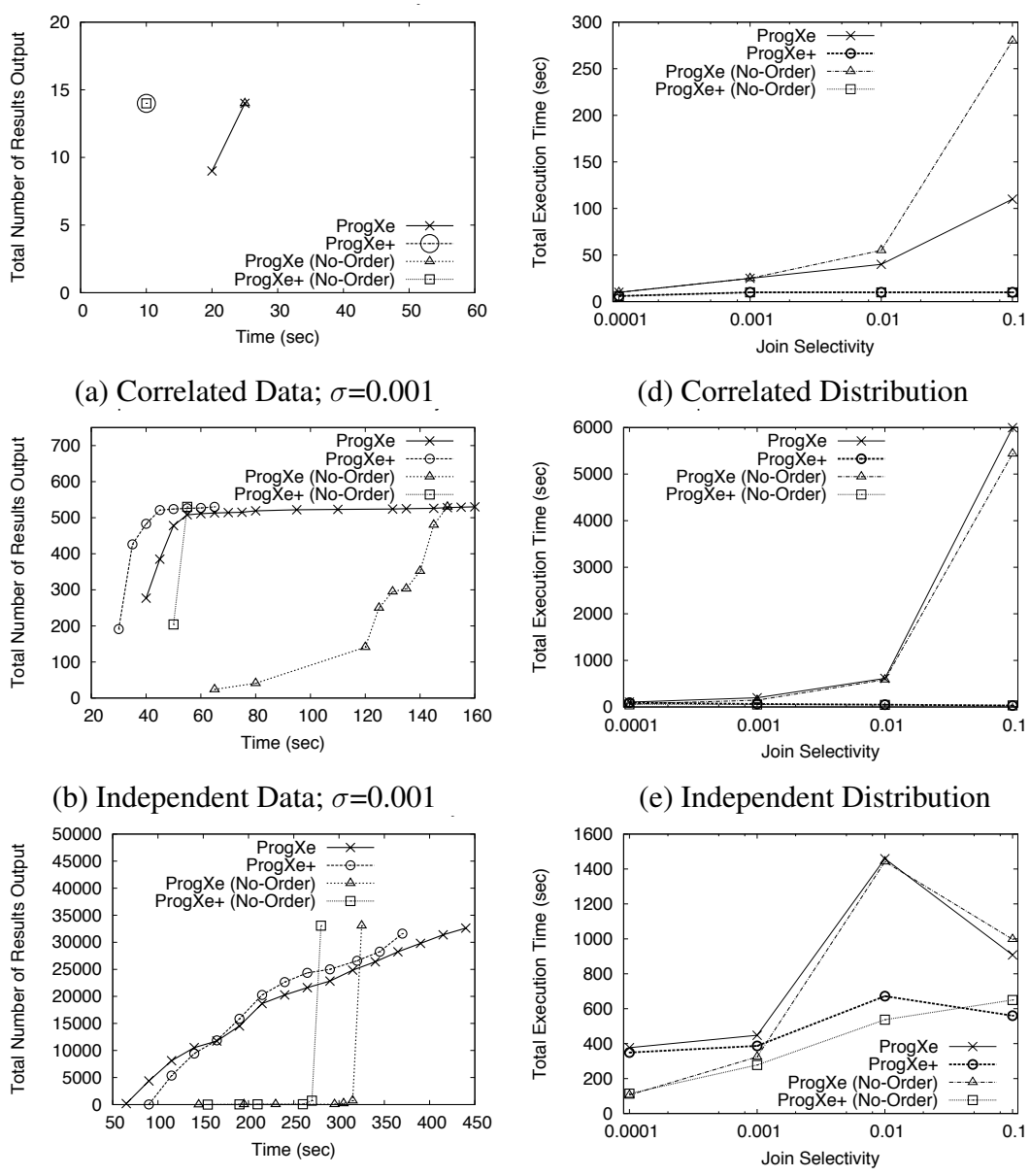
tions, (2) cardinality N, and (3) dimensions d. For each setting we measure the following: (1) the time stamp of when the output results were reported by the various algorithms to measure progressiveness, and (2) the total execution time to return the complete result set.

Data Sets. We conducted our experiments using data sets that are the *de-facto* standard for stress testing skyline algorithms in the literature (BKS01) described in Chapter 4.1.5.

9.2 Experimental Analysis of ProgXe Variations

9.2.1 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* (BKS01, HK05) 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 (RRS11).



(a) Correlated Data; σ=0.001

(d) Correlated Distribution

(b) Independent Data; σ=0.001

(e) Independent Distribution

(c) Anti-correlated Data; σ=0.001

(f) Anti-correlated Distribution

Figure 9.1: Performance Study of *ProgXe* and its variations $\lceil$*ProgXe+*, *ProgXe (No-Order)* and *ProgXe+ (No-Order)*$\rceil$ when d=4 and $|N|$=500K. Progressiveness Comparisons (a, b, and c); Total Execution Time Comparisons (d, e, and f)

9.2.2 Progressive Result Generation

In Figures 9.1.a, 9.1.b and 9.1.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 9.1.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 data sets, as in Figure 9.1.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.

9.2.3 Total Execution Time

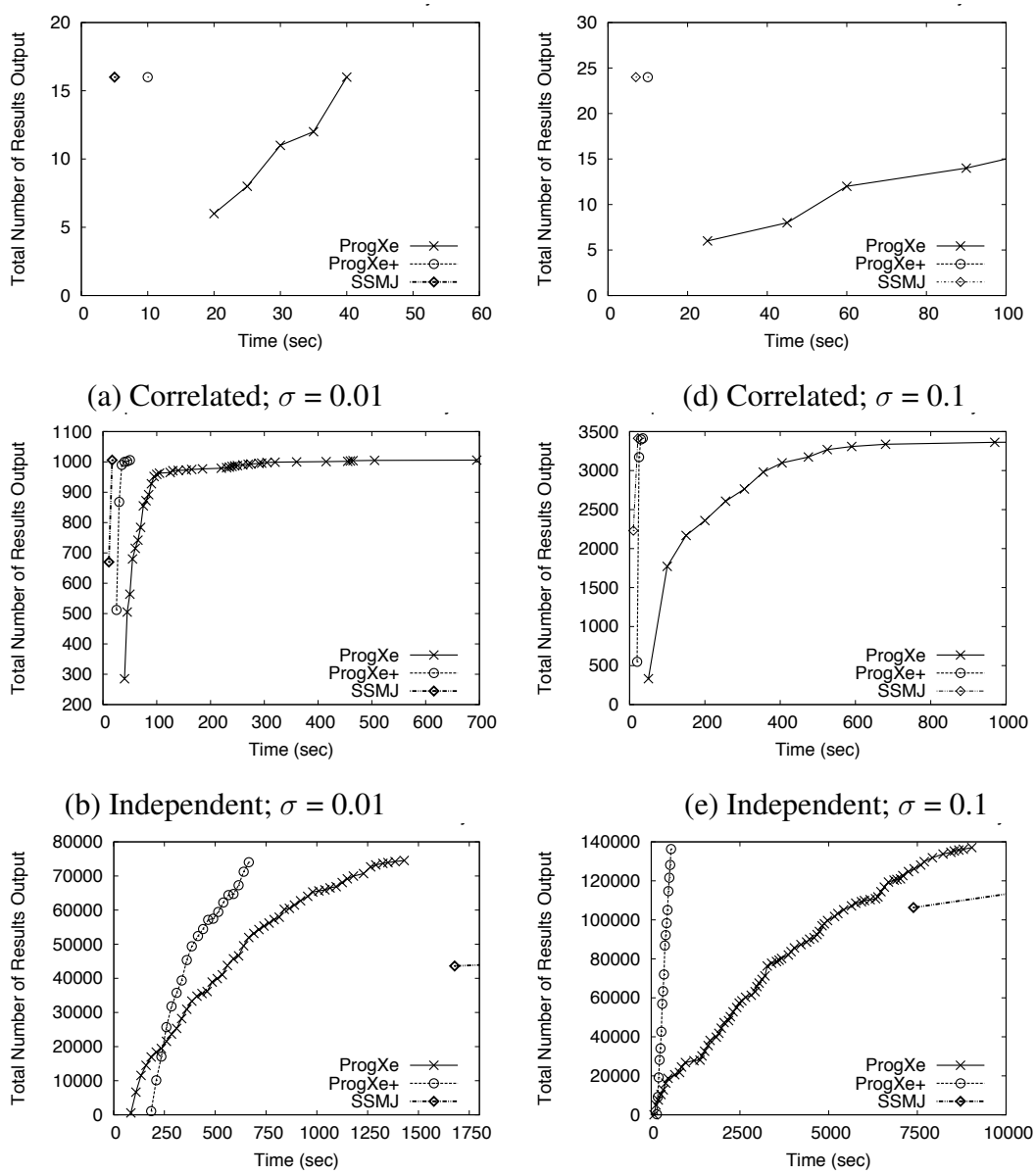
To measure the overhead of ordering we compare the total query execution time. When $\sigma < 0.01$, Figures 9.1.d, 9.1.e and 9.1.f show that *ProgXe* has identical execution time as *ProgXe (No-Order)*. This highlights that our *ProgOrder* and *ProgDetermine* algorithms are cheap and in fact can be 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 9.1 a–c.

9.3 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 (RRS11). (SWLT08) 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 9.2, 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 9.2.c and 9.2.d. For correlated data, Figures 9.2.a, and 9.2.d, we observe that *ProgXe+* has almost similar performance to *SSMJ*. In Figures 9.2.b and 9.2.e, for independent distribution, *ProgXe+* has a slightly better performance than *SSMJ*.

For **d=5** and independent data (Figure 9.4.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 9.4.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*.



(a) Correlated; $\sigma = 0.01$

(d) Correlated; $\sigma = 0.1$

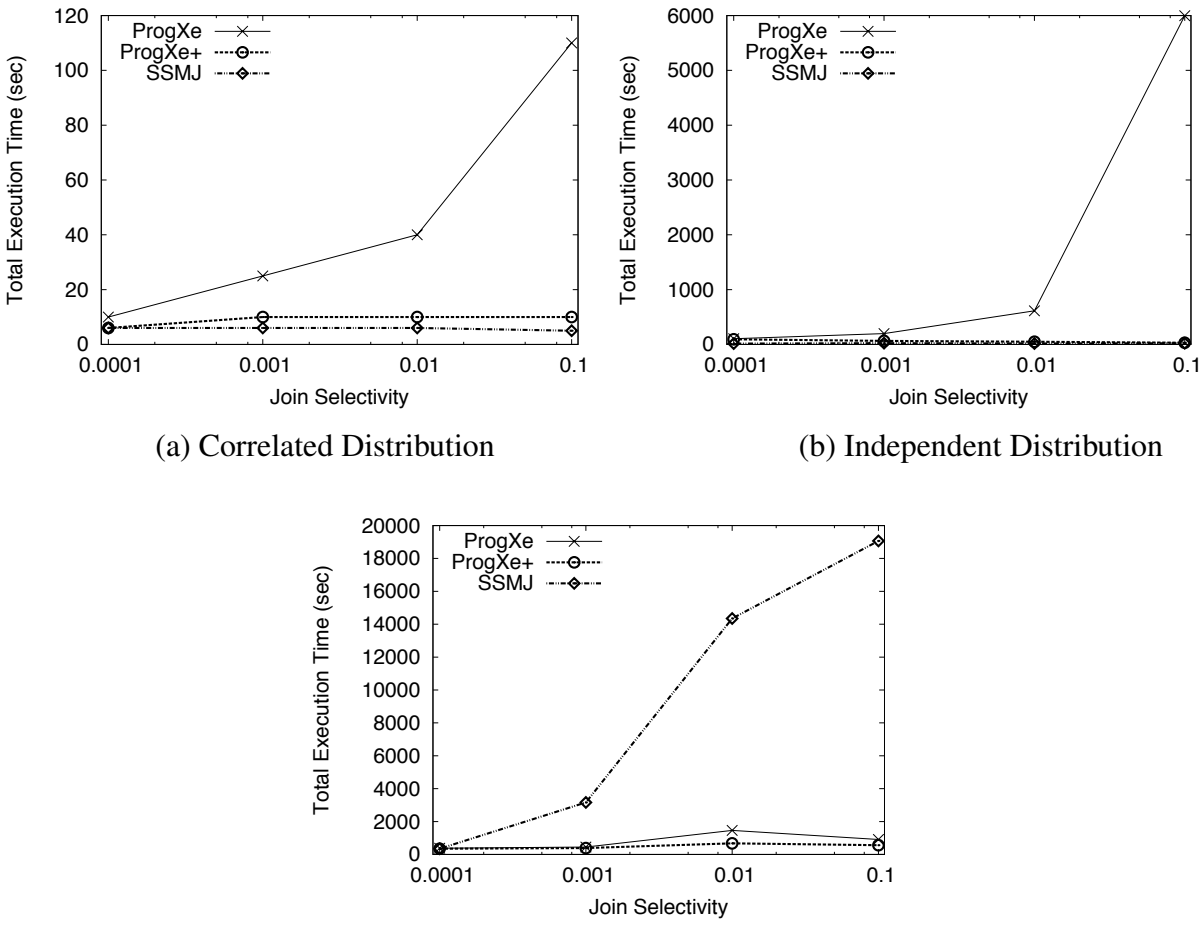
(b) Independent; $\sigma = 0.01$

(e) Independent; $\sigma = 0.1$

(c) Anti-correlated; $\sigma = 0.01$

(f) Anti-correlated; $\sigma = 0.1$

Figure 9.2: Progressiveness Comparisons of *ProgXe*, *ProgXe+* and *SSMJ*; $d = 4$ $N = 500K$



(a) Correlated Distribution

(b) Independent Distribution

(c) Anti-correlated Distribution

Figure 9.3: Total Execution Time Comparison: Proposed techniques vs. *SSMJ* ($d = 4$; $N = 500K$)

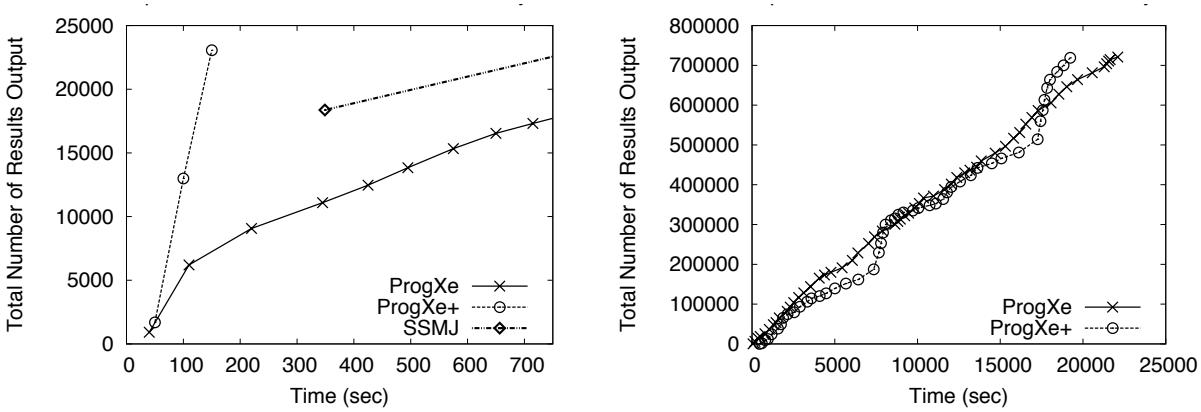
9.3.1 Summary of Experimental Conclusions

1. The progressive query execution framework *ProgXe* and its optimized *ProgXe+* are robust for all distributions, cardinalities and join factors.

2. 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



(a) Independent (b) Anti-correlated (SSMJ did not output)

Figure 9.4: Higher Dimension of $d = 5$ and $\sigma = 0.1$; *SSMJ* for Anti-Correlated Data Fails to Return Any Results After Several Hours

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 9.4.b).

10

Related Work for Part II

10.1 Blocking vs. Non-Blocking Query Operators

Relational algebra operators can be classified as either being *blocking* or *non-blocking* (HcZ07). 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 all *blocking* operations in the query into a *non-blocking* (in some case at least *partially-blocking*) operation has received much attention in the literature (CS94, YL94, HcZ07). 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.

10.2 Progressive Skyline Algorithms

In the context of single-set skyline algorithms, (TEO01, PTFS03) proposed progressive algorithms that pre-loads 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 (JMP$^+$10). 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 (TEO01, PTFS03).

Part III

Contract-Driven Processing of Multiple Multi-Criteria Decision Support Queries

11

Contract-Driven Processing of Concurrent Decision Support Queries: A Piece of CAQE

In this chapter, we propose **C***ontract*-**A***ware* **Q***uery* **E***xecution* framework - **CAQE** (pronounced *cake*) to solve the Contract-MQP problem defined in Chapter 1.2.3. *CAQE* takes as input a set of skyline-over-join queries (S_{Q}) and the associated set of contracts (S_{C}). The core principle exploited in this work is: "*different portions of the input contribute to different and often multiple queries.*" Each sub-problem serves different subset of queries with varying degrees of progressiveness. Thus, the division of work enables CAQE's execution model to adaptively choose different chunks of the input depending on the run-time satisfaction of the workload queries. Given this overall approach, we address the following open questions in CAQE:

1. Given a workload of skyline-over-join queries, how do we effectively partition the total work into chunks that maximizes execution sharing as well as progressiveness?

2. Results produced by processing one chunk can determine which subsets of the re-

sults produced by other chunks contribute to the final results of workload queries. How do we capture and exploit such output dependence amongst chunks?

3. Given a particular division of labor, how does the processing of a given input chunk affect the run-time satisfaction of the different workload queries?

4. How can we adaptively process these chunks to maximize the overall satisfaction of the workload?

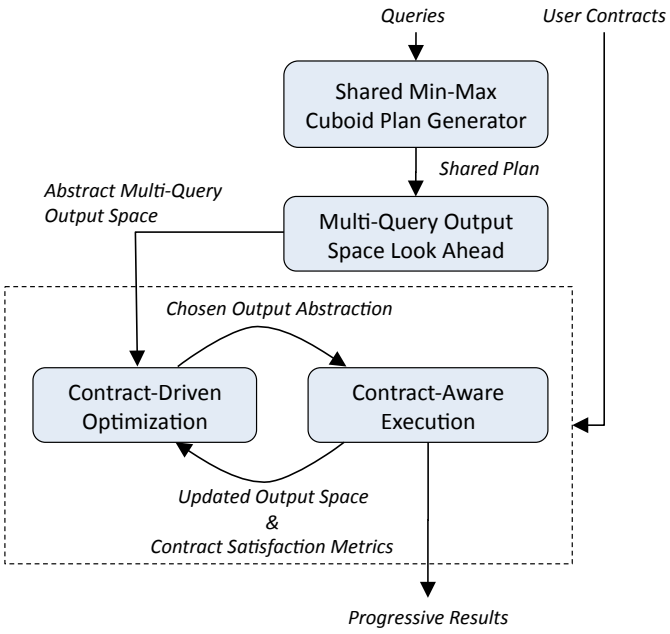


Figure 11.1: CAQE Framework

The CAQE framework depicted in Figure 11.1 is composed of four pipelined steps as described below. As the first step, CAQE generates the **shared min-max cuboid plan** $\mathcal{P}_{shared}$ for the given workload S_{Q} that maximizes the sharing of expensive operations (join evaluation and skyline comparisons) across queries (see Chapter 13).

Multi-Query Output Look Ahead evaluates the workload at a coarser granularity over the shared min-max cuboid plan to build an abstract multi-query output space that can then be divided into smaller chunks. In other words each output chunk also called the

output region containing the results of different queries is associated with a subset of coarse input abstractions containing input tuples. This model facilitates CAQE to quickly identify groups of input tuples that contribute to multiple queries – thereby facilitating execution sharing (see Chapter 14).

Contract-Driven Optimization analyzes the abstract-level output space to enable CAQE to determine the output dependencies amongst output regions across multiple queries that can be exploited to increase progressiveness for queries as designated by contracts. In this optimization step we employ a novel contract-based benefit model to determine the order in which the output regions are considered for tuple-level processing (see Chapter 15 for details).

Contract-Aware Execution iteratively processes the next output region, selected by the previous step, over the shared min-max cuboid plan. Among the join results generated thus far, the executor exploits the output dependency knowledge captured by our abstract multi-query output space to identify which subset of results can be output to any of the workload queries. Lastly, the satisfaction metrics for the QoS contracts are fed back to the optimizer to immediately rectify any inappropriate choices (see Chapter 16 for details).

To elucidate the novelty features of CAQE we use the example workload S_{Q} as in Figure 11.2 as our running example in Chapters 12 – 16.

Q_1: SMJ $_{[JC_1, \{f_1, f_2\}, X_1, P_1]}(\sigma_1(R), T)$; $P_1 = \{d_1, d_2\}$

Q_2: SMJ $_{[JC_2, \{f_1, f_2, f_3\}, X_2, P_2]}(\sigma_2(R), T)$; $P_2 = \{d_1, d_2, d_3\}$

Q_3: SMJ $_{[JC_1, \{f_2, f_3\}, X_3, P_3]}(R, T)$; $P_3 = \{d_2, d_3\}$

Q_4: SMJ $_{[JC_2, \{f_2, f_3, f_4\}, X_4, P_4]}(R, T)$; $P_4 = \{d_2, d_3, d_4\}$

Figure 11.2: Running Query Workload

12

Specifying Progressiveness Requirements via Contracts

We introduce a simple yet extensible model for specifying user QoS preferences, called the *progressiveness contract*. Based on this model we design a success metric called *progressiveness score* that measures how contracts are met by a given execution. We utilize this metric to formulate our optimization goal.

12.1 Progressiveness Contract

The progressiveness contract follows the micro-economic principle of result tuple utility(LQX06). Put differently, the *progressiveness contract* $\mathcal{C}$ for query Q is a *progressive utility function* ϑ that assigns a *utility score* to each result tuple. The result tuple τ_i is reported at time $\tau_i.ts$.

Definition 12.1 Result$(\mathcal{E}, Q, t_{start}, t_{end})$*, for query* Q *and execution* $\mathcal{E}$*, is defined as the set of all results* $\{\tau_1, \ldots \tau_N\}$ *ordered by time of the respective result generation. Here* t_{start} *is the query submission time, and* t_{end} *is the time query execution finished.*

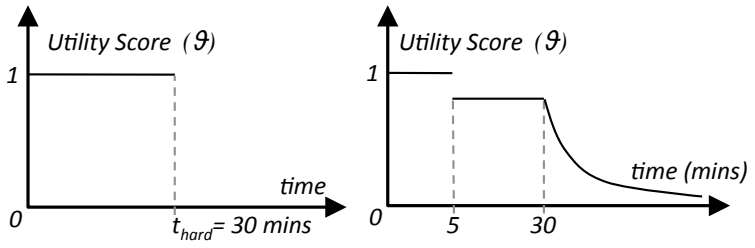
Definition 12.2 *For query Q and execution run $\mathcal{E}$, the* **progressive utility function** *ϑ is defined as a function that maps each result tuple $\tau_k \in Result(\mathcal{E}, Q, t_{start}, t_{end})$ to a utility score between 1 (most useful) and -1 (least useful) based on its usefulness.*

12.1.1 Contract Specification Models

Next, we present alternative contract models supported in *CAQE*.

12.1.1.1 Time Based Specification

Users can specify a "*time constraint*" that indicate the deadline by which all results need to be reported. Result tuples produced after the time t_{hard} are useless to the application.



(a) (b)

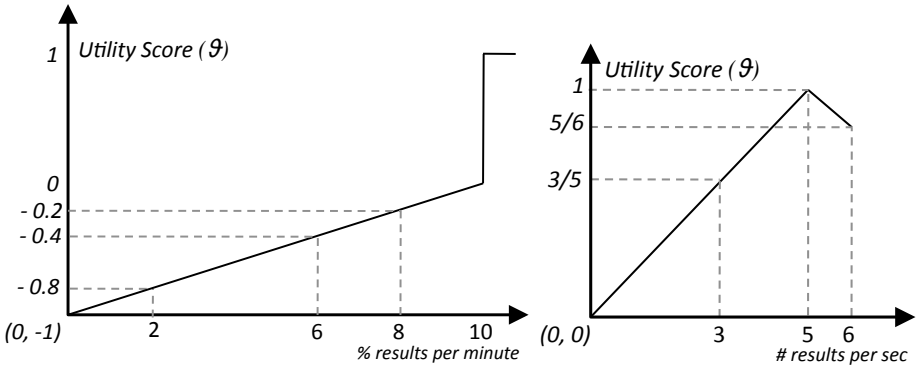
Figure 12.1: Time Based Progressive Utility Function

Example 12.1 *Figure 12.1.a depicts a time constraint where all tuples generated after 30 minutes have no use. The utility function for such time-based contract is:*

$$\vartheta_{time}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \leq 30 \\ 0 & \text{for} \quad \tau_k.ts > 30 \end{cases} \tag{12.1}$$

Example 12.2 *Alternatively, one can specify a decay function that decreases the result utility as query execution progresses. For example, the utility function of the decay function in Figure 12.1.b. is:*

$$\vartheta_{time}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \leq 5 \\ 0.8 & \text{for} \quad 5 < \tau_k.ts \leq 30 \\ log(1/\tau_k.ts) & \text{for} \quad \tau_k.ts > 30 \end{cases} \tag{12.2}$$



(a) (b)

Figure 12.2: Cardinality Based Progressive Utility Function

12.1.1.2 Cardinality Based Specification

A user may be interested in the number of results being generated at a certain time (an exact count or a percentage).

Example 12.3 *The requirement that 10% of total results to be returned every minute is represented in Figure 12.2.a where the x-axis represents the % of the estimated total number of results to be returned every minute. In Figure 12.2.a, for* $x \geq 10$ *the utility score of each tuple is equal to 1 and when* $x < 10$ *the utility score is a negative score of the ratio of the actual number of tuples generated and the required number of result tuples. The utility function for this contract is:*

$$\vartheta_{card}(\tau_k) = \begin{cases} 1 & \text{for} \quad n_{i,j}/N_{est} \geq 0.1 \\ n_{i,j}/(N_{est} * 0.1) - 1 & \text{for} \quad n_{i,j}/N_{est} < 0.1 \end{cases} \tag{12.3}$$

where N_{est} is the estimated result size of the query Q, $n_{i,j} = |Result(Q, \mathcal{E}, t_i, t_j)|$ is the number of results generated between the time interval t_i and t_j.

Example 12.4 *An example preference about the query output rate is when the user can handle at most 5 tuples/sec, the corresponding contract is depicted in Figure 12.2.b. Here, the x-axis represents the number of tuples generated every second and y-axis is the utility of each tuple. The utility function of such a contract is:*

$$\vartheta_{card}(\tau_k) = \begin{cases} (n_{i,j}/5) & \text{for} \quad n_{i,j} \leq 5 \\ (5/n_{i,j}) & \text{for} \quad n_{i,j} > 5 \end{cases} \tag{12.4}$$

12.1.2 Hybrid Specification

The user can flexibly combine several classes of specifications to specify a hybrid specification.

Example 12.5 *A stock market analyst John Doe requires at least 10% of all results to be reported every minute, while all results must be generated within 30 minutes. The utility score of tuple τ_k for this hybrid contract is obtained as the product of the utility score defined via the cardinality- and time-based contracts (see Equations 12.3 and 12.1 respectively).*

For ease of elaboration, we assume the utility scores to be independent.[1] The combined utility score of a tuple thus is:

$$\vartheta(\tau_k) = \vartheta_{card}(\tau_k) * \vartheta_{time}(\tau_k) \tag{12.5}$$

[1]The framework can support richer models that capture the dependence between the cardinality and time-based utility scores.

12.2 The CAQE Optimization Goal

Definition 12.3 *Given a set of skyline-over-join queries $S_{\mathbb{Q}}$ where each query $Q_i \in \mathcal{S}_Q$ is associated with a contract $\mathcal{C}_i$. The* **contract-driven multi-query optimization** *problem is to design a shared execution strategy $\mathcal{E}_{shared}$ of the workload that maximizes the cumulative progressiveness score of the queries in $\mathcal{S}_Q$. That is,*

$$Maximize : \sum_{i=1}^{|S_{\mathbb{Q}}|} pScore(Q_i, \mathcal{C}_i, \mathcal{E}_{shared}) \tag{12.6}$$

where *pScore* is defined as follows. Each contract $\mathcal{C}_i$ is modeled by its utility function ϑ_i and the **progressiveness score** of $\mathcal{E}_{shared}$ for $Q_i \in \mathcal{S}_Q$ is defined as the total utility score assigned to each tuple τ_k generated at time instance $\tau_k.ts$ and computed as follows:

$$pScore(Q_i, \mathcal{C}_i, \mathcal{E}_{shared}) = \sum_{k=1}^{|Result(Q_i, \mathcal{E}_{shared}, t_{start}, t_{end})|} \vartheta_i(\tau_k) \tag{12.7}$$

where $\tau_k \in Result(Q_i, \mathcal{E}_{shared}, t_{start}, t_{end})$ (see Definition 12.1).

13

Shared Min-Max Cuboid Plan

Given the set of MCDS queries, we generate a shared plan that compactly represents the query workload $\mathcal{S}_Q$. To simplify the discussion and highlight the novelty of CAQE henceforth we focus on workloads containing queries that differ in their skyline dimensions while having identical select, mapping and join conditions. However the principles presented in this work can equally apply when these conditions are relaxed.

We elucidate the intuition of our approach via the example in Figure 11.2 presented in Chapter 2.1. Under the assumption that the *Distinct Value Attributes* (DVA) *-property*[1] holds, the skyline results over the subspaces $\{d_2, d_3\}$ are guaranteed to also be in the skyline over the subspaces $\{d_1, d_2, d_3\}$ and $\{d_2, d_3, d_4\}$. If however the *DVA-property* does not hold and the tuple τ_i belongs in the skyline $SKY_{(d_2,d_3)}$, then only those tuples with the same d_2 and/or d_3 attribute values of τ_i need to be compared against to determine if τ_i belongs in $SKY_{(d_1,d_2,d_3)}$ or $SKY_{(d_2,d_3,d_4)}$. This allows us to perform dominance comparisons along dimensions d_2 and d_3 only once rather than separately for queries Q_2, Q_3 and Q_4. For d dimensions, there are $2^d - 1$ possible subspaces called *skycube* (YLL$^+$05). Since maintaining the entire *skycube* (as in Figure 13.1.a) is unnecessary, we

[1] *DVA-property states that no two tuples share the same value for any given skyline dimension (YLL$^+$05).*

instead propose to prune the space to only contain subspaces that contribute to at least one query. We call this the *reduced lattice* (Figure 13.1.b).

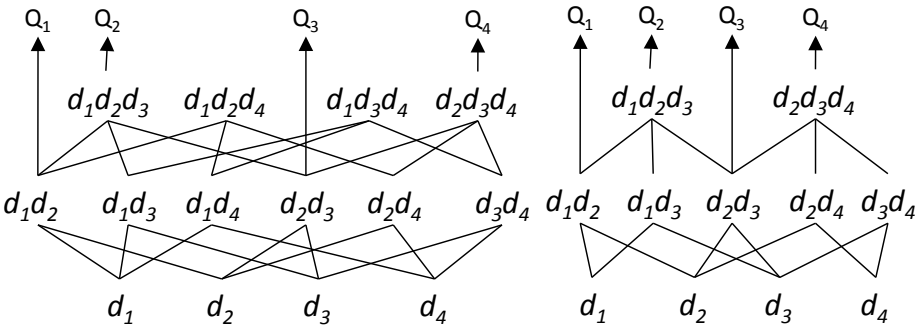


Figure 13.1: Shared Plan Generation: (a) Full Skycube; (b) Reduced Lattice Structure

Definition 13.1 *A subspace* $\mathcal{U}$ **serves** *query* $Q_i \in \mathcal{S}_Q$ *of the form* $Q_i = SMJ_{[\mathcal{JC}_i,\mathcal{F}_i,X_i,P_i]}(R,S)$ *if and only if* $\mathcal{U} \subseteq P_i$*. The set of all queries that* $\mathcal{U}$ *contributes to is denoted as* $Q_{Serve}(\mathcal{U},\mathcal{S}_Q)$*.*

Example 13.1 *In Figure 13.1.b the results in subspace* $\{d_2,d_3\}$ *contributes to queries* Q_2*,* Q_3 *and* Q_4*, where subspace* $\{d_2,d_4\}$ *contributes to only* Q_4*.*

If a given subspace $\mathcal{U}$, such as $\{d_2,d_3\}$, contributes to more than one query then the skyline comparisons performed for skyline attributes $d_i \in \mathcal{U}$ can be shared. In contrast, for query Q_4 with the final skyline dimensions $\{d_2,d_3,d_4\}$ maintaining skyline results in subspace $\{d_2,d_4\}$ does not aid in sharing skyline evaluation as $\{d_2,d_3\}$ only serves Q_4.

Next, if we have a sub-tree in the lattice rooted at subspace $\mathcal{V}$ where each subspace $\mathcal{U} \subset \mathcal{V}$ serves the same set of queries then we compactly maintain the root subspace $\mathcal{V}$ and avoid the maintenance costs of all subspaces $\mathcal{U} \subset \mathcal{V}$. We therefore propose our **min-max-cuboid** (see Definition 13.2) that is guaranteed to contain the minimal subset of subspaces while maximizing sharing.

Definition 13.2 *For a workload* $\mathcal{S}_Q$*, the* **min-max-cuboid** $\mathbb{M}$ *is the set of subspaces such that for each subspace* $\mathcal{U} \in \mathbb{M}$ *at least one of the following properties holds:*

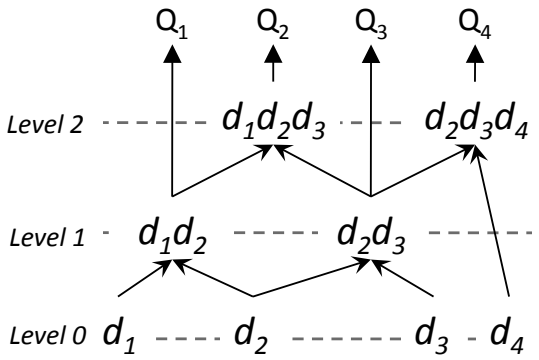


Figure 13.2: Shared Plan Generation: Min-Max Cuboid

1. $(|Q_{Serve}(\mathcal{U}, \mathcal{S}_Q)| \geq 1) \wedge (|\mathcal{U}| = 1)$

2. $\exists Q_i \in \mathcal{S}_Q$ of the form $\text{SMJ}_{[\mathcal{JC}_i, \mathcal{F}_i, X_i, P_i = \mathcal{U}]}(R, S)$

3. $\nexists \mathcal{V}$ s.t. $[(\mathcal{U} \subset \mathcal{V}) \wedge (Q_{Serve}(\mathcal{U}, \mathcal{S}_Q) \subseteq Q_{Serve}(\mathcal{V}, \mathcal{S}_Q))]$

where $|\mathcal{U}|$ *is the number of skyline dimensions in subspace* $\mathcal{U}$.

14

Multi-Query Output Look Ahead

14.1 Overview

The objective of the **M***ulti*-**Q***uery output* **L***ook* **A***head* (MQLA) step is to process the input workload at a higher granularity. This allows us to identify where the skyline results for the different queries lie in the abstract multi-query output space, thereby enabling CAQE to follow the principle of chopping the total work into smaller chunks that can be later be individually processed. We achieve this by first partitioning the input sources and then evaluating these input abstractions over the shared min-max cuboid plan generated by the previous step. Later during the actual tuple-level query execution we populate this multi-query output space with the actual skyline results as they are being generated. Mapping the input space to the multi-query output space enables CAQE to accomplish:

1. **Advanced Execution Sharing.** The coarser-grained evaluation enables us to establish the mapping between the coarse abstractions of the input space containing input tuples and the *output regions*, containing the results of different queries that are generated when the query plan is executed. Our model facilitates CAQE to quickly identify groups of input tuples that contribute to multiple queries – thereby

facilitating execution sharing and prioritization.

2. **Selective Unblocking.** The generation of the abstract-level output space enables us to identify, for each query, the subset of output regions that are guaranteed to contain skyline results. This output dependency knowledge among regions for different queries facilitates us to partially unblock the processing of skyline-over-join queries and thus prioritize the processing of such regions.

3. Aggressively **prune** output regions that are guaranteed to not generate even a single skyline result for any workload query.

We now describe the steps involved in building the multi-query output space at a coarse granularity. In Chapters 14.2-14.4 we provide a more detailed description of the coarse-level skyline processing as well as optimization steps to efficiently perform dominance comparisons. Each input table is maintained in a $d-$dimensional quad-tree. For instance, for our running example in Figure 11.2 we maintain a 4-D quad-tree on $\{d_1, d_2, d_3, d_4\}$ (the full space) one for each of input sources R and T. We define $\mathcal{L}^R$ as the set of all leaf cells for the index on R. Each input cell, say $L_i^R \in R$, is defined by its $d-$dimensional lower and upper bounds, that is $L_i^R(l_i, u_i)$.

Coarse-Level Join Operation. To facilitate joins at a coarse granularity, a leaf cell maintains a signature for each join predicate that captures the domain values of its member tuples[1]. For the running workload in Figure 11.2 with join predicates JC_1 and JC_2, each cell maintains the signatures $\{Sig_1, Sig_2\}$. Next, we perform a coarse-level join evaluation on the leaf cells to determine which output regions are guaranteed to be populated. For query Q_1 and input cells $L_i^R \in R$ and $L_j^T \in T$, if the condition $(|L_i^R[Sig_1] \cap L_j^T[Sig_1]| \neq \phi)$ holds then the output region generated by $L_i^R \bowtie L_j^T$ is guaranteed to be populated with at least one join result for query Q_1.

[1]This can be achieved by either a Bloom Filter or bit vector.

Coarse-Level Map Operation. For all output regions that are guaranteed to be populated, we apply the mapping functions, such as the total cost of the trip. To determine the d-dimensional upper bound of the output region we apply the mapping functions to the upper bounds of the two respective input cells. Similarly we obtain the lower-bound of the output region.

Coarse-Level Skyline Operation. In this step, we determine the list of output regions that do not contribute to a single query in the workload and therefore can be safely eliminated from further processing. We achieve this by performing dominance comparisons at the granularity of output regions to identify for each query Q_i, (1) the subset of output regions that are guaranteed not to be dominated, (2) the subset of output regions that can potentially contribute to the skyline results depending on the actual result distribution determined during tuple-level processing, and (3) the remaining output regions that are guaranteed to not contribute to query Q_i and therefore can be safely eliminated.

At the end of this step, for each query $Q_j \in \mathcal{S}_Q$, we return a set of non-dominated regions $REG(Q_j)$ that contribute to Q_j. Conversely, for a region R_i we define the set of queries that R_i serves as **region query lineage** $RQL(R_i)$. In the following sections we present the details of this MQLA step, including additional optimizations in building the abstract output space as well as the pseudo-code.

14.2 Coarse-Level Skyline Evaluation

The dominance comparison between any two regions R_i and R_j is only meaningful if there exists at least one common workload query that both regions serve. Henceforth, when we discuss dominance between two regions R_i and R_j we focus our attention on queries in $RQL(R_i) \cap RQL(R_j)$.

Definition 14.1 (Region Domination) *Given a subspace $\mathcal{V}$, and two regions $R_i(l_i, u_i)$,*

$R_j(l_j, u_j)$ *, the dominance relationship between these output regions is characterized as: (1)* R_i **dominates** R_j *if* $u_i \preceq_{\mathcal{V}} l_j$*; (2)* R_i **partially dominates** R_j *iff at least one output cell* $O_f \in R_i$ *and* $O_g \in R_j$*, s.t.* $u_f \preceq_{\mathcal{V}} l_g$*, (3) else* **incomparable**.

Theorem 14.1 *Given subspaces* $\mathcal{V}$ *and* $\mathcal{U}$ *such that* $\mathcal{U} \subset \mathcal{V}$*, if* R_i *is a non-dominated region the subspace* $\mathcal{U}$ *then it is guaranteed to not be dominated in subspace* $\mathcal{V}$[1].

Proof: Proof by contradiction. For subspace $\mathcal{U} \subset \mathcal{V}$ and a pair of output region $\{R_j, R_i\}$, assume that condition $(R_i \not\prec_{\mathcal{U}} R_j) \wedge (R_i \prec_{\mathcal{V}} R_j)$ holds. Given that the DVA property(YLL+05) means $\forall_{a_k \in \mathcal{V}}(u_j[a_k] < l_i[a_k])$. Since $\mathcal{U} \subset \mathcal{V}$, this translates to the fact that $\forall_{a_m \in \mathcal{U}}(u_j[a_m] < l_i[a_m])$. Hence we $(R_i \prec_{\mathcal{U}} R_j)$. This is a contradiction to our assumption that $(R_i \not\prec_{\mathcal{U}} R_j)$. Thus if R_i is not dominated in subspace $\mathcal{U} \subset \mathcal{V}$ then it is also not dominated in subspace $\mathcal{V}$. ■

Corollary 14.1 *Given subspaces* $\mathcal{U}_1$*,* $\mathcal{U}_2$ *such that* $\mathcal{U}_1 \subset \mathcal{V}$ *and* $\mathcal{U}_2 \subset \mathcal{V}$ *and* $\mathcal{U}_1 \neq \mathcal{U}_2$*, if region* $R_i \in SKY_{\mathcal{U}_1}$ *and* $R_j \in SKY_{\mathcal{U}_2}$ *then* $\{R_i, R_j\} \in SKY_{\mathcal{V}}$.

The pairwise dominance comparison between regions is conducted in a *bottom-up fashion* starting at subspaces in Level 0 of the *Min-Max Cuboid* and moving upwards. By utilizing Theorem 14.1 and its Corollary 14.1, we populate the *Min-Max Cuboid* $(\mathbb{M})$ in a bottom-up fashion to determine the list of queries a region R_i contributes to.

Example 14.1 *Let the output space consist of three regions:*
$R_1[(6, 8, 8, 4)\ (8, 10, 10, 6)]$*,* $R_2[(8, 6, 6, 5)\ (10, 8, 8, 7)]$*, and*
$R_3[(7, 5, 4, 1)\ (9, 7, 6, 4)]$*. For level* 0 *in the min-max cuboid in Figure 13.2,* R_1 *belongs in* $SKY_{(d_1)}$*, and* R_3 *belongs to* $SKY_{(d_2)}$*,* $SKY_{(d_3)}$*, and* $SKY_{(d_4)}$*. For level* 1 *by Theorem 14.1 we know that* $SKY_{(d_1,d_2)} = \{R_1, R_3\}$ *and* $SKY_{(d_3,d_4)} = \{R_3\}$*. Next, we investigate if* R_1 *contributes to* $SKY_{(d_3,d_4)}$ *and if* R_2 *belongs to either* $SKY_{(d_1,d_2)}$ *or* $SKY_{(d_3,d_4)}$*. At the end of processing, level* 1 *has* $SKY_{(d_1,d_2)} = \{R_1, R_2, R_3\}$ *and* $SKY_{(d_2,d_3)} = \{R_2, R_3\}$.

[1] Under the DVA assumption.

14.3 Optimizing MQLA

A default strategy is to perform skyline comparison for each region R_{new} with all other regions for all subspaces in the *min-max cuboid* $\mathbb{M}$. We now propose optimizations to further reduce the number of dominance comparisons.

14.3.1 Sort-Based Traversal

To reduce the number of dominance comparisons we maintain d sorted lists of regions. That is, for dimension d_j, the **ordered region list** ORL_j maintains all the output regions in a sorted order by the function f_{upper}. For region $R_f(l_f, u_f)$ with lower bound l_f and upper bound u_f the function f_{upper} is defined as:

$$f_{upper}(R_f, d_j) = u_f[d_j] \tag{14.1}$$

The intuition is that the regions with *more desired* (smaller) upper bound values on the given dimension d_i are considered before regions with less desired (larger) upper bound values.

14.3.2 Merging Subspace Skylines

For all subspaces $\mathcal{V}$ we apply Theorem 14.1 to generate the union (merge) of all non-dominated regions in the child subspaces $\mathcal{U}_i \in \mathbb{M}$ s.t. $\mathcal{U}_i \subset \mathcal{V}$. When $|\mathcal{V}| = 1$, i.e., $\mathcal{V} = d_j$, we calculate SKY_{d_j} by performing a scan on the sorted list ORL_j.

14.3.3 Sorted Subspace Skyline Maintenance

Given subspaces $\mathcal{U}_1, \mathcal{U}_2 \subset \mathcal{V}$, an output region $R_f \notin (SKY_{\mathcal{U}_1} \cup SKY_{\mathcal{U}_2})$ can still belong in $SKY_{\mathcal{V}}$. A simple but expensive way to identify such regions is to scan through all the

output regions. Instead, we propose an alternative approach that allows us to not scan the list of output regions but rather stop early. To achieve this we maintain for each subspace $\mathcal{V}$ an *ordered region list* ORL_j[1] where regions are sorted by the following function f_{sum}:

$$f_{sum}(R_f, \mathcal{V}) = \sum_{d_i \in \mathcal{V}} l_f[d_i] \tag{14.2}$$

The intuition of sorting the regions in $SKY_{\mathcal{V}}$ is that if the topmost region R_{next} in the sorted list ORL_i, has a smaller f_{sum} value, for the subspace $\mathcal{V}$, than the h^{th} region in the sorted skyline $SKY_{\mathcal{V}}$, denoted as R_h, then:

1. All output regions after R_h, including R_h, in the sorted skyline $SKY_{\mathcal{V}}$ cannot dominate R_{next}.

2. All output regions after R_h, including R_h, in the sorted skyline $SKY_{\mathcal{V}}$ can potentially be dominated by R_{next}.

This enables us to eliminate dominated regions early on and reduce the total number of skyline comparisons.

Theorem 14.2 *If R_{next} is the top most region in ORL_i and $R_f \in SKY_{\mathcal{V}}$, if $(f_{sum}(R_f, \mathcal{V}) \geq f_{sum}(R_{next}, \mathcal{V}))$ then $R_f \not\succ_{\mathcal{V}} R_{next}$.*

Proof: Case (i): $f_{sum}(R_{next}, \mathcal{V}) = f_{sum}(R_f, \mathcal{V})$. Given that $R_f \in SKY_{\mathcal{V}}$, therefore either (a) $R_f \in SKY_{\mathcal{U}}$ where $\mathcal{U} \subset \mathcal{V}$, or (b) R_f was picked before R_{next} in the ORL_i as $R_f[d_i] \leq R_{next}[d_i]$. In both cases since $f_{sum}(R_f, \mathcal{V})$ = $f_{sum}(R_{next}, \mathcal{V})$, $\exists d_h \in \mathcal{V}$ s.t. $(d_h \neq d_i) \wedge (l_{next}[d_h] < l_f[d_h])$. Therefore $R_{next}, R_f \in SKY_{\mathcal{V}}$ and $R_f \not\succ_{\mathcal{V}} R_{next}$.

Case (ii): $f_{sum}(R_f, \mathcal{V}) > f_{sum}(R_{next}, \mathcal{V})$. Let us assume $R_f \succ_{\mathcal{V}} R_{next}$. Since $f_{sum}(R_f, \mathcal{V}) > f_{sum}(R_{next}, \mathcal{V})$, $\exists d_h \in \mathcal{V}$ such that $l_f[d_h] > l_{next}[d_h]$. This is a contradiction to our assumption. Thus $R_f \not\succ_{\mathcal{V}} R_{next}$. ■

[1]The selection criteria is the dimension with the maximum number of domain values to be used to distinguish between two regions.

Corollary 14.2 *If $f_{sum}(R_{next}, \mathcal{V}) \leq f_{sum}(R_f, \mathcal{V})$ then R_{next} can potentially dominate R_f.*

14.4 Putting MQLA Together

Algorithm 10 depicts the pseudo-code of *MQLA* where we first generate all populated regions in the output space. We next sort the regions into d sorted lists. For each subspace $\mathcal{V}$ in the min-max-cuboid $\mathbb{M}$, we build the subspace skyline of output regions bottom up by applying the above mentioned optimization techniques.

Algorithm 10 Multi-Query Output Look Ahead

Input: Set S_{Q} of queries; Min-Max Cuboid $\mathbb{M}$
Output: Set $\mathfrak{R}$ of regions where each region $R_i \in \mathfrak{R}$ is not dominated for a set of queries in the workload.

1: **for** each partition $L_i^R \in \mathcal{L}^R$ **do**
2: **for** each partition $L_j^T \in \mathcal{L}^T$ **do**
3: $R_{new} \leftarrow$ Associated output region for $[L_i^R, L_j^T]$
4: Add R_{new} to $\mathfrak{R}$
5: Sort regions in $\mathfrak{R}$ by their upper-bounds (in ascending order) on every dimension d_l into sorted list ORL_l
6: **for** Subspace $\mathcal{V} \subset \mathbb{M}$ **do**
7: Initialize $SKY_{\mathcal{V}} \leftarrow$ union of skylines of all child cuboids
8: Pick ORL_i s.t. $d_i \in \mathcal{V}$
9: **while** $ORL_i \neq \phi$ **do**
10: $R_{next} \leftarrow$ Pop the top-most region in ORL_i
11: **if** $R_{next} \notin SKY_{\mathcal{V}}$ **then**
12: **for** each region $R_f \in SKY_{\mathcal{V}}$ **do**
13: **for** $\forall Q_j \in R_f[RQL] \cap R_{next}[RQL]$ **do**
14: **if** $(f_{sum}(R_{next}, \mathcal{V})) > f_{sum}(R_f, \mathcal{V}) \wedge (R_f \preceq_{\mathcal{V}} R_{next})$ **then**
15: Remove Q_j from $R_{next}[RQL]$; Go to Line 9
16: **else if** $(f_{sum}(R_{next}, \mathcal{V}) < f_{sum}(R_f, \mathcal{V})) \wedge (R_{next} \preceq_{\mathcal{V}} R_f$ **then**
17: Remove $\mathcal{R}_f$ from $SKY_{\mathcal{V}}$

15

Contract-Driven Optimization

The **contract-driven optimizer** is responsible for ordering the execution of output regions generated during the previous MQLA step such that (1) the contracts of the different queries are being met and (2) to maximally exploit sharing opportunities across these queries. More formally, for a given set of user contracts and the set $\mathfrak{R}$ of output regions, the objective of the **contract-driven optimization** is to guide query execution such that the cumulative satisfaction of the workload is maximized (see Definition 12.3)

Finding the optimal execution ordering of regions requires the comparison of all possible orderings. This involves estimating where in the d- dimensional multi-query output space the skyline results of the different workload queries will fall and to determine how they will affect the QoS requirements. This is not practical – requiring the exhaustive generation all join and skyline results of all queries. Instead, we iteratively pick the next best region until all regions have been considered based on the estimated benefit of satisfying the QoS requirements of the different queries. This feedback-driven iterative approach provides us an opportunity to identify the impact of each region selection decision on the contract satisfaction metric and take corrective immediate actions whenever a poor choice is being made.

15.1 Contract Satisfaction Metric

We identify "*the current best*" candidate for execution as the output region with the highest contract-based satisfaction metric at the given time instance t_{curr}. Let us assume that output region R_c requires time t_c to complete its tuple-level evaluation over the shared min-max cuboid plan, and is estimated to progressively output $ProgEst(R_c, Q_i, t_c)$ (see Equation 15.3) for query Q_i after time $t_{curr} + t_c$. We assign each query a run-time weight w_i. At the start of the query execution we set $\forall_{Q_i \in \mathcal{S}_Q}(w_i = 1)$. The **Cumulative Satisfaction Metric** (CSM) of R_c at time t_c is:

$$CSM(R_c, \mathcal{E}_{shared}, \mathcal{C}, t_c) = \sum_{Q_i \in S_Q} w_i * \sum_{j=1}^{N^i_{est}} \vartheta_i(\tau_j) \tag{15.1}$$

where $N^i_{est} = ProgEst(R_c, Q_i, t_c)$ and the utility score ϑ_i is associated with the contract $C_i \in \mathcal{C}$ of query Q_i.

To compute CSM for each region R_c, we develop a cardinality model to estimate: (1) for each query $Q_i \in \mathcal{S}_Q$ the number of skyline results that can be output early at time $t_{curr} + t_c$ — **the benefit of considering** R_c and (2) the number of intermediate results generated by the shared query plan that will affect the execution time for R_c (i.e., t_c) — **the cost of considering** R_c for query execution. Next, we describe our cardinality estimation model.

15.2 Multi-Query Progressiveness Based Benefit Model

To estimate the progressiveness benefit of a region, we introduce the **multi-query dependency graph** that captures the output dependencies among the different output regions.

Definition 15.1 *A* **multi-query dependency graph** *is a directed graph, denoted as* $DG(V, E)$*, where (1)* V *is the set of vertexes, where each vertex represents a region; (2)* E *is a set*

of directed edges between regions where an edge $e_{i,j}$ between regions R_i and R_j is annotated with the set of queries $\mathbb{W}_{i,j}$ for which R_i dominates one or more output cells in R_j.

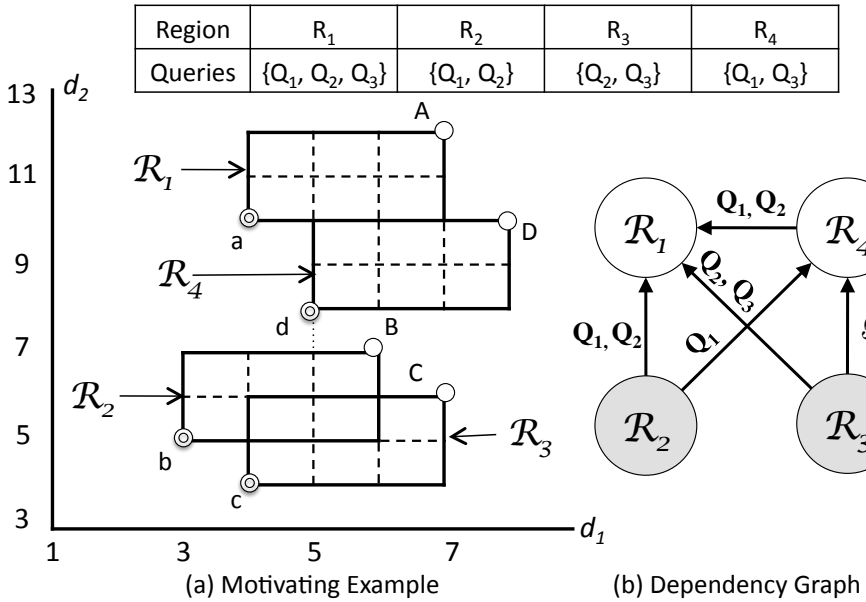


Figure 15.1: multi-query dependency graph

Example 15.1 *In Figure 15.1, output regions R_1, R_2, R_3, and R_4 contribute to different subsets of workload queries. For queries Q_1 and Q_2, R_2 has cells, that if populated during query tuple-level processing, can completely dominate R_1. Therefore R_2 should be considered for execution before R_1 to avoid unnecessary computation. We denote this dependency by the directed edge $\overrightarrow{R_2R_1}$ annotated by the set $\mathbb{W}_{2,1} = \{Q_1, Q_2\}$.*

Example 15.2 *In Figure 15.1 the shaded nodes representing regions R_2 and R_3 are both root nodes. This is because cells $O[(3,5)(4,6)]$, $O[(3,6)(4,7)] \in R_2$ and cells $O[(4,4)(5,5)]$, $O[(5,4)(6,5)]$, $O[(6,4)(7,5)] \in R_3$ cannot be dominated by any other cell in the output space for queries Q_1, Q_2 and Q_3. At the end of evaluating R_2, result tuples that map to these cells in R_2 can be progressively output for queries Q_1, Q_2 and*

Q_3. *Similarly if* R_3 *were to be chosen for execution the above mentioned cells can also output early.*

As root regions are sent for execution, non-root regions become root nodes making them candidates for possible future execution.

Definition 15.2 *The* **progressive estimate** *of region* R_c *for query* Q_i *at time* t *is the fraction of all the results produced by* R_c *that are guaranteed to be in the final skyline at time* t.

Let L_a^R and L_b^T be the input cells contributing to the region R_c. For query Q_i with selectivity σ_i, the estimated number of skyline results R_c can produce is established by (Buc89) as:

$$Cardinality(R_c, Q_i) = ln(\sigma_i \cdot n_a^R \cdot n_b^T)^{d-1}/(d-1)! \tag{15.2}$$

where $n_a^R = |L_a^R|$ and $n_b^T = |L_b^T|$.

Definition 15.3 *The* **progressive cell count** *(*$ProgCount$*) for a region* R_c *at time* t *and query* $Q_i \in RQL(R_c)$ *is the total number of cells in* R_c *that are not dominated by cells mapped to another region that contributes to the same query* Q_i.

Example 15.3 *In Figure 15.1.a let us assume that the output cells* $O[(3,5)(4,6)]$ *and* $O[(3,6)(4,7)]$ *are populated during tuple-level processing of* R_2. *Then, all output cells in the output region* R_1 *can be dominated for queries* $\{Q_1, Q_2\}$. *Thus, the* $ProgCount(R_1, Q_1) = ProgCount(R_1, Q_2) = 0$. *In contrast, for query* Q_3 *the progressive count for* R_1 *is 2 since tuples that map to its cells* $O[(5,8)(6,9)]$ *and* $O[(5,9)(6,10)]$ *can be progressively output at the end of processing* R_1 *since the remaining output cells could potentially be dominated by tuples that map to region* R_3.

From Definition 15.3 and Equation 15.2 the progressiveness estimate of R_c for query Q_i can defined as follows:

$$ProgEst(R_c, Q_i, t) = \Big(\frac{ProgCount(R_c, Q_i, t)}{CellCount(R_c, Q_i)}\Big) * Cardinality(R_c, Q_i) \tag{15.3}$$

where $CellCount(R_c, Q_i)$ denotes the total number of output cells in the region R_c for query Q_i.

15.3 Multi-Query Cost Model

The time t_c required to process R_c is considered to be the penalty for choosing R_c. This includes the time needed : (1) to materialize the join results (C_{join}), (2) conduct mapping (C_{map}), and (3) perform skyline comparisons (C_{sky}).

$$t_c = C_{join}(R_c) + C_{map}(R_c) + C_{sky}(R_c) \tag{15.4}$$

For an output region R_c we estimate the cost of join as the product of the cardinalities of their respective input cells (L_a^R and L_b^T).

$$C_{join}(R_c) = (n_a^R \cdot n_b^T) \tag{15.5}$$

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 can be estimated as $O(1)$, then:

$$C_{map}(R_c) = (\sigma \cdot n_a^R \cdot n_b^T) \tag{15.6}$$

We estimate the skyline comparisons with regards to the full d-dimensional space. Assuming an independent data distribution, the average skyline execution time according to Kung et. al. (KLP75) 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 $d \geq 4$. For each newly generated join result $r_f t_g$ that maps to the output cell O_{curr} we need to perform dominance comparison with tuples in at most CL_{avg} cells in the full d-dimensional subspace, where CL_{avg} is the average number of comparable cells to O_{curr}. We therefore conduct $(CL_{avg} \cdot s_{avg})$ comparisons, where s_{avg} is the average number of tuples in each output cell. Thus, the amortized time for evaluating the dominance comparison for a single intermediate result $r_f t_g \in O_{curr}$ is:

$$O\left(\left(CL_{avg} \cdot s_{avg}\right) \cdot log^{\alpha}\left(CL_{avg} \cdot s_{avg}\right)\right) \tag{15.7}$$

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

From Equations 15.4, 15.5, 15.6 and 15.7, the amortized time for processing the output region $R_{a,b}$ now is modeled as:

$$\begin{aligned} O\Bigg(&\left(n_a^R \cdot n_b^T\right) + \left(\sigma \cdot n_a^R \cdot n_b^T\right) \\ &+ (\sigma \cdot n_a^R \cdot n_b^T)\left(\left(CL_{avg} \cdot s_{avg}\right) \cdot log^{\alpha}\left(CL_{avg} \cdot s_{avg}\right)\right)\Bigg) \end{aligned} \tag{15.8}$$

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

15.4 Putting Contract-Driven Ordering Together

The *contract-driven optimizer* iteratively determines the next region to be considered for query execution. The pseudo-code of the *contract-driven optimizer* is listed in Algo-

Algorithm 11 *Contract-Driven Ordering*

Input: $\mathfrak{R}$ (Region Collection); input cells ($\mathcal{L}^R$, $\mathcal{L}^T$); query workload $\mathcal{S}_Q$ contracts $\mathcal{S}_C$
Output: Iteratively pick the next region R_{next} to process.

Build the initial *multi-query dependency graph*, $Multi - Query - DG$
for each R_c in DG_{root}: **do**
  computeCSM($R_c, \mathcal{S}_Q, \mathcal{S}_C$)
  Add R_c to inverted priority queue $PQueue$ (sort by CSM)
while $|\mathfrak{R}| \neq \phi$ **do**
  $R_c \leftarrow$ remove($PQueue$) {Top of the list}
  Perform *contract-driven execution* for region R_c
  Discard regions dominated by the generated tuple(s) in R_c.
  for each edge $e_{c,f} = \overrightarrow{R_c, R_f} \in DG$ **do**
    Remove $e_{c,f}$
    if $R_f \in PQueue$ **then**
      Update its CSM scores.
      $DG_{root'} \leftarrow$ new root nodes due to the removal of $e_{c,f}$
  for each $R_g \in DG_{root'}$ **do**
    computeCSM($R_g, \mathcal{S}_Q, \mathcal{S}_C$)
    Add R_g to $PQueue$
  $DG_{root} \leftarrow DG_{root} \cup DG_{root'}$
  Remove R_c from $\mathfrak{R}$
return

function computeCSM($R_c, \mathcal{S}_Q, \mathcal{S}_C$)
Calculate the cost to process the region t_c
for each query $Q_i \in \mathcal{S}_Q$ **do**
  Compute ProgEst(R_c, Q_i) $\rightarrow |RS(R_c, Q_i)|$ (Definition 15.3)
  $CSM = CSM + (w_i * pScore(R_c, Q_i, \mathcal{C}_i))$
return sat

rithm 11. The *progressive benefit model* (Equation 15.3) estimates the number of tuples that a region can output after its evaluation. Our cost model (Equation 15.8 in Chapter 15.3) estimates the time needed (t_c) to evaluate region R_c over the shared query plan. The root regions in the *dependency graph* are ranked based on their CSM scores (Equation 15.1) and maintained in an inverted priority queue. We iteratively pick the topmost region from the queue for tuple-level processing. Based on how the run-time QoS contracts of each query Q_i are being met, we update the CSM-based benefit model by adjusting its

weight (see Chapter 16). This process is repeated until all regions have either been considered for tuple-level processing or have been dominated by newly generated tuple(s).

16

Contract-Driven Execution

Next, CAQE's *contract driven execution* engine then executes the chosen output region, recommended by the optimizer, over the shared min-max cuboid plan. For each scheduled region R_c the **contract driven executor** performs the following three operations:

1. **Tuple Level Processing.** Conduct tuple-level evaluation (join, map and skyline) over the shared min-max cuboid plan.

2. **Progressive Result Reporting.** For each workload query $Q_i \in \mathcal{S}_Q$ determine the subset of the generated result tuples that is guaranteed to be in the final skyline.

3. **Run-time Satisfaction Metric and Optimizer Feedback.** Based on user contracts and the skyline results generated so far update the run-time satisfaction metric of each query $Q_i \in \mathcal{S}_Q$. Use this run-time metric update the benefit model used by the contract-driven optimizer to pick the next region.

16.1 Tuple Level Processing

For the chosen region $\mathcal{R}_c$, we first evaluate the join conditions between the tuples in the input cell L_a^R and those in L_b^T. Join results are then mapped to their output cells by

applying the mapping functions. For each output cell O_x we maintain the **cell query-lineage** (CQL) bit vector representing the list of queries that the cell contributes to. The CQL is easily derived from the *region query-lineage* of all the regions that O_x contributes to. For subspace $\mathcal{V}$ in the min-max cuboid $\mathbb{M}$, we limit the skyline comparisons for the new generated tuples in cell $O_x \in R_c$ to tuples in cells, say O_y, that satisfy both these conditions:

1. $|CQL(O_x) \cap CQL(O_y)| \neq \phi$

2. $\exists z \in \mathcal{V}$ s.t. $(l_x[a_z] = l_y[a_z])$

For all regions R_f such that there exists an edge $\overrightarrow{R_c, R_f}$ in the dominance graph $Multi-Query-DG$ (see Chapter 15.2) and queries $RQL(R_c) \cap RQL(R_f)$, we identify output cells in R_f that are dominated by the newly generated tuples in R_c. This allows us to discard all join results that map to such dominated cells for $RQL(R_c) \cap RQL(R_f)$ as they are guaranteed to not contribute to its final result of queries in $RQL(R_c) \cap RQL(R_f)$.

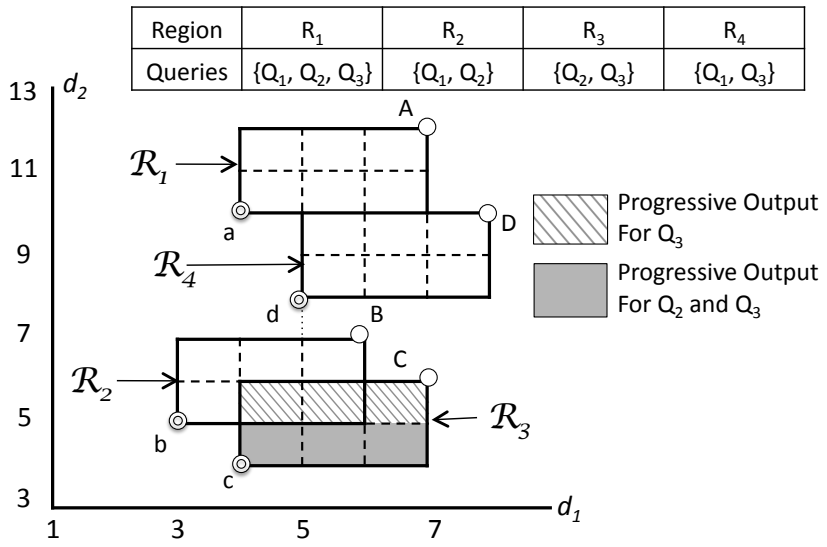


Figure 16.1: Multi-Query Progressive Output

16.2 Multi-Query Progressive Result Reporting

To support progressive result reporting, we push the decision making to the coarser granularity of output cells. More precisely, we translate the problem of determining which result can be progressively output into the process of determining output cells, say O_x, that are guaranteed to (1) not have any future results map to them, and (2) that contain only tuples that are guaranteed to not be dominated by future tuples.

Example 16.1 *In Figure 16.1, if region R_3 is picked for tuple-level processing, at the end of its processing, we can safely output all tuples in all of its output cells for query Q_3. This is due to the fact that no future tuples can dominate it (as derived from the dependency graph) and tuples in region R_2 do not contribute to Q_3. In contrast, for Q_2 we can only progressively output tuples in cells $O[(4,4)(5,5)]$, $O[(5,4)(6,5)]$ and $O[(6,4)(7,5)]$ since the tuples in the remaining cells may be dominated by future tuples that map to cells $O[(3,5)(4,6)]$, $O[(4,5)(5,6)]$ and $O[(5,5)(6,6)]$ of R_2.*

16.3 Satisfaction Based Feedback Mechanism

For each progressive result reported for query $Q_i \in \mathcal{S}_Q$ we calculate its utility by the utility function v_i defined in this contract $\mathcal{C}_i$. The Q_i's QoS metric at time instance t, is denoted as $v(Q_i, t)$, is the average utility score of all the results reporting at time t. Based on this metric we adjust the query Q_i's weight w_i in Equation 15.1 for next iteration of processing. This enables us to pick regions that satisfy queries with low run-time satisfaction to meet their respective contracts in the future. This translates to changing the weight w_i to w_i' in our CSM-based benefit model (Equation 15.1):

$$w_i' = w_i + \frac{v_{curr-max} - v(Q_i)}{\sum_j^N (v_{curr-max} - v(Q_j))} \tag{16.1}$$

where $v_{curr-max}$ is the maximum satisfaction of any single query during the current time period.

Example 16.2 *At the end of picking region R_3, let the run-time satisfaction metric of the queries be $\{0, 1, 0.7, 0\}$ i.e., $v_{curr-max} = 1$. By Equation 16.1 the new weights are $\{1.43, 1, 1.13, 1.43\}$[1]. In other words, we bump up the priorities of Q_1, Q_2 and Q_3 since they have not yet meet their respective QoS contracts.*

[1]Let us assume that the original weights $\forall_i w_i = 1$

17

Experimental Evaluation on CAQE

17.1 Experimental Settings

17.1.1 Experimental Platform

All measurements obtained on a workstation with AMD 2.6GHz Dual Core CPUs with Java heap set to 4GB. All algorithms were implemented in Java.

17.1.2 Contract Models

As described in Chapter 12.1.1, progressiveness contracts in CAQE follow the micro-economic principle to determine the utility of a result tuple. We tested CAQE's effectiveness under different classes of contracts, namely time-based (C1, C3 and C3), cardinality-based (C4) and hybrid (C5) contracts. Table 17.1 summarizes the contract models used in this study where t_{C1} and t_{C3} are tunable parameters for contracts $C1$ and $C3$ respectively, while n_{ij} is the time interval used in contracts $C4$ and $C5$. In Table 17.1 N is the total of output tuples for query Q and $\tau_k.ts$ is the output time of the result tuple τ_k.

	Utility Functions
C1	$\vartheta_{C1}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \leq t_{C1} \\ 0 & \text{for} \quad \tau_k.ts > t_{C1} \end{cases}$
C2	$\vartheta_{C2}(\tau_k) = 1/log(\tau_k.ts)$
C3	$\vartheta_{C3}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \leq t_{C3} \\ 1/(\tau_k.ts - t_{C3}) & \text{for} \quad \tau_k.ts > t_{C3} \end{cases}$
C4	$\vartheta_{C4}(\tau_k) = \begin{cases} 1 & \text{for} \quad n_{i,j}/N \geq 0.1 \\ n_{i,j}/(N * 0.1) - 1 & \text{for} \quad n_{i,j}/N < 0.1 \end{cases}$
C5	$\vartheta_{C5}(\tau_k) = \vartheta_{card}(\tau_k) * \vartheta_{time}(\tau_k)$, where $\vartheta_{time}(\tau_k) = 1/\tau_k.ts$; $\vartheta_{card}(\tau_k) = \vartheta_{C4}(\tau_k)$

Table 17.1: Progressive Contracts Used in the Experimental Study

17.1.3 Data Sets

We conducted our experiments using the *de-facto* standard datasets used to stress test skyline algorithms (BKS01). This includes three extreme attribute correlations, namely *independent*, *correlated*, or *anti-correlated*. For correlated data a few tuples dominate a vast majority of tuples in that table. In contrast, for anti-correlated datasets a large portions of the input can potentially correspond to the final skyline results, making skyline operations resource intensive (both memory and CPU). For each data set R (and T), we vary the cardinality N [10K–500K] and the # of skyline dimensions d [2-5]. The attribute values are real numbers in the range [1–100]. The join selectivity σ is varied in the range $[10^{-4}$–$10^{-1}]$. We set $|R| = |T| = N$.

17.1.4 Query Workload

We focus on queries similar to the motivating examples in Chapter 1.2. That is, queries that perform join, mapping and then skyline operations. More specifically, we consider queries that differ in their skyline dimensions. Each workload query is assigned a **query priority** pr_i [1 – 0] that classifies the queries into HIGH [1 – 0.7], MEDIUM [0.69 – 0.4] and LOW [0.39, 0] priority.

17.1.5 Competitor Techniques

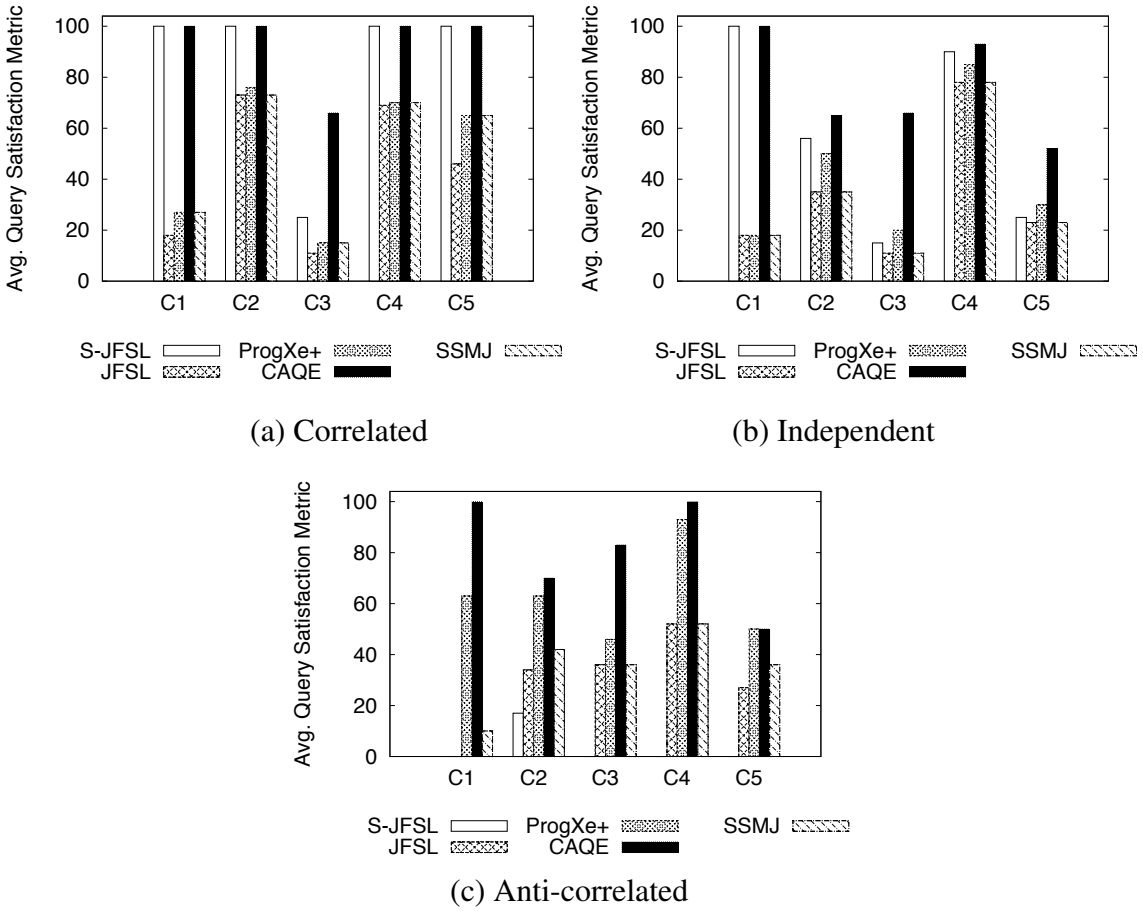
To the best of our knowledge, CAQE is the first technique to support the Contract-MQP. To showcase the effectiveness of CAQE, we compared against existing skyline-over-join algorithms, namely *JFSL* (KLTV06), *Skyline-Sort-Merge-Join* (*SSMJ*) (JMP$^+$10) and ProgXe (RR10). In all systems, queries are processed in the order of the priority pr_i but these existing techniques do not share work across skyline queries. To compare CAQE against sharing-based technique, we propose a **shared skyline approach** (S-JFSL) that pipelines the join tuples over our min-max cuboid plan (see Chapter 13).

17.1.6 Evaluation Metrics

To examine the performance of CAQE we vary the: (1) contract model used, (2) query priorities, (3) data distributions, and (4) number of workload queries. For a given workload and its associated contract model, we measure: (1) the latency and the utility of each result tuple, (2) the total execution time to return the complete result set, (3) total memory usage (number of join results), (4) CPU computations by means of the number of skyline comparisons needed. Lastly, we calculate the average satisfaction metric of each workload query.

17.2 Contract Satisfaction Metric

We now analyze the performance of the different algorithms under varying contract and data distribution models. In this set of experiments, we vary the priority of the different queries such that for contract models $\{C1, C2\}$, in Figure 17.1 and Figure 17.3, queries with a larger number of skyline dimensions have a higher priority than queries with a smaller dimensions. In contrast, for $\{C3, C4\}$ we assigned queries with smaller number of skyline dimensions a higher priority. Lastly, for $C5$ priorities were uniformly assigned.



(a) Correlated

(b) Independent

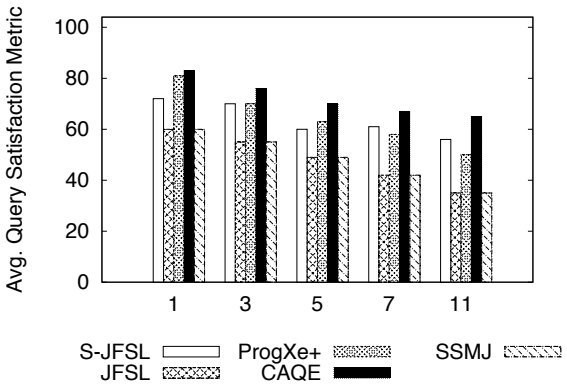
(c) Anti-correlated

Figure 17.1: Comparing the Avg. Query Satisfaction Metric for CAQE, S-JFSL, JFSL, ProgXe+ and SSMJ; $|\mathcal{S}_Q| = 11$ $N = 500K$

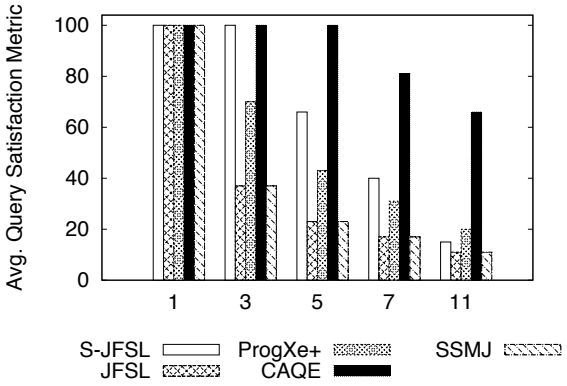
Correlated datasets are tailor made for skyline algorithms since a handful of join tuples can dominate the entire result space. Therefore we set the contract parameters t_{C1} = t_{C3} = $10s$ and $n_{i,j} = 1s$. In Figure 17.1.a we observe that for contracts $\{C1, C3, C4, C5\}$ CAQE and S-JFSL both exploit the sharing opportunity provided by our min-max cuboid plan to progressively output the dominating tuples early on. They also exploit it to prune vast amounts of intermediate tuples. For these same contracts, existing techniques return tuples that have at worst 4x smaller utility score ($C1$) than CAQE and at best have 1.5x smaller utility score. Contract $C3$ is our toughest requirement to meet, for instance a tuple with a time stamp 12 seconds has utility of 0.5. Even under such stringent contract

requirements, CAQE's contract-driven ordering technique allows us to meet a satisfaction metric of 66% which is approximately 4x better than both ProgXe+ and SSMJ, and 3x better than the shared execution strategy S-JFSL.

For independent 4-d dataset, several hundreds of join tuples contribute to the final skyline rather than the mere 16 tuples produced in the correlated dataset. Accordingly we set t_{C1} = t_{C3} = $40s$. Cardinality-based contract $C4$ requires 10% of the tuples to be progressively produced every 10s ($n_{i,j}$). Under this model, the performance of the count based ProgXe+ algorithm is comparable to CAQE's satisfaction based metric. In Figure 17.1.a for contracts $\{C2, C3, C5\}$ the contract-driven, rather than count-driven, approach of CAQE enables it to perform 2.5x better than the others.



(a) Contract Model: C2



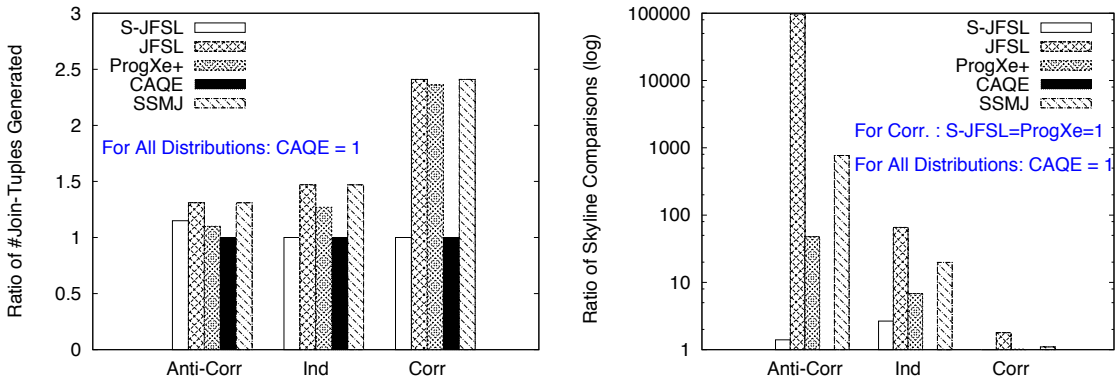
(b) Contract Model: C3

Figure 17.2: Increasing Number of Queries in the Workload

The anti-correlated data distribution is the most resource intensive dataset for a skyline algorithm. This is evident from the fact that a 4-d skyline has 75K+ join tuples in the final skyline. Given the expensive nature of this dataset we set t_{C1} = t_{C3} = 30 minutes and $n_{i,j}$ = 10 minutes. For all contract models except $C2$, skyline results returned by JFSL have no value to users of the workload queries. In Figure 17.1.c we observe that both CAQE and ProgXe+ outperform the other techniques by a factor of ≈2x. For time-based contracts such as $\{C1, C3\}$ CAQE returns skyline results that have 1.5x and 1.8x better utility than that of ProgXe+. For contracts $\{C4, C5\}$ when smaller dimensional skyline queries have higher priority, ProgXe+ is competitive with CAQE. However, when the higher dimensional queries are of more importance, then CAQE outperforms ProgXe+ by ≥ 1.5x.

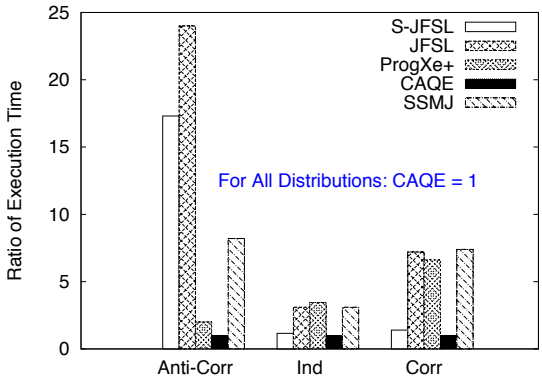
17.3 Increasing Size of Workload

We now measure the effectiveness of the compared techniques for varying workload sizes. In this section, we focus the discussion to the general independent data distribution datasets and to contracts $\{C2, C3\}$ which are the strictest contract models presented in Table 17.1 (Chapter 17.1). In Figure 17.2 as the number of workload queries increases the average satisfaction of each query in the workload drops proportionally. In Figure 17.2.a, for all workload sizes due to the nature of the logarithmic decay function contract $C2$ none of the techniques can achieve the optimal 100% satisfaction. As the workload size increases, the adaptive execution strategy enables CAQE to the smallest drop of 20% in comparison to the 36% and 38% drop in performance for the ProgXe+ and SSMJ respectively. In contrast in Figure 17.2.b for the contract model $C3$ all compared techniques exhibit optimal query satisfaction when only handling a single query in the workload. However, as the number of queries increases we observe in Figure 17.2.b



(a) Join Tuples Generated

(b) Skyline Comparisons Conducted



(c) Total Execution Time

Figure 17.3: Comparing the Statistics Measured for S-JFSL, JFSL, ProgXe and SSMJ Against CAQE ($|\mathcal{S}_Q| = 11, N = 500K, C2$)

alternative techniques suffer from a step drop in performance. In contrast, the adaptive execution strategy employed enables CAQE to only experiences a relatively smaller drop in query satisfaction.

17.4 Comparing CPU and Memory Utilization

The CPU and memory utilization of the skyline-over-join algorithm are directly related to the number of intermediate tuples generated by the join operation as well as the expensive pairwise dominance comparison needed to evaluate the final skyline. In Figures 17.3.a- 17.3.c we illustrate that the by employing a shared execution approach enables

both CAQE and S-JFSL to produce fewer join tuples than their competitors. In fact, for the independent dataset CAQE generates 31% fewer join results than both JFSL and SSMJ and 16% fewer than ProgXe+.

In terms of skyline comparisons, the contract-driven processing of join results over the min-max cuboid plan empowers CAQE to deliver skyline results earlier than its competitors while having to perform several fold fewer pairwise skyline comparisons. In particular as shown in Figure 17.3.b, for independent datasets CAQE requires 66x, 2.7x, 7x, and 20x fewer comparisons than the JFSL, S-JFSL, ProgXe+, and SSMJ techniques respectively. By generating a smaller number of join tuples as well as performing fewer dominance comparisons CAQE is able to outperform the compared techniques in the overall execution time of the query workload. In fact, CAQE is at least 2x faster than ProgXe+, and $\approx$ 24x better than JFSL. Lastly, CAQE outperforms the shared execution strategy S-JFSL by 17x.

18

Related Work for Part III

	Skyline-Over Join	Multiple Queries	Progressive	Supports User QoS
SkyCube (YLL$^+$05)		✓	✓	
BUS, TDS(PJET05)		✓	✓	
PruningJoin$^+$ (KLTV06)	✓			
SAJ(KLTV06)	✓			
Sort-Based (KML11, VDP11, JMP$^+$10)	✓		✓	
ProgXe (RR10)	✓		✓	
✠ **Our Approach**	✓	✓	✓	✓

Table 18.1: Summary of Related Work for Part III (CAQE)

18.1 Subspace Skylines over Single Relation

(PJET05, YLL$^+$05) proposed *Skyline Cube* containing skylines results for all combinations of skyline dimensions. This is reminiscent of the precomputed *data cube* technique in data warehousing (GBLP96). However, these techniques ignore (1) multi-relational skylines, and (2) do not support QoS sensitive query evaluation – both now tackled by our work.

18.2 Skylines over Join Queries

Existing techniques (BKS01, RR10, JMP+10, KML11, VDP11) process a *single* skyline-over-join query, while ours is the first effort at processing multiple skyline-over-join query. Table 18.1 summarizes the differences between our approach versus the state-of-the-art skyline techniques. In Chapter 5 we provide a detailed descriptions of the above mentioned techniques.

18.3 Quality of Service

In *computer networking*, QoS defines varying levels of services for applications and types of data. Applications such voice over IP and streaming multimedia must ensure a certain level of user experience by reducing packet drops. This is accomplished by reserving network capacity based on bandwidth, delay, and error rates (PL07). In *streaming databases*, to provide real-time responses, (LQX06, XZH05) enable the user to specify a *contract* in terms of latency, data freshness, CPU and memory usage. Their focus is different from ours in the complexity of the queries targeted, the objective and the approach taken. First, (XZH05) sheds data from incoming streams to handle load and meet the desired QoS. Second, they only consider simple stream queries namely Select-Project-Join. They do not support the more complex nor blocking queries such as skyline-over-join queries.

Part IV

Cardinality Assurance Via Proximity-driven Refinement

19

Proximity-Driven Cardinality Assurance

In the following chapters, we address the problem of *Proximity-Driven Cardinality Assurance*, introduced in Chapter 1.4.4. In this context, we consider conjunctive Select-Project-Join (SPJ) queries of the form $Q = P_1 \wedge \ldots \wedge P_d$, where P_i's denote predicates spanning relations $R_1 \ldots R_k$ in database $\mathcal{D}$.

19.1 Running Query

In this work, we will use query Q_2 below as our running example. For simplicity, Q_2 contains only one select and one join predicate.

```
Q2: SELECT * FROM A, B WHERE A.x=B.x AND B.y<50
```

19.2 Query Representation

For a given query Q, we divide each predicate P_i into two parts: the *predicate function* ($P_i^{\mathcal{F}}$) and the *predicate interval* ($P_i^{\mathcal{I}}$). $P_i^{\mathcal{F}}$ is a monotonic function on attributes of $R_1 \ldots R_k$ while $P_i^{\mathcal{I}}$ denotes the interval of acceptable values for $P_i^{\mathcal{F}}$, that is, $P_i^{\mathcal{I}} = (min_i^{\mathcal{I}}, max_i^{\mathcal{I}})$. To illustrate, the predicate $(B.y < 50)$, in Q_2 is decomposed into $P_i^{\mathcal{F}} = B.y$ and $P_i^{\mathcal{I}} = (0, 50)$. Range predicates like $(10 < B.y < 50)$ are rewritten as two one-sided predicates, $(B.y > 10) \wedge (B.y < 50)$. For equi-joins ($A.x = B.x$) and non-equi joins ($2 * A.x < 3 * B.x$), the form of $P_i^{\mathcal{I}}$ is unchanged; however, $P_i^{\mathcal{F}}$ takes the form $Diff((P_i^{\mathcal{F}})_1, (P_i^{\mathcal{F}})_2)$, where $(P_i^{\mathcal{F}})_1$ and $(P_i^{\mathcal{F}})_2$ are separate predicate functions and $Diff$ is the function measuring distance between them. Therefore, join predicate $A.x = B.x$ in Q_2 is decomposed into $(P_i^{\mathcal{F}})_1 = A.x$ and $(P_i^{\mathcal{F}})_2 = B.x$. Its $P_i^{\mathcal{I}} = (0, 0)$ signifies that values of the two functions must match exactly.

19.3 Measuring Refinement

A query Q=$(P_1 \wedge \ldots \wedge P_d)$ is refined to Q' by refining one or more predicates $P_i \in Q$ to predicates $P_i' \in Q'$. The refinement of Q' along P_i, called **predicate refinement score** or $PScore_i(Q, Q')$, is measured as the percent departure of $(P_i^{\mathcal{I}})'$ from $P_i^{\mathcal{I}}$ (Equation 19.1). For equality join predicates, the denominator is set to 100. Measuring relative change, as opposed to absolute change, in predicate intervals compensates for the differing scales of query attributes. While percent refinement is the default predicate refinement metric used in this work, a user can override the metric with custom (monotonic) functions without changes to our algorithm. By computing the refinement score for each query predicate, a refined query Q' can be represented as a d-dimensional vector of predicate refinement scores, called the **predicate refinement vector** or $\overline{PScore(Q, Q')}$ (Equation 19.2).

$$PScore_i(Q, Q') = \frac{|(P_i^{\mathcal{I}})_{min} - (P_i^{\mathcal{I}})'_{min}| + |(P_i^{\mathcal{I}})_{max} - (P_i^{\mathcal{I}})'_{max}|}{|(P_i^{\mathcal{I}})_{max} - (P_i^{\mathcal{I}})_{min}|} \cdot 100 \quad (19.1)$$

$$\overline{PScore(Q, Q')} = (PScore_1(Q, Q') \ldots PScore_d(Q, Q')) \quad (19.2)$$

The **query refinement score** of Q', denoted by $QScore(Q, Q')$ is defined as a monotonic function $f : \mathcal{R}^d \rightarrow \mathcal{R}$ used to measure the magnitude of $\overline{PScore(Q, Q')}$. We use the popular weighted vector p-norms (Kol08) to calculate $QScore(Q, Q')$, though other norms can be also be plugged in.

Weighted Vector P-norms. Given a vector $\overline{v}$, the p-norm or L_p norm of $\overline{v}$, denoted by $\| \overline{v} \|_p$, is defined by Equation 19.3. A weighted norm is a norm assigning weights to the individual components of the vector $\overline{v}$ through a diagonal matrix W.

$$\| \overline{v} \|_p = \Big(\sum_{i=1}^{d} | v_d^p | \Big)^{1/p} \quad (19.3)$$

$$\| \overline{v} \|_{W_p} = \| W\overline{v} \|_p \quad (19.4)$$

$$\| \overline{v} \|_\infty = \max_{1 \leq i \leq d} | v_i | \quad (19.5)$$

Equations 19.6-19.9 show $QScore$ for the common vector norms.

$$L_1: \ QScore(Q, Q') = \Big(\sum_{i=1}^{d} PScore_i(Q, Q') \Big) \tag{19.6}$$

$$L_2: \ QScore(Q, Q') = \Big(\sum_{i=1}^{d} PScore_i(Q, Q')^2 \Big)^{1/2} \tag{19.7}$$

$$L_{W_1}: \ QScore(Q, Q') = \Big(\sum_{i=1}^{d} w_i \cdot PScore_i(Q, Q') \Big) \tag{19.8}$$

$$L_\infty: \ QScore(Q, Q') = \max_{1 \le i \le d} PScore_i(Q, Q') \tag{19.9}$$

Example 19.1 *Consider the following refinement to Q_2.*

```
Q2': SELECT * FROM A,B WHERE A.x=B.x AND B.y<60
```

The refined query Q'_2 expands the ranges of acceptable values for $B.y$ by 10 units. Therefore, Q'_2 is represented as $\overline{PScore(Q_2, Q'_2)} = (0, \frac{10}{50} \cdot 100)$ and has $QScore(Q_2, Q'_2)$=20 for the L_1 norm.

We can similarly define refinement scores for tuples. The refinement score of a tuple quantifies how much the original query must be refined to produce it as a result.

Example 19.2 *A join tuple τ $\{(A.x$=10010), $(B.x$=10015, $B.y = 55)\}$ is represented by the refinement vector (5, $\frac{5}{50} \cdot 100$) to indicate that the join and select predicates must be refined by 5 and 10 units respectively to produce τ as a result. The total refinement of τ is 15 for L_1, $5\sqrt{5}$ for L_2 and 10 for L_∞.*

Theorem 19.1 *Given a refined query Q' for Q and a tuple τ expressed in terms of their predicate refinements respectively as $(u'_1, u'_2, \ldots u'_d)$ and $(t_1, t_2, \ldots t_d)$. τ satisfies query Q' if and only if $0 \le t_i \le u'_i \ \forall \ i$.*

Proof: Proof omitted due to space limitations. ■

19.4 Problem Definition

Definition 19.1 Exact Proximity-driven Cardinality Assurance *(E-PCA). Given database $\mathcal{D}$, query Q and desired cardinality $\mathcal{C}$, E-PCA finds a set $\mathbf{Q}_F$ of refined queries s.t. $\forall Q'_i \in \mathbf{Q}_F$ (1)* **Cardinality Constraint:** *Cardinality$(Q'_i) = \mathcal{C}$, and (2)* **Proximity Constraint:** $QScore(Q, Q'_i) = min\{QScore(Q, Q'_j)| \forall Q'_j$ *s.t. (Cardinality(Q'_j) =* $\mathcal{C})\}$.

In many scenarios, however, no query can attain the exact cardinality $\mathcal{C}$ and even an exhaustive search cannot find a single query producing the cardinality $\mathcal{C}$. Hence, to make the problem tractable, we augment the problem with two tunable thresholds: cardinality threshold δ and refinement threshold γ.

Definition 19.2 Approximate Proximity-driven Cardinality Assurance *(A-PCA). Given database $\mathcal{D}$, query Q, desired cardinality $\mathcal{C}$, cardinality threshold δ and refinement threshold γ, A-PCA finds a set $\mathbf{Q}_F$ of queries s.t. $\forall Q'_i \in \mathbf{Q}_F$ (1)* **Cardinality constraint***: |Cardinality(Q'_i)$-\mathcal{C}|\leq\delta$, and (2)* **Proximity constraint***: $|QScore_{opt} - QScore(Q, Q'_i)| \leq\gamma$, where $QScore_{opt}$= min $\{QScore(Q, Q'_j)| \forall Q'_j$ s.t. $(|$Cardinality$(Q'_j)-\mathcal{C}| \leq \delta)\}$. For simplicity, we henceforth use the term PCA interchangeably with A-PCA.*

Theorem 19.2 *For a conjunctive query Q with desired cardinality $\mathcal{C}$, cardinality threshold δ and a refinement threshold γ, the PCA problem is NP-hard.*

Proof: We prove the NP-hardness of PCA by a simple reduction to the *approximate cardinality assurance* ($A - CA$) problem which is known to be NP-hard (BCT06). First, we recast PCA as a decision problem: Given database $\mathcal{D}$, query Q, expected cardinality $\mathcal{C}$, cardinality threshold δ, and refinement threshold γ, does there exist a set of queries satisfying constraints (1) and (2) of Definition 2? Similarly, $A - CA$ takes the following form: Given database $\mathcal{D}$, query Q, expected cardinality $\mathcal{C}$, and cardinality threshold δ, does there exist a set of queries satisfying constraint (1) from Definition 2? The $A -$

CA problem is known to be NP-hard. Next, we define a procedure *reduce*, which given an instance of $A - CA$, creates an equivalent PCA instance with $\gamma = \infty$. *Reduce* is therefore (trivially) polynomial, and any solution of $A - CA$ solves the corresponding PCA problem and vice versa. Since the approximate cardinality problem $A - CA$ is NP-hard, and $A - CA$ can be reduced to PCA via a polynomial time algorithm, PCA is NP-hard. ■

In this work, we first focus our attention on queries that return *too few* results. We show in Chapter 23 that CAPRI can be extended to handle queries that produce *too many* results.

20

Phase I: Expand

As described in the previous section, the *Expand* phase of CAPRI is responsible for iteratively generating refined queries that meet two criteria: (1) they satisfy the proximity threshold, and (2) their refinement scores ($QScore$ values) are greater or equal to the scores of previously generated queries.

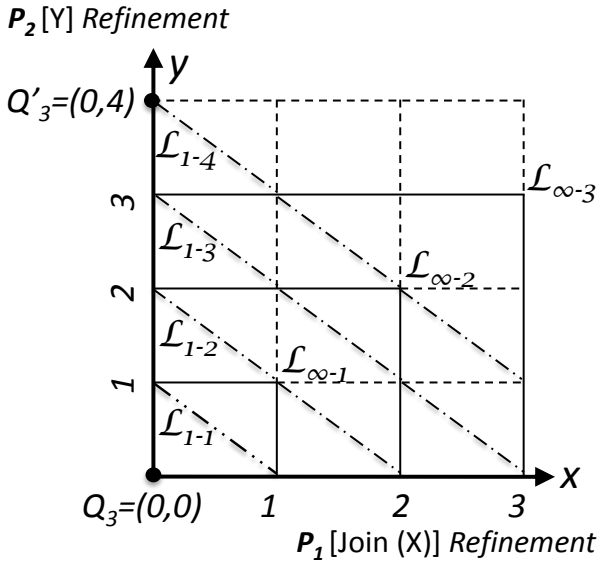


Figure 20.1: Expand Phase: Refined Space and Generation of Refined Queries

To meet the above query generation goals, CAPRI uses an abstraction called the **Refined Space** to represent all refined queries. Given an original query Q having d predi-

cates, the *Refined Space*, denoted henceforth by $RS(Q)$, is a d-dimensional space, where the origin represents Q and the axes measure individual predicate refinement. To illustrate, consider a refined query Q' and assume that the L_p norm is used to compute $QScore$. Q' would then be represented in $RS(Q)$ as $(u_1, u_2 \ldots u_d)$ where $u_i = (PScore_i(Q, Q'))^p$ $\forall i = 1 \ldots d$, making $QScore(Q, Q')$ = $(\sum_{i=1}^{d} u_i)^{\frac{1}{p}}$. Conversely, every point in the refined space $(u_1, u_2 \ldots u_d)$ corresponds to some query Q' with $PScore_i(Q, Q') = u_i^{1/p}$. By extension, any d-dimensional hyper-rectangle on $RS(Q)$ also corresponds to a query.

Theorem 20.1 *Given original query Q and optimal query Q_{opt} meeting the cardinality constraint and having minimum refinement. Let $RS(Q)$ be a multi-dimensional grid with step-size on each axis equal to $\frac{\gamma}{d}$, then at least one refined query Q' lying on the $RS(Q)$ grid will satisfy the proximity constraint w.r.t. to Q_{opt}.*

Proof: Let $Q_{opt} = \{u_1, u_2 \ldots u_d\}$ in a grid cell G in $RS(Q)$. Since the refined space grid has step-size $\frac{\gamma}{d}$, any query $Q' = \{u'_1, u'_2 \ldots u'_d\}$ on G satisfies:

$|{u_1}^p - {u'_1}^p|$ + $|{u_2}^p - {u'_2}^p|$ + ... + $|{u_d}^p - {u'_d}^p| \leq \frac{\gamma}{d} \cdot d = \gamma$

$\Rightarrow |({u_1}^p + {u_2}^p + \ldots + {u_d}^p) - ({u'_1}^p + {u'_2}^p + \ldots + {u'_d}^p)| \leq \gamma$

$\Rightarrow |QScore(Q_{opt}, Q)^p - QScore(Q', Q)^p| \leq \gamma$

$\Rightarrow QScore(Q_{opt}, Q)^p - QScore(Q', Q)^p \leq \gamma$ (assume $QScore(Q_{opt}, Q)^p \geq QScore(Q', Q)^p$)

$\Rightarrow (QScore(Q_{opt}, Q) - QScore(Q', Q)) \cdot (QScore(Q_{opt}, Q)^{p-1} + QScore(Q_{opt}, Q)^{p-2} \cdot$

$QScore(Q', Q) + \ldots + QScore(Q', Q)^{p-1}) \leq \gamma$

$\Rightarrow (QScore(Q_{opt}, Q) - QScore(Q', Q)) \leq \gamma$ ($\gamma > 1$) ■

CAPRI divides $RS(Q)$ into a multi-dimensional grid with step-size $\frac{\gamma}{d}$ to avoid an exhaustive search of $RS(Q)$ and stay within the proximity threshold. Each query on the multi-dimensional grid is called the *grid query*. Figure 20.1 depicts the refined space abstraction for running query Q_2 assuming γ=10. Since Q_2 has two predicates, step-size=5 and $RS(Q_2)$ is a 2-dimensional space with the axes respectively measuring

the refinements along the select and join predicates. A refined query like Q_2' having $\overline{PScore(Q_2, Q_2')} = (0, 20)$ is represented as (0, 4) in $RS(Q_2)$.

The second goal of the *Expand* phase is to generate refined queries in order of increasing refinement. CAPRI achieves this goal by producing queries close to the origin in $RS(Q)$ before those far from it. In particular, the *Expand* phase generates refined queries in layers where queries in a given *query-layer* have the same $QScore$. Consequently, for all L_p norms except L_∞, query-layers take the form of d-dimensional planes corresponding to $QScore = k \Rightarrow QScore^p = k^p \Rightarrow (\sum_{i=1}^{d} u_i') = k^p$. For L_∞, however, query-layers are L-shaped and intersect each axis at k^p. Figure 20.1 shows query-layers for Q_3 assuming the L_1 and L_∞ norms.

Beginning with the query-layer with refinement 0, CAPRI generates all grid queries in the current query-layer. If no query from the current layer produces adequate results, CAPRI proceeds to the next query-layer having $QScore$ increased by $\frac{\gamma}{d}$. Since this iterative expansion model examines queries in order of increasing refinement, CAPRI can stop immediately after a query is found to meet the required cardinality, thus reducing the number of queries examined by CAPRI. Algorithms 12 and 13 respectively describe the pseudocode for generating queries using the L_p and L_∞ norms. The L_p algorithm generates query-layers using a breadth-first search while the L_∞ norm sequentially enumerates queries in the given layer.

Algorithm 12 GetNextQuery(Queue *queryQue*)

```
1: int[] currQuery = queryQue.Pop()  //Array indexed from 1
2: for i = 1 to d do
3:    nextQuery ← GetNextNeighbor(i)  //Increment the (i)^th dimension of currQuery
      by stepsize
4:    if (!queryQue.Contains(nextQuery)) then
5:       queryQue.Push(nextQuery)
6: return currQuery
```

Algorithm 13 GetNextQuery(Queue queryQue, int currRef)

```
1: if (!queryQueue.Empty()) then
2:    return queryQue.Pop()
3: else
4:    Query newQuery = 0
5:    for i = 1 ... d do
6:       newQuery[i] = currRef; queryQue.Push(newQuery)
7:       while newQuery != null do
8:          IncrementQuery(newQuery, i, currRef) // Enumerate queries with i-th dimension fixed at currRef all other dimensions < currRef
9:          queryQue.Push(newQuery)
```

Theorem 20.2 *A grid query Q'_i with $QScore(Q, Q'_i) = k$ is investigated after all grid queries with $QScore(Q, Q'_i) = (k-1)$ have been investigated.*

Proof: Consider the refined space to be a directed graph with the origin as the root and every grid query as a node. Every grid query is connected to d queries obtained by incrementing one dimension by the unit step-size. These connections form the graph's edges. Then $GetNextQuery$ for the L_p norm performs a breadth-first search on the refined space grid, guaranteeing that all queries at distance $k-1$ from the root are investigated before those at distance k. The result is trivially true for L_∞ norm since our algorithm explicitly generates queries in each query layer. ■

Time Complexity. The worst case complexity of the *Expand* phase is $\mathcal{O}(V + E)$ where V is maximum number of refined queries in the grid and $|E| = d \cdot |V|$.

21

Phase II: Explore

The *Explore* phase of CAPRI is responsible for efficiently computing the cardinalities of queries produced in the *Expand* phase. For this purpose, we introduce a light-weight query execution methodology based on two key contributions: (1) a novel, efficient incremental query execution algorithm, and (2) an efficient, predictive index structure. The former exploits dependencies between refined queries using a specialized recursive model. For each query, our model requires execution of only one sub-query and computes the overall cardinality by intelligently combining partial results from previously queries. CAPRI guarantees that a query result is examined at most once, irrespective of how many queries contain it. Our predictive index structure, on the other hand, ensures that CAPRI only examines tuples likely to satisfy a query and does not regenerate previously produced query results.

21.1 Incremental Query Execution

The principle underlying our query execution algorithm is that refined queries often share results and therefore, once a query result is found to satisfy a given query, it must never

be re-evaluated for any other query.

Query Containment. A refined query Q'=$(u'_1, u'_2 \ldots u'_d)$ is said to be *contained* within another refined query Q''=$(u''_1, u''_2 \ldots u''_d)$ if $(u'_i \leq u''_i)\ \forall\ i = 1 \ldots d$.

Theorem 21.1 *If refined query Q' is contained within refined query Q'': (1) all results of Q' also satisfy Q''. (2) Q' is guaranteed to be generated before Q'' in the Expand phase.*

Proof: Let tuple τ that satisfies Q'. (1) By Equation 19.2:

$PScore_i(\tau, Q) \leq PScore_i(Q', Q)\ \forall\ i = 1 \ldots d$

$\Rightarrow PScore_i(\tau, Q)^p \leq PScore_i(Q', Q)^p = u'_i\ \forall\ i = 1 \ldots d$ ($PScore \geq 0$)

$\Rightarrow PScore_i(\tau, Q)^p \leq u''_i$

$\Rightarrow PScore_i(\tau, Q) \leq PScore_i(Q'', Q)$.

Consequently, all the query results of Q' also satisfy Q''. For (2), from definition of contained queries, $QScore(Q', Q) \leq QScore(Q'', Q)$. Therefore, by Theorem 20.2, the *Expand* phase will produce Q' before Q''. ■

Since all contained queries are produced and executed before those containing them, CAPRI can extensively use previously-generated query results. In particular, CAPRI exploits the concept of query containment by artificially constructing contained queries, called *sub-queries* henceforth, that are used as units of query execution and result sharing. We now describe the sub-queries used.

21.1.1 Query Decomposition

Consider query Q' with d predicates, represented as point $(u'_1, \ldots, u'_d)$ in the refined space. In addition to Q', CAPRI defines d specialized sub-queries contained in it, giving $d + 1$ queries in all. Figure 21.1 shows these queries for a 2-predicate query. The first sub-query (A) corresponds to the unit square in $RS(Q)$ with its upper-right corner at Q'=(u'_1, u'_2), the second sub-query (B) corresponds to a unit-width rectangle in $RS(Q)$

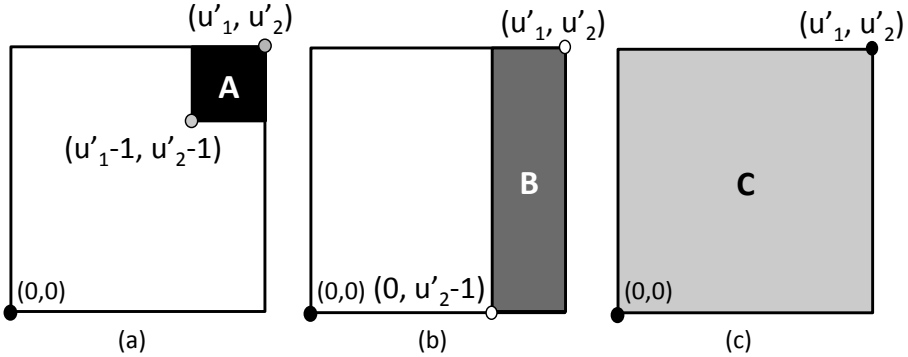


Figure 21.1: Sub-orthotopes of a 2-D Query Orthotope

with Q' at its upper-right corner, and the third sub-query is the entire query (C). Similarly, for a 3-predicate query as in Figure 21.2, the first sub-query (A) is the unit cube, the second (B) is a unit length and width parallelepiped, the third (C) is a unit width parallelepiped, and the fourth (D) is the entire query sub-query. For ease of exposition, we refer to the first sub-query as **cell**, the second as **pillar**, the third as **wall**, and the fourth as **block**.

In a d-dimensional refined space, the $d+1$ sub-queries, called O_1, $O_2 \ldots O_{d+1}$, can be formally defined as shown in Equations 21.1-21.4. All $d+1$ sub-queries have the same upper bound ($Q' = (u'_1 \ldots u'_d)$), but different lower bounds. For instance, the cell sub-query O_1 has a lower bound which is unit length away from $(u'_1, \ldots u'_d)$ on all dimensions (Equation 21.1). The cell sub-query corresponds to the cell in the refined space grid having $(u'_1, \ldots u'_d)$ as its upper bound. Similarly, the pillar sub-query has a lower bound with the first dimension equal to 0 and all remaining dimensions j ($j = 2 \ldots d$) unit length away from u'_j (Equation 21.2). In general, the lower bound of the jth sub-query O_j is $(0, \ldots 0, u'_j - 1, \ldots u'_d - 1,)$. For simplicity, we will refer to an sub-query O_i corresponding to query $(u'_1, \ldots u'_d)$ as $O_i(u'_1, \ldots u'_d)$.

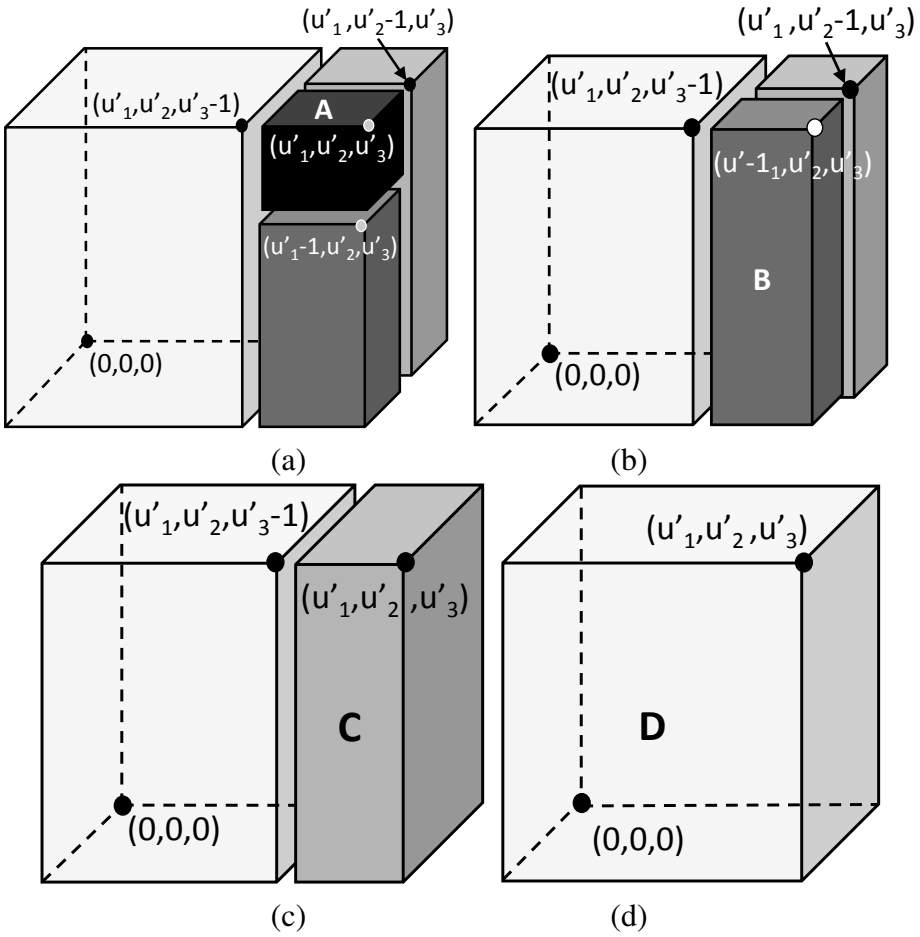


Figure 21.2: Sub-orthotopes of a 3-D Query Orthotope

$$O_1 = ((u'_1 - 1, \ldots, u'_d - 1), (u'_1, \ldots, u'_d)) \tag{21.1}$$

$$O_2 = ((0, u'_2 - 1, \ldots, u'_d - 1), (u'_1, \ldots, u'_d)) \tag{21.2}$$

$$O_j = ((0, 0, ...u'_j - 1, \ldots, u'_d - 1), (u'_1, .., u'_d)) \tag{21.3}$$

$$O_{d+1} = ((0, \ldots, 0), (u'_1, .., u'_d))\} \tag{21.4}$$

By decomposing a query into the sub-queries defined above, we can reuse previously obtained results. To see how this can be done, consider Figure 21.3.a where the 2-D query decomposed into 3 sub-queries. We observe that sub-query A is the $Cell(u'_1, u'_2)$, B is

the $Pillar(u'_1, u'_2 - 1)$, and C is the $Wall(u'_1 - 1, u'_2)$. Similarly, Figure 21.3.b shows the decomposition of a 3-predicate query into the four sub-queries A, B, C and D which are respectively the $Cell(u'_1, u'_2, u'_3)$, the $Pillar(u'_1 - 1, u'_2, u'_3)$, the $Wall(u'_1, u'_2 - 1, u'_3)$, and the $Block(u'_1, u'_2, u'_3 - 1)$. In general, a d-predicate query can be decomposed into the previously defined $(d + 1)$ sub-queries:

$$\mathbf{2 - predicate\ Query}: \tag{21.5}$$

$$O_3(u'_1, u'_2) = O_1(u'_1, u'_2) + O_2(u'_1 - 1, u'_2) + O_3(u'_1, u'_2 - 1)$$

$$\mathbf{3 - predicate\ Query}: \tag{21.6}$$

$$O_4(u'_1, u'_2, u'_3) = O_1(u'_1, u'_2, u'_3) + O_2(u'_1 - 1, u'_3, u'_3) + O_3(u'_1, u'_2 - 1, u'_3) + O_4(u'_1, u'_2, u'_3 - 1)$$

$$\mathbf{d - predicate\ Query}: \tag{21.7}$$

$$O_{d+1}(u'_1, u'_2, \ldots, u'_d) = O_1(u'_1, u'_2, \ldots, u'_d) + O_2(u'_1 - 1, u'_2, \ldots, u'_d) + O_3(u'_1, u'_2 - 1, u'_3 \ldots u'_d) + \ldots + O_{d+1}(u'_1, u'_2, \ldots, u'_d - 1)$$

Thus, if the cardinalities for the $(d + 1)$ sub-queries have been pre-computed, the cardinality of query Q' is the mere addition of these sub-cardinalities. We must store only the cardinality *values* for the $d + 1$ sub-queries. The corresponding result tuples can either be stored in main memory or paged to disk. The above sub-query decomposition also leads to two crucial observations: (1) **The only part of a query unique to itself is the cell**; all remaining parts of the sub-query are shared with other queries. (2) **The** $d+1$ **sub-queries defined above belong to queries completely contained in** Q'. Therefore, Theorem 21.1 guarantees that these queries would have been produced and hence executed before in-

vestigating Q'. As a consequence, CAPRI must only execute the cell sub-query and can directly reuse cardinalities of the remaining sub-queries.

21.1.2 Recursive Cardinality Computation

Query decomposition assumes that the cardinalities of the $d + 1$ sub-queries have already been computed. But independently determining cardinalities of these sub-queries is redundant. Instead, we present a recursive strategy to calculate the cardinalities of the sub-queries in constant time.

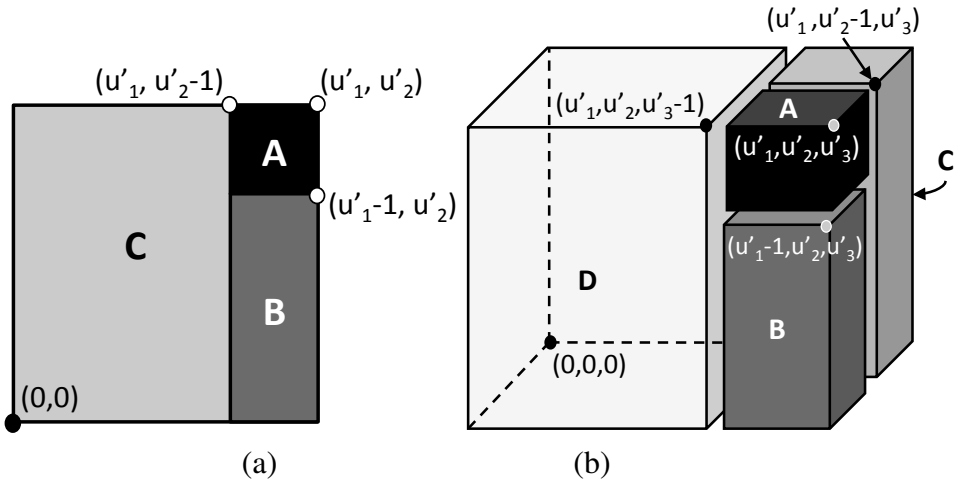


Figure 21.3: Orthotope Decomposition: (a) 2-D (b) 3-D

Reconsider Figure 21.3 focusing now on the relationship between sub-queries. We observe that for 2-predicate sub-queries (Figure 21.3.a) the $Pillar(u'_1, u'_2)$ is equivalent to $Cell(u'_1, u'_2)$ and $Pillar(u'_1\text{-}1, u'_2)$ combined. Similarly, the $Wall(u'_1, u'_2)$, which is the entire query is equal to the sum of $Pillar(u'_1, u'_2)$ and $Wall(u'_1, u'_2 - 1)$. For the 3-predicate query, in Figure 21.3.b, we have three similar recurrences as shown below.

$$\mathbf{2-Recurrences}: \tag{21.8}$$

$$Pillar(u'_1, u'_2) = Cell(u'_1, u'_2) + Pillar(u'_1 - 1, u'_2)$$

$$Wall(u'_1, u'_2) = Pillar(u'_1, u'_2) + Wall(u'_1, u'_2 - 1) \tag{21.9}$$

$$\mathbf{3-Recurrences}: \tag{21.10}$$

$$Pillar(u'_1, u'_2, u'_3) = Cell(u'_1, u'_2, u'_3) + Pillar(u'_1 - 1, u'_2, u'_3)$$

$$Wall(u'_1, u'_2, u'_3) = Pillar(u'_1, u'_2, u'_3) + Wall(u'_1, u'_2 - 1, u'_3) \tag{21.11}$$

$$Block(u'_1, u'_2, u'_3) = Wall(u'_1, u'_2, u'_3) + Block(u'_1, u'_2, u'_3 - 1) \tag{21.12}$$

In general, this recursion for a d-predicate query is:

$$O_i(u'_1, \ldots, u'_d) = O_{i-1}(u'_1, \ldots, u'_d) + \tag{21.13}$$
$$O_i(u'_1, u'_2, \ldots, u'_{i-1} - 1, \ldots, u'_d)\ where\ i = 2 \ldots d + 1$$

Since the sub-query O_1 has no recurrences, its cardinality must be computed by executing the query. However, once the cardinality of O_1 is determined, it takes d (constant) steps to calculate the total cardinality for query Q'.

21.1.3 Cardinality Computation Algorithm

Algorithm 14 takes as input the query $Q'(u'_1, \ldots, u'_d)$ being investigated and produces its cardinality. For this, Algorithm 14 first computes the cardinality of $Cell(u'_1, \ldots, u'_d)$, and then iteratively applies the recurrence in Equation 21.13 to compute cardinalities of the remaining sub-queries. Algorithm 15 is used to compute cell cardinality. Based on

the input query, Algorithm 15 finds the bounds of the cell sub-query and determines the set of unmaterialized output regions that overlap with it. These output regions are then materialized to calculate the cell cardinality.

Algorithm 14 ComputeCardinality(Query *currQuery*, int d)

```
1: int[d + 1] currCard  //All arrays are indexed from 1
2: currCard[1] = ComputeCellCardinality(currQuery)
3: for i = 2 to d + 1 do
4:     prevQuery ← GetPreviousNeighbour(i-1)  //decrement the (i − 1)^th dimension of
       currQuery by step-size
5:     int[] prevCard = GetAllCardinalities(prevQuery)
6:     currCard[i] = currCard[i − 1] + prevCard[i]
7: StoreAllCardinalities(currQuery, currCard)
8: return currCard[d + 1]
```

Algorithm 15 ComputeCellCardinality(int[] *currQuery*)

```
1: List<OPRegion> opRegs = GetOverlap(GetCell(currQuery))
2: List<ResultTuples> tuples = GetResultTuples(opRegs)
3: for t in tuples do
4:     boundingCell = GetBoundingCell(t)
5:     IncrementCellCardinality(boundingCell)
6: RemoveMaterializedRegions(opRegs)
7: return GetCellTupleCount(currQuery)  //Get number of tuples lying in the unit cell
   with currQuery as upper bound
```

21.2 Predictive Index Structure

The incremental query execution algorithm described above ensures that CAPRI can perform extensive result sharing between refined queries and that each individual query requires the execution of only the unit cell sub-query. The predictive index structure described below in turn ensures that the unit cell sub-query is executed efficiently by only examining those results that (1) are likely to satisfy a query and (2) haven't been examined before. CAPRI achieves these twin goals through the strategy of mapping potential

results to queries at a coarse granularity and keeping a running count of the cardinality of previous queries. We illustrate the construction of our index structure via the running query Q_2 and an L_1 norm.

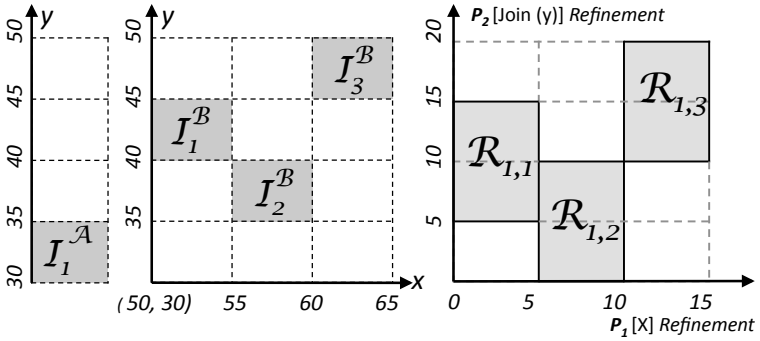


Figure 21.4: Building Output Regions: (a) Table A (b) Table B (c) Output Regions in Refined Space

The first step in mapping potential results to queries is to partition input tables along the attributes included in the original query (or equivalently use existing multi-dimensional indexes). Therefore, as depicted in Figure 21.4, Table A is partitioned along $A.y$, while B is partitioned along attributes $B.x$ and $B.y$. To cope with skewed data, if a multi-dimensional partition is found to have more than c tuples, the partition is divided into half along each dimension. The resulting input partitions are identified by their lower and upper bounds.

21.3 CAPRI: Putting It All Together

Algorithm 16 presents the pseudo code for the CAPRI framework. Given an initial query Q and the refinement threshold γ, begins to iteratively *Expand* and *Explore* refined queries, starting at the origin of the refined space and sequentially traverses queries in subsequent layers. For each refined query, CAPRI estimates its cardinality following the Incremental Query Execution technique described in Algorithm 14. Once the query

cardinality has been determined, it is compared to the expected cardinality, $\mathcal{C}$. If the cardinality is within the cardinality threshold δ of $\mathcal{C}$, the query is stored in the answer list. In this case, query search terminates with the exploration of all queries in the current layer, i.e., all alternate queries with the same refinement score. If all queries in a layer undershoot the expected cardinality by more than δ, CAPRI explores the next higher layer. Last, if any query overshoots the expected cardinality by more than δ, we repartition the cell corresponding to the given query. The cell is divided into ϑ segments along each dimension (ϑ is a tunable parameter) to examine queries lying within it. Since the result tuples potentially contributing to these new queries have already been generated, CAPRI can directly jump to cardinality estimation. We repeat the repartitioning process for b iterations (b is a tunable parameter). If, at the end of repartitioning, no query is found to satisfy the cardinality constraint, CAPRI returns the query attaining the closest cardinality.

Algorithm 16 CAPRI(Query *origQuery*, int $\mathcal{C}$, int δ, double γ)

```
1: A =[]  //Set of refined Queries
2: Queue queryQueue = []   //Data structure for traversal
3: d ← Soft predicates in origQuery
4: ConstructRefinedSpace(origQuery, γ, d)
5: int[d] currQuery = {0,..., 0}  //Origin represents origQuery
6: queryQueue.push(currQuery)
7: int minRefLayer = MAX_INTEGER_VALUE
8: int currRefLayer = 0
9: while (currRefLayer ≤ minRefLayer) do
10:    int card = ComputeCardinality(currQuery, d)   //Algorithm 14
11:    if ( | card - C | ≤ δ) then
12:       A.add(currQuery)
13:       minRefLayer = currRefLayer
14:    else if ( card > C) then
15:       A.add(Repartition(currQuery))
16:    currQuery = GetNextQuery(queryQueue)  //Algorithm 12
17:    currRefLayer = QScore(currQuery)
18: return A
```

22

Experimental Evaluation of CAPRI

22.1 Experimental Setup

22.1.1 Platform

All algorithms were implemented in Java 1.5. Measurements were obtained on AMD 2.6GHz Dual Core CPUs, Java HotSpot 64-Bit Server VM and Java heap of 2GB.

22.1.2 Evaluation Metrics

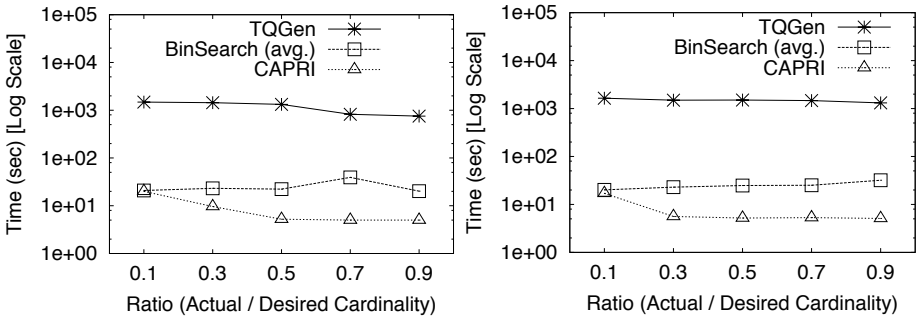
We studied the robustness of CAPRI by varying: (1) the number of predicates (select or join) and the combination of attributes in these predicates, (2) the ratio $\mathcal{C}_{actual}/\mathcal{C}$ between the *actual cardinality* of query Q and the desired cardinality, (3) cardinality of the input table, (4) different data distributions, (5) the presence of join refinement, and (6) the refinement (γ) and cardinality (δ) thresholds. For each setting, we measured the time needed to return the set $\mathbf{Q}_F$ of refined queries, avgerage refinement score for $Q'_i \in \mathbf{Q}_F$, and relative cardinality error = $|Cardinality(Q'_i) - \mathcal{C}|/\mathcal{C}$.

22.1.3 Alternative Techniques

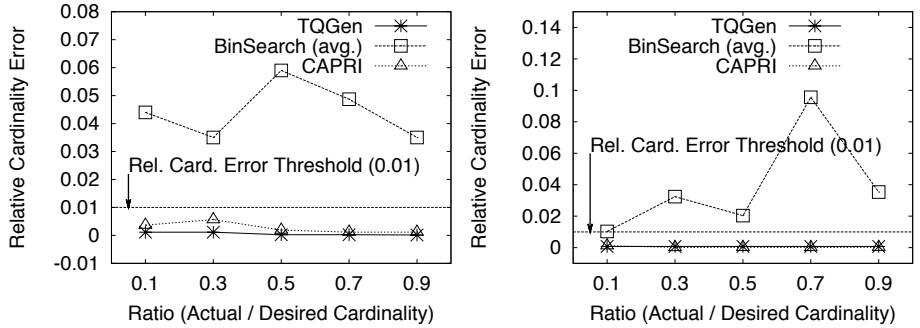
We ran comparative studies of CAPRI with state-of-the-art techniques proposed in (MKZ08), namely, TQGen and binary search (referred to as BinSearch). These techniques do not solve the PCA problem, but are able to produce a single refined query meeting the cardinality constraint by ignoring the proximity criteria. The techniques work as follows. First, both techniques generate the complete set of join results. Following this, TQGen bounds the result space along each select predicate and follows a divide-and-conquer methodology to find a refined query with cardinality close to $\mathcal{C}$. Our experiments used the TQGen parameters reported in (MKZ08). In contrast with TQGen, after join results have been generated, BinSearch performs sequential binary searches on each query predicate to find a query attaining the required cardinality. Since the order of predicate refinement affects the quality of queries returned by BinSearch, we executed the algorithm with all predicate orderings and report the average values for evaluation metrics. It is important to note that neither TQGen nor BinSearch can refine join predicates.

22.1.4 Data Sets

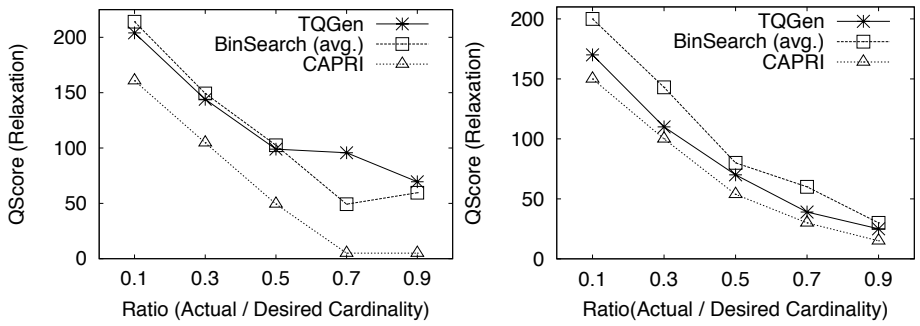
We utilized the TPC-H benchmark data for our experiments and generated datasets of varying sizes and data distributions. Specifically, since the standard TPC-H data is uniformly distributed (i.e., $Z = 0$), we used (CN) to generate TPC-H data with $Z = 1$. In addition, to illustrate the benefits of join refinements we used a synthetic dataset (BKS01) having tables of 10K and join selectivity $\sigma = 0.01$. For our test query workload, we used queries having [2-5] numeric predicates on the TPC-H benchmark dataset.



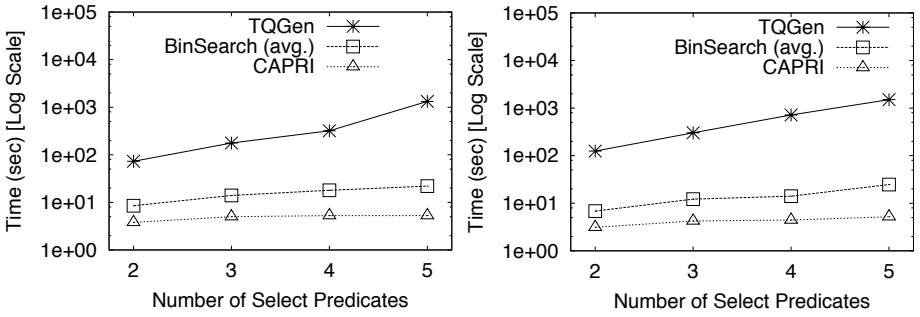
(a) Total Execution Time ($Z = 0$) (e) Total Execution Time ($Z = 1$)



(b) Relative Cardinality Error ($Z = 0$) (f) Relative Cardinality Error ($Z = 1$)



(c) Query Relaxation Score ($Z = 0$) (g) Query Relaxation Score ($Z = 1$)



(d) Effects of Dimensionality d ($Z = 0$) (h) Effects of Dimensionality d ($Z = 1$)

Figure 22.1: Performance comparison: CAPRI against state-of-the-art BinSearch and TQ-Gen; [Experimental Settings: N=100K; $d = 5$ for (a)-(c) and (e)-(g)]

22.2 Performance Comparisons

22.2.1 Refining Select Predicates

We first present a detailed comparison between CAPRI, TQGen and BinSearch for selection queries. For this set of experiments, we used selection queries with [2–5] predicates and different sets of attributes. For each query, we varied the ratio of actual to desired cardinality, $\mathcal{C}_{actual}/\mathcal{C}$, between 0.1 – 0.9. $\mathcal{C}_{actual}/\mathcal{C} = 0.1$ signifies that the desired cardinality is far from the actual cardinality, requiring a large refinement, while a 0.9 ratio means $\mathcal{C}$ is close to $\mathcal{C}_{actual}$, requiring a small refinement. Figures 22.1.a and 22.1.d depict the execution times for the three methods over data sets with $Z = 0$ and $Z = 1$ respectively. We observe that CAPRI is consistently 2 orders of magnitude faster than TQGen and on average 77% faster than *BinSearch*. Unlike CAPRI, both the alternate methods perform repeated cardinality calculations without reusing results, and therefore CAPRI can significantly outperform both techniques. In Figures 22.1.a and 22.1.d, we also notice a decreasing trend in CAPRI execution time. This trend reflects CAPRI's search strategy where more queries are investigated when C_{actual} is not close the desired cardinality C. Figures 22.1.b and 22.1.f compare the relative cardinality errors of the refined queries generated by all three techniques ($\delta = 0.01\%$). We observe that both TQGen and CAPRI are well within the cardinality threshold of 0.01%, while BinSearch is on average 0.05% away from $\mathcal{C}$. Further, as seen in these figures, BinSearch produces uncontrolled errors in cardinality depending on predicate refinement order, giving inconsistent results and leading to lack of robustness. Figures 22.1.c and 22.1.f compare the refinement scores of queries generated by all three techniques. As a consequence of CAPRI's special grid structure and Proximity-driven Exploration, our algorithm is shown to generate queries that on average have 25% better *refinement scores* than queries produced by TQGen or BinSearch. Thus, CAPRI outperforms existing techniques in all three metrics.

Next, we studied the effects of varying the number of predicates on the total execution time for all methods. Figures 22.1.d and 22.1.g report the execution time for $(\mathcal{C}_{actual}/\mathcal{C}) = 0.5$. CAPRI again consistently outperforms TQGen and BinSearch. As the number of predicates increases, the number of queries (independently) executed by TQGen grows exponentially. In contrast, CAPRI extensively shares results during query execution and adopts an early termination strategy. Consequently, CAPRI performs ≈ 2 orders of magnitude faster than TQGen. While BinSearch's search space is not as large as TQGen, it still needs to execute multiple queries independently. Thus, it is on average 55% slower than CAPRI.

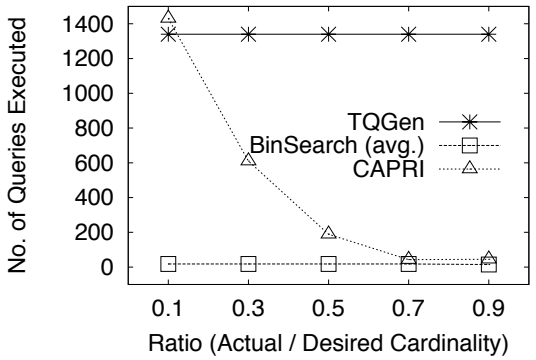


Figure 22.2: Number of Queries Executed; $d = 3$

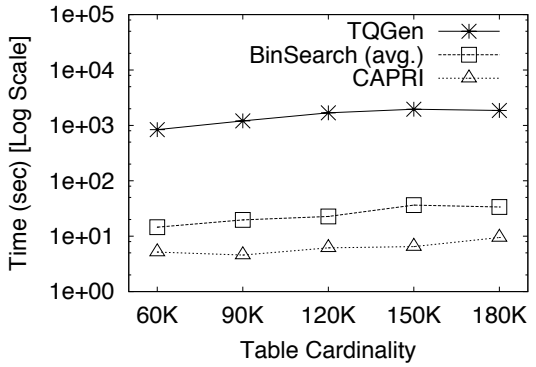


Figure 22.3: Effects of Input Cardinality N

In Figure 22.2, we compare the number of queries executed by the three techniques for varying ratios of $\mathcal{C}_{actual}/C$. The number of candidate queries executed by TQGen

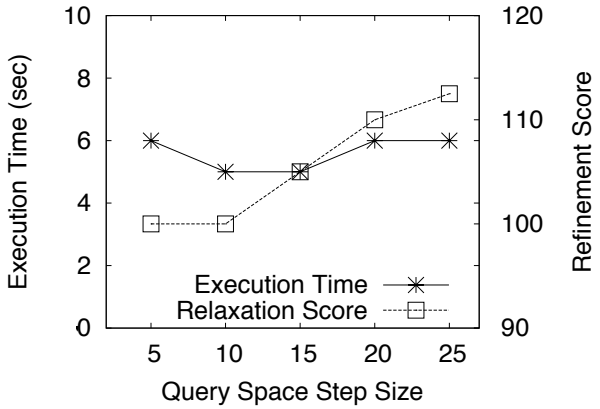


Figure 22.4: Effects of *step-size* $\frac{\gamma}{d}$; $d = 5$; $\mathcal{C}/\mathcal{C}_{actual} = 0.5$

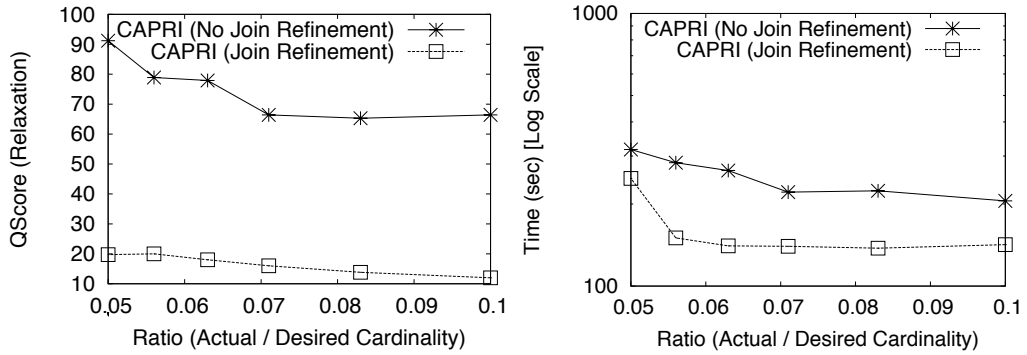
and BinSearch is constant across different ratios of $\mathcal{C}_{actual}/\mathcal{C}$ as they search a fixed number of candidate queries irrespective of how far the desired cardinality is from the actual cardinality. In contrast, the number of queries investigated by CAPRI depends on the ratio $\mathcal{C}_{actual}/\mathcal{C}$. When the desired cardinality $\mathcal{C}$ is close to the actual cardinality $\mathcal{C}_{actual}$, Proximity-driven Exploration enables CAPRI to investigate fewer queries, as shown Figure 22.2. On the other hand, when $\mathcal{C}$ is farther away from $\mathcal{C}_{actual}$, the number of candidate queries investigated by CAPRI increases. However, we note that even when CAPRI investigates a large number of queries, need-based result generation and result sharing enable CAPRI to outperform TQGen and BinSearch.

Finally, in Figure 22.3, we report how the execution times for the three techniques vary with differing input table cardinality. We observe that CAPRI is robust and its execution time is stable for a variety of input sizes.

22.2.2 Refining Both Join and Select Predicates

To highlight the advantage of join refinement, we conducted experiments comparing the results of allowing and disallowing join predicate refinement. Since the TPC-H benchmark only involves equality joins, join refinement is semantically invalid for this dataset, and we instead used a synthetic dataset. In this set of experiments, we joined two tables

with initial join selectivity 0.01, and varied the $\mathcal{C}_{actual}/\mathcal{C}$ ratio in the range of [0.05 – 0.1]. The choice of small $\mathcal{C}_{actual}/\mathcal{C}$ ratios mimicks scenarios where the user query is highly selective, making join refinement attractive. Since TQGen and BinSearch both do not support join refinements, we omit their comparisons. In this experiment, we compared two instances of CAPRI: (1) CAPRI with join refinement enabled, and (2) CAPRI with join refinement disabled. Figures 22.5.a and 22.5.b depict comparisons of the quality of refined queries generated in terms of $QScore$ and the total execution time.



(a) Query Relaxation Score (b) Total Execution Time

Figure 22.5: Refining Join and Select Predicates

Figure 22.5.a indicates that when join predicates are refined, queries recommended by CAPRI have a $QScore$ at least 80% smaller than that of queries generated by select-only refinement. This is because even small changes to join selectivity can produce a large increase in query cardinality. Similarly, in Figure 22.5.b, we observe that the instance of CAPRI with join refinement enabled is 20% faster than CAPRI with join refinement disabled. Join refinement runs faster because CAPRI has to investigate much fewer queries compared to select-only refinement. Thus, we observe that refining join predicates is a powerful tool when the original query cardinality differs vastly from the desired cardinality ($\mathcal{C}_{actual}/\mathcal{C} \leq 0.1$).

22.2.3 Analyzing CAPRI Parameters

Last, we present results of our experiment measuring the effects of step-size $\frac{\gamma}{d}$ on the total execution time and refinement score of queries returned by CAPRI (Figure 22.4). For different $\frac{\gamma}{d}$ ratios and a fixed $\mathcal{C}_{actual}/\mathcal{C} = 0.5$ setting, we observe that the execution time is relatively constant because approximately the same number of result tuples need to be generated. However, a large γ value indicates that the user permits a large departure from the original query. Therefore, we observe that as $\frac{\gamma}{d}$ increases, the refinement score of the queries recommended by CAPRI increases proportionally.

23

Discussion

23.1 Optimizations to CAPRI

To further optimize the performance of CAPRI, we present two important optimization strategies. The first strategy enhances the efficiency of refined space transformation by incrementally building the *result space*. The brute-force approach is to create the *result space* at a higher level abstraction by generating all output regions that map to the result space. However, if we were to only generate output regions that overlap the layer of queries currently under investigation, we eliminate the expenses of generating output regions which subsequently may never be utilized. To implement this optimization, we represent input partitions by their maximum (selection-only) refinement scores and maintain partitions in each table as sorted lists. These lists are then traversed in a round-robin fashion until all output regions likely to lie in the current layer are generated. The next batch of output regions are generated only if CAPRI explores the subsequent layer of queries.

The second optimization enhances proximity-driven refinement by enabling CAPRI to skip query layers and directly jump to the layer containing queries that have a high

probability of satisfying the cardinality criteria. We note that as long as we employ layered exploration, we can still provide the same proximity guarantees as the core CAPRI algorithm. CAPRI can skip query layers by estimating the cardinality of each output region (based on the corresponding input partitions) and using a data distribution model (e.g. normal, uniform, etc.) to calculate the probability of a query in a given layer satisfying the expected cardinality. If for a particular layer this has a probability higher than a tunable threshold, CAPRI skips previous layers and begins exploring the queries in the current layer. Once the cardinalities for all queries in the current layer have been calculated, the recurrences proposed in Chapter 21 can be used to explore subsequent layers as required.

23.2 Handling Non-numeric Predicates

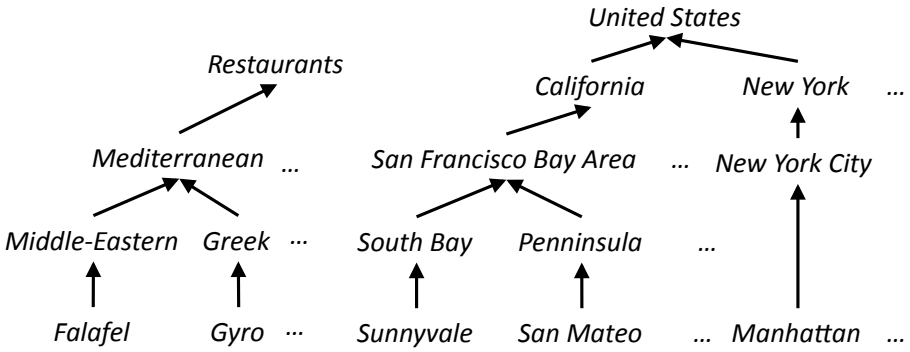


Figure 23.1: Ontology for Categorical Data

The focus of this work is to handle numeric predicates. Measuring refinement distance between categorical data points is in itself a challenging problem, requiring the analysis of taxonomy information. CAPRI can be extended to support categorical predicates by plugging in the appropriate means for measuring the *distance between any two categorical values*. For example, Figure 23.1 depicts sample ontology trees related to *food* preferences and *location*. The refinement distance between the original query de-

siring places that serve *Gyro* to restaurants that have any *Mediterranean cuisine* may be defined based on the relative depths of the two nodes. In general, the roll-up operation on an ontology tree corresponds to making the predicate less selective, i.e., relaxation. While the drill down operation translates to query contraction. To support relaxation of categorical predicates, we augment input partitions with meta-information about the contained categorical values. Given this meta-information and the ontology tree, the CAPRI framework can be used to refine categorical predicates.

23.3 Preferences in Refinement

In Chapter 2.1, to easily expose the core idea, we assumed that the user wants to refine all predicates. However, CAPRI supports scenarios where the user wants to refine only a subset of predicates as well as have upper bounds on maximum possible refinements. The default refinement metric in CAPRI is the L_1 norm, but users can opt to use another L_p norm. In fact, they could provide predicate-specific weights in the L_{W_p} norm. While we provide several avenues for user control, user intervention is necessary and each tunable parameter is provided an appropriate default setting.

23.4 Contracting Queries With Too Many Results

CAPRI can handle scenarios where there are *too many results*. This is achieved by constructing a query Q'_{min} with each predicate of the original query Q set to its minimum value. Since Q'_{min} will produce too few results, we can now construct a refined space bounded by Q and Q'_{min}. CAPRI now traverses the refined space to find queries that meet the cardinality constraint, this time minimizing refinement with respect to Q instead of Q'_{min}.

24

Related Work for Part IV

(Luo06) identified that many real-world queries produce empty results. Techniques that address the empty result set problem can be classified into (1) *tuple-oriented approaches* which only generate the required number of tuples but no query that indicates the criteria that fully describes the retrieved tuple set, and (2) *query-oriented approaches* which generate queries attaining the desired cardinality but ignore the proximity criteria.

Table 24.1: Summary of Related Work for Part IV (CAPRI)

Techniques	Proximity	Cardinality	Query
Tuple-Oriented: skyline (KLTV06), top-k (CG99)	✓	✓	
Query-Oriented: BinSearch (MKZ08), IQR (MK09)			✓
Query-Oriented: TQGen (MKZ08), Hill-Climbing (BCT06)		✓	✓
✠ **Our Approach**	✓	✓	✓

24.1 Tuple-Oriented Techniques

(Mus04, KLTV06) do not focus on the problem of generating refined queries that explain how the result tuples are selected, information that is crucial in many scientific and busi-

ness applications. Similarly, top-k algorithms, such as (CG99), can be used to attain the desired cardinality by producing a set of highly ranked results. While powerful in many instances, this approach does not address the PCA problem. To illustrate, we now attempt to refine query $Q1$ using the top-k approach. To simply, let us focus on refining the select predicated *R.income* and *R.weeklyExercise*. The approach will return the results of the query: "SELECT * FROM Records R, HighRiskLoc L WHERE R.zipcode = L.zipCode AND (R.BMI $>$ 30) AND (18 $\leq$ R.age $\leq$ 30) ORDER BY CASE R.income $<$ 60000 THEN 0 ELSE (R.income - 60000) + CASE R.weeklyExercise $>$ 3 THEN 0 ELSE (3 - R.weeklyExercise) STOP AFTER 5000." The problem with this approach is that the produced tuples are likely to be skewed in certain predicate dimensions. To mitigate this problem, one can attempt to construct the $smallest$ refined query containing all the results obtained using *top-k*.

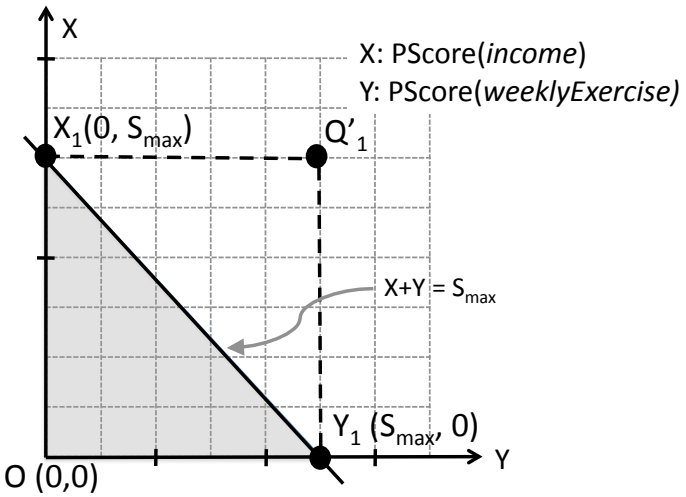


Figure 24.1: Top-k Based Approach

The top-k approach guarantees that the tuples returned all have $PScore(R.income)$ + $PScore(R.weeklyExercise) \leq S_{max}$. Therefore, in the refined space, the 5000 top-k tuples all lie below the line representing $PScore(income)$ + $PScore(weeklyExercise)$ = S_{max} as shown in Figure 24.1. However, the refined query Q'_1 corresponds the rectangle

$OX_1Q'_1Y_1$ instead of the contained triangle OX_1Y_1. If we provide users the 5000 tuples generated by top-k, but provide query Q'_1, we are providing users an extremely biased set of tuples belonging to Q'_1. On the other hand, if we were to provide users the query Q'_1 and "*all*" tuples that satisfy Q'_1 (i.e., those belonging to $OX_1Q'_1Y_1$), we would be giving users many more tuples than required. The discrepancy in cardinality will be even larger for skewed distributions where triangle $Q'_1X_1Y_1$ contains many more tuples than OX_1Y_1. Thus, a pure top-k based approach as well as variations of it prove inadequate to address the PCA problem that requires the auto-generation of refined queries that meet the required cardinality.

24.2 Query-Oriented Approach

In the context of database testing, (BCT06, MKZ08) generate test queries that satisfy cardinality constraints while disregarding proximity criteria. (BCT06) has shown that *generating targeted test queries* is an NP-hard problem. (MK09) proposed a variation of BinSearch, that iteratively narrows the bounds on each selection predicate in a query and asks the user to manually refine the predicate within the constrained dimensions. This approach however cannot be extended to support the refinement of join predicates as CAPRI does. For simple select-only queries, (MK09) seeks only to attain the desired cardinality and disregards proximity. Consequently, it cannot guarantee that the refined query has the least refinement. Moreover, the quality of queries is heavily influenced by the order in which predicates are refined; some orders produce accurate results whereas others produce large errors. Finally, unlike CAPRI, (MK09) does not generate a set of alternative refined queries to the user to choose from. To summarize, CAPRI is the first technique to refine select and join queries to meet the dual constraints of proximity to the original query and desired cardinality.

25

Conclusions of This Dissertation

Decision support systems are technologies that support complex multi-criteria decision making and problem solving. In the early 1970s decision support systems involved developing a robust database management system with effective user interface designs to manage as well as view data in the form of sophisticated report generation tools. In the recent years, decision support systems has evolved into an effective tool to aid users in making informed decisions over "big-data" by complex multi-criteria decision support queries. This growing interest in multi-criteria decision queries has resulted in several classes of queries such as OLAP, Top-K, Nearest Neighbor as well as skyline queries. The intuitive nature of specifying user preferences has made skyline a computation critical component for many multi-criteria decision support applications. The traditional approach of viewing the skyline computation as an "add-on" to SPJ queries can represent performance bottleneck. In this dissertation, we leverage mature DBMS technology by treating the skyline operation as a first-class citizen within a query plan.

In the first part of this dissertation, we focus on the problem of the efficient evaluation of the skyline operationo ver disparate sources. The processing of over skylines over multiple sources is burdened by two primary component cost factors, namely the cost of

generating the intermediate join results and the cost of dominance comparisons to compute the final skyline of join results. State-of-the-art techniques handle this by primarily relying on making local pruning decisions at each source, and are therefore shown to be not robust for a wide variety of data. In addition, existing techniques such as do not consider alternative scenarios when attributes across these sources through user-defined mapping functions to characterize the final result. To address these shortcomings, we propose our **SKIN** approach that supports the robust and efficient evaluation of *SkyMapJoin* queries. We achieve this by taking advantage of optimization opportunities that are available by looking ahead into the mapped output space and exploiting this knowledge at the level of both individual sources and the complete query. We demonstrate the superiority of our approach over existing techniques by consistently outperforming them by several folds, for a wide range of data sets, confirming the robustness of our methodology.

In the second part of the dissertation, we address the need of real-time multi-criteria decision support systems to support the early output of results rather than waiting until the end of query processing. In this work, we propose a progressive query evaluation framework **ProgXe** that is successful in achieving this goal. By exploiting the principle of *SKIN*, *ProgXe* is able to looking ahead into the mapped output space. *ProgXe* by employing 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. Our experimental analysis demonstrates the effectiveness of *ProgXe* over state-of-the-art techniques in progressively outputting the skyline results confirming the robustness of our methodology.

Next in the third part of this dissertation, we address the open problem of contract-driven processing of multiple multi-criteria decision support queries where each is augmented by quality of service (QoS) requirements. Specifically, we propose our **C***ontract-***A***ware* **Q***uery* **E***xecution* (**CAQE**) built on top of *SKIN* to effectively unblocks query

processing by using a multi-granular execution strategy. We develop a uniform model to express a rich set of contracts to represent progressiveness, as well as a metric to continuously measure the degree to which the contracts are satisfied. By analyzing the early-output dependencies amongst the different output regions across different queries, our *contract-driven optimization* methodology is able to maximize the overall satisfaction of the workload. In addition, our *contract-aware executor* is able to exploit sharing of work across multiple queries, continuously monitor the run-time satisfaction of queries and aggressively take corrective steps whenever the contracts are not being met. We demonstrate the superiority of CAQE over existing techniques by showing that in many cases CAQE is $\approx$ 2x more effective in satisfying the queries in the workload.

Lastly, to elucidate the novelty and the generality of the core-principle of abstract-level processing exploited in this dissertation, we successfully apply the principles of *SKIN* to address an orthogonal research problem. More specifically, we introduce the problem of *Proximity-driven Cardinality Assurance* (*PCA*) that seeks to generate refined queries meeting both cardinality and proximity constraints. We establish the NP-hardness of *PCA*, and propose CAPRI – the first framework to address this problem. CAPRI adopts the *Expand and Explore* strategy - that iteratively *expands* the original query to minimize refinement and efficiently *explores* refined queries via a novel incremental execution technique and a predictive index structure. By exploiting *SKIN*'s principle of query processing at different levels of data abstraction, *CAPRI* not only able to efficiently process the refined queries it can also guarantee that each result tuple is processed at most once, regardless of the number of queries it contributes to. Unlike existing techniques, CAPRI provides seamless support for the refinement of select, *as well as* join predicates, crucial in many applications. Moreover, our *Expand-and-Explore* strategy enables CAPRI to perform up to 2 orders of magnitude faster than existing query-oriented techniques, and consistently produce queries with 25% smaller refinements to the original query.

26

Future Work

26.1 Scaling Skyline over Join Queries

Databases systems such as Oracle, DB2, PostgreSQL, etc., must effectively maintain large databases in the order of several gigabytes to petabytes while still efficiently handling large number of concurrent queries. This heavy data and user volume coupled with varying levels of performance requirements make scalability a high priority in enabling a robust database management system. In this section, we highlight future directions in the area of scalability.

26.1.1 Handling Larger Data Sets

Our experimental evaluation reveals that a main-memory based implementation of *SKIN* can safely handle SkyMapJoin queries over data sources of 500K tuples each. We now briefly discuss an adaptation to *SKIN* when the available main-memory is not sufficient.

Pre-processing Phase: The objects associated with the same input partition are clustered together in one or if needed possibly multiple disk pages. The index loading time is similar to that of other indexing techniques such as *R-Trees* and *Bit Map-Index*. For

each source, we only maintain the input-partition abstraction in main-memory, that is, the meta-data describing each of its input partitions such as the identifier, upper and lower bounds and number of tuples.

Region- and Partition- Level Elimination: To determine which region is dominated, the region-level elimination phase solely needs to access the meta-data for each input source – which is compact and thus could be kept efficiently in a main-memory structure. Similarly, the partition-level elimination phase can be performed without any disk access.

Object-Level Execution: For each input partition pair $[I_i^R, I_j^T]$, during the join evaluation, we only retrieve pages associated with the two current input partitions I_i^R and I_j^T and the output partitions that map to the region $\mathcal{R}_{i,j}$. Next we exploit the concept described in Chapter 3.4 to efficiently handle skyline comparisons namely we only retrieve the subset of pages that are associated with the comparable output partitions.

In general, traditional skyline evaluation over large data sets requires disk I/O operations. If the sequence in which the output regions are considered for join and skyline evaluation are poorly chosen, it is quite possible that pages are repeatedly paged in and out of the main-memory, thereby decreasing the overall performance of the algorithm. Disk input/output (I/O) efficient query execution is an important topic with respect to DBMS performance (IK84, SAC$^+$79). To decrease the total number of I/O operations, one can order the sequence in which input-partition pairs are considered for join evaluation. The intuition is to consider two consecutively generated output regions that share the largest number of pages during the skyline comparison phase. To summarize, the SKIN approach proposed in this dissertation can naturally handle scenarios when the raw data resides on disk. We leave this extension as a future work.

26.1.2 Handling High Dimensional Datasets

Scientific applications in fields such as bio-informatics, pharmaceutical drug modeling, on-line text processing, hyper-spectral imaging and astrophysics data processing, work with datasets of high dimensionality in the order of several hundreds of attributes per tuple. In this dissertation, we target multi-dimensional applications (such as those outlined in the introduction) where the number of skyline dimensions is in the order of 2 to 6 dimensions. The reasoning here is that as the number of attributes on which the skyline is applied increases, the cardinality of the output results also increases drastically. Human analyst cannot make effective decisions when faced with such high cardinality result sets. Thus, further processing such as linear ranking of results or clustering, would be needed to be employed (CJT$^+$06b). Increasing usability of the results in an orthogonal problem to the one addressed in this work. In addition to the above concern, high-dimensional skyline evaluation (even for single sets) has exponential time complexity (BKS01). Therefore, it is clearly a challenging task to extend this effort to high-dimensional applications with say hundreds of dimensions. We leave the extensions of this work to handle high-dimensional applications for our future work.

26.1.3 Adaptive Spatial Partitioning

To efficiently handle data sets with low dimensionality, both state-of-the-art and proposed SKIN skyline algorithms make use of popular indexing methodologies such as R-Trees (Gut84), quad-trees or simple grid indices like those in SKIN. Our experiments demonstrate that this is a simple yet extremely effective strategy. Conceptually, these indexing techniques can be extended to handle high dimensional data sets. However, in practice these techniques are usually time and space intensive - making them not viable for large dimensions. It would be interesting in the future to explore alternate partitioning meth-

ods. For instance, we could explore variable-sized partitioning, considering aspects of the population of partitions. This could be achieved by adaptively re-partitioning method triggered by the observed potential benefit for dominance-driven region purging.

Adaptive repartitioning input partitions is guided by the knowledge of the mapped output space thus can alleviate the expensive nature of skyline over join operations. To achieve this we explored in (SRR11) heuristics to determine the threshold value for any given output region, $\mathcal{R}_{i,j}$, which when met will trigger re-partitioning of the input partitions I_i^R and I_j^T . Our re-partitioning policies is guided by three parameters namely, (1) estimated density of the partition, (2) proximity to the origin, and (3) estimated elimination benefits. For example, low density regions which are closer to the origin and have potential to dominate output region(s) with higher density are favored for re-partitioning compared to others that have higher density with lesser elimination benefits.

Alternatively, one could study the distribution-sensitive pre-processing to determine optimal partitioning based on criteria such as data density, relative closeness to the origin, etc. We leave these challenges for the future.

26.1.4 Approximation through Dimension Reduction

One avenue to exploit the capability of traditional dimension reduction techniques such as Principle Component Analysis (PCA) and Fast Map (FL95) to convert high dimensional points is to a set of low dimensional points while maintaining the relative distance between points in the reduced space. It would be interesting in the future to explore their usage as a pre-processing method to convert the input data sets into data sets with lower dimensions and then apply SKIN which works efficiently for lower dimensions. Some of the parameters to be considered before selecting suitable dimension reduction techniques are: first, the cost of converting high dimensional data points into low dimensional data points is low. Second, a data point that is dominated in the reduced space must be

dominated in the original space as well. The opposite should hold true as well. That is, a non-dominated data point in the reduced space must also be non-dominated in the original space. Lastly, in scenarios where the dominance property does not hold, one could able to quantify the percentage to which the skyline on the reduced space accurately represents the skyline on the original space.

26.1.5 Meaningfulness of Skyline Results

The total number of skyline points increases dramatically with the number of skyline dimensions. For a data set with 100K tuples; $d = 10$ and anti-correlated distribution, the skyline operation will result in $\approx$ 75K skyline tuples. In this situation, the skyline operation over high dimensional data sets may no longer offer any interesting insights for decision making. The reasoning behind the growth in skyline results is that as the number of dimensions increases, for a given tuple p, there is a high likelihood that at least one dimension exists in which the attribute value of p is better than that of all other tuples in the data set. In this context, several alternatives have been proposed in the literature such as *k*-dominant skylines (CJT$^+$06a), representative skylines (LYZZ07) and subspaces skyline queries (JTEH07). In the future, one can look into how these concepts translate to the context of skyline and mapping aware query evaluation over disparate sources.

26.1.6 Cardinality Estimation for Skyline-Aware Operators

To make an informed decision when choosing between the various implementation strategies of skyline-aware operators in a DBMS, one needs to study the cardinality estimation of the skyline operation and its interaction with other relational operators. In the context of a *select* operator, adding a select condition will always reduce the overall cardinality of the operator. However, when processing a skyline operator adding a preference

can increase the cardinality, even up to the size of the entire relation. Therefore, skyline cardinality estimation is challenging even under the assumptions such as attribute value independence and attributes having unique values.

In the context of computational geometry, (BKST78) addressed the problem of estimating the average number of maxims in a set of vectors. (BKST78) established that the loose upper-bound cardinality to be $\mathcal{O}((\ln(n))^{d-1})$, where d is the number of skyline dimensions and n is the cardinality of the input data set. While in the average case it is $\mathcal{O}((\ln(n))^{d-1}/(d-1)!)$ (Buc89). The above mentioned approaches address the estimation problem with the strong assumption that the attribute values are independent, unique and completely ordered (CDK06).

The attribute value independence assumption is known to lead to erroneous cardinality estimates even in canonical operators such as joins (HNM$^+$07). This problem is further exacerbated in the context of skyline-aware operators which are very sensitive to the correlation of the different skyline attributes. To illustrate, consider the relation CAR{*age, mileage, price*}. The higher the *age* and the *mileage*, the lower the *price* of the car will be. In this example, the skyline query with the preference P_1 = {LOW(*mileage*), LOW(*age*)} will have a significantly lower cardinality than the query with preference P = {LOW(*mileage*), LOW(*price*)}. This is because the former preference has attributes that show a high degree of correlation while the later preference involves anti-correlated attributes. (CDK06) highlights through experimental analysis that the estimation techniques proposed in (BKST78, Buc89, God04) 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. In the future, we plan to address the novel problem of cardinality estimation of skyline aware operators presented in this dissertation.

26.1.7 Execution Cost-Aware Query Optimization

To treat the *skyline operation* as a first class citizen, the query optimizer must be able to estimate the execution costs of various implementations of the given query and then choose the cheapest alternative. In other words, we must 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. (CDK06) presented cost estimation solutions for the physical implementations of the Block-Nested-Loop (BNL) (BKS01) as well as Sorting-Based approach (CGGL03). An important future area of research is to study various physical implementation of different execution strategy of the skyline-aware operators.

26.2 Multi-Query Multi-Constraint Plan Generation

State-of-the-art algorithms in static databases (IK84, SAC$^+$79, SI93, IK91, KS00) primarily focus on generating an optimal or near-optimal plan by minimizing a single cost function, typically the processing costs is comprised of either I/O or CPU (SMK97). Continuous query processing (MSHR02, CcC$^+$02) differs from its static counterpart in several aspects. First, the incoming streaming data is unbounded and the query lifespan is potentially infinite. Therefore, run-time output rate is a better metric than the total CPU time needed to handle all input data (VNB03). When the per-unit-time CPU usage of a query plan is less than the available system CPU capacity, the query execution is able to keep up with incoming tuples and produce real-time results at an optimal output rate (AN04).

Second, real-time response requirements make continuous queries memory resident (MSHR02). Stateful operators, such as joins, store input tuples in states with which fu-

ture incoming tuples of other streams will join. In time-critical applications, such as fire-sensor monitoring, it is common to have multi-join queries with large numbers of participant streams with high input rates. In such scenarios, the size of the in-memory operator states could potentially grow to be very large, making memory a constrained resource. Memory overflow can result in unacceptable outcomes, such as temporary halt of query execution (VNB03, LZR06, UF00), approximation of query results (TcZ$^+$03) and in some cases thrashing (XZH05). Therefore in this scenario, generating a query plan that is optimal in one resource usage while out-of-bound in the other is not an acceptable solution. Therefore, the aim is to generate a query plan with both resource consumptions within their respective system resource capacities, henceforth called a *qualified plan* (AN04). All qualified plans are guaranteed to produce results at the same output rate (AN04).

To address this qualified plan generation problem, one could attempt to design a combined (singular) cost function that captures both resource usages. This would be beneficial as we could then capitalize on state-of-the-art optimization techniques. However, such an approach suffers from the following drawbacks that make it unsuitable. First, a singular cost function that captures both CPU and memory usages and their correlation *a priori* is in practice hard to obtain (SAL$^+$96). This is because the problem is no longer a minimization problem but rather a system resource constraint satisfaction problem. Also, there is no monotonic function that can clearly characterize the relationship between the resources. On the contrary, we show that these resources in parts of the search space may be positively correlated and in others negatively correlated. Second, a query plan that is minimal by this new singular function need not be optimal or near-optimal in either resource usage nor guaranteed to be qualified. Additionally it is important to note that the plan generation problem is NP-complete (IK84), yet efficient algorithms are a must in the streaming context for run-time optimization. To summarize, *continuous query plan*

generation can be viewed as a **multi-criteria decision making process**.

References

[ACc+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003. 9

[Agg02] Charu C. Aggarwal. Towards meaningful high-dimensional nearest neighbor search by human-computer interaction. In *International Conference on Data Engineering (ICDE)*, pages 593–604, 2002. 2

[AN04] Ahmed Ayad and Jeffrey F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD Conference*, pages 419–430, 2004. 192, 193

[APPK08] Vassilis Athitsos, Michalis Potamias, Panagiotis Papapetrou, and George Kollios. Nearest neighbor retrieval using distance-based hashing. In *International Conference on Data Engineering (ICDE)*, pages 327–336, 2008. 2

[BCP06] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Salsa: computing the skyline without scanning the whole sky. In *International Conference on Information and Knowledge Management (CIKM)*, pages 405–414, 2006. 5, 12, 47, 72

[BCT06] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006. 16, 21, 153, 180, 182

[BGS07] Wolf-Tilo Balke, Ulrich Güntzer, and Wolf Siberski. Restricting skyline sizes using weak pareto dominance. *Inform., Forsch. Entwickl.*, 21(3-4):165–178, 2007. 2

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *International Conference on Data Engineering (ICDE)*, pages 421–430, 2001. 1, 2, 5, 12, 13, 14, 15, 16, 17, 18, 19, 22, 23, 33, 43, 44, 47, 55, 57, 59, 60, 61, 72, 73, 74, 75, 98, 139, 147, 170, 188, 192

[BKST78] Jon Louis Bentley, H. T. Kung, Mario Schkolnick, and Clark D. Thompson. On the average number of maxima in a set of vectors and applications. *J. ACM*, 25(4):536–543, 1978. 83, 191

[BP05] Gloria Bordogna and Giuseppe Psaila. Extending sql with customizable soft selection conditions. In *ACM Symposium on Applied Computing (SAC)*, pages 1107–1111, 2005. 10

[Bre96] David Bremner. Incremental convex hull algorithms are not output sensitive. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 26–35, 1996. 2

[Buc89] Christian Buchta. On the average number of maxima in a set of vectors. *Inf. Process. Lett.*, 33(2):63–65, 1989. 83, 129, 191

[CcC+02] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B.

Zdonik. Monitoring streams - a new class of data management applications. In *Very Large Data Bases (VLDB)*, pages 215–226, 2002. 192

[CD03] Surajit Chaudhuri and Gautam Das. Automated ranking of database query results. In *Conference on Innovative Data Systems Research (CIDR)*, pages 888–899, 2003. 16

[CDD+09] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. Mad skills: New analysis practices for big data. *Proceedings of Very Large Database Conference (PVLDB)*, 2(2):1481–1492, 2009. 10

[CDK06] Surajit Chaudhuri, Nilesh N. Dalvi, and Raghav Kaushik. Robust cardinality and cost estimation for skyline operator. In *International Conference on Data Engineering (ICDE)*, pages 64–73, 2006. 15, 19, 76, 191, 192

[CET05] Chee Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD Conference*, pages 203–214, 2005. 2

[CG99] Surajit Chaudhari and Luis Gravano. Evaluating top-k selection queries. In *Very Large Data Bases (VLDB)*, pages 397–410, 1999. 16, 180, 181

[CGGL03] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting. In *International Conference on Data Engineering (ICDE)*, pages 717–816, 2003. 5, 12, 13, 16, 17, 47, 72, 192

[Cha93] Bernard Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10:377–409, 1993. 2

[CJT+06a] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and

Zhenjie Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD Conference*, pages 503–514, 2006. 190

[CJT+06b] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. On high dimensional skylines. In *International Conference on Extending Database Technology (EDBT)*, pages 478–495, 2006. 188

[CK97] Michael J. Carey and Donald Kossmann. On saying "enough already!" in sql. In *SIGMOD Conference*, pages 219–230, 1997. 16

[CLW03] R. Chen, L. Li, and Z Weng. Zdock: An initial-stage protein docking algorithm. *Proteins*, 52(1), 2003. 7

[CN] S. Chaudhuri and V. Narasayya. Program for tpc-d data generation with skew. 170

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Very Large Data Bases (VLDB)*, pages 354–366, 1994. 105

[DF11] Mark A. Beyer Donald Feinberg. Magic quadrant for data warehouse database management systems. *Gartner RAS Core Research Note G00209623*, January 2011. 1

[DSRS01] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *Symposium on Principles of Database Systems (PODS)*, pages 59–70, 2001. 15, 20

[DTB09] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of Very Large Database Conference (PVLDB)*, 2(1):1246–1257, 2009. 59

[FL95] Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD Conference*, pages 163–174, 1995. 189

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Symposium on Principles of Database Systems (PODS)*, pages 102–113, 2001. 2, 3, 56, 97

[FTAA01] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. Approximate nearest neighbor searching in multimedia databases. In *International Conference on Data Engineering (ICDE)*, pages 503–511, 2001. 2

[Gaa97] Terry Gaasterland. Cooperative answering through controlled query relaxation. *IEEE Expert: Intelligent Systems and Their Applications*, 12(5):48–59, 1997. 15

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *International Conference on Data Engineering (ICDE)*, pages 152–159, 1996. 146

[GÖ05] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD Conference*, pages 658–669, 2005. 20

[God04] Parke Godfrey. Skyline cardinality for relational processing. In *FoIKS*, pages 78–97, 2004. 191

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD Conference*, pages 47–57, 1984. 188

[HcZ07] Jeong-Hyon Hwang, Ugur Çetintemel, and Stanley B. Zdonik. Fast and reliable stream processing over wide area networks. In *ICDE Workshops*, pages 604–613, 2007. 105

[HGHS07] Ryan Huebsch, Minos N. Garofalakis, Joseph M. Hellerstein, and Ion Stoica. Sharing aggregate computation for distributed queries. In *SIGMOD Conference*, pages 485–496, 2007. 20

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997. 20

[HK05] Bernd Hafenrichter and Werner Kießling. Optimization of relational preference queries. In *ADC*, pages 175–184, 2005. 19, 33, 55, 75, 98

[HNM+07] Wook-Shin Han, Jack Ng, Volker Markl, Holger Kache, and Mokhtar Kandil. Progressive optimization in a shared-nothing parallel database. In *SIGMOD Conference*, pages 809–820, 2007. 191

[HRK+09] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan J. Demers. Rule-based multi-query optimization. In *International Conference on Extending Database Technology (EDBT)*, pages 120–131, 2009. 15, 20

[IAE03] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *Very Large Data Bases (VLDB)*, pages 754–765, 2003. 2, 16

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984. 187, 192, 193

[IK91] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD Conference*, pages 168–177, 1991. 192

[JCE+07] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD Conference*, pages 13–24, 2007. 10

[JEHH07] Wen Jin, Martin Ester, Zengjian Hu, and Jiawei Han. The multi-relational skyline operator. In *International Conference on Data Engineering (ICDE)*, pages 1276–1280, 2007. 5, 13, 18, 19, 23, 56, 70, 73, 74, 76, 78, 97

[JMP+10] Wen Jin, Michael D. Morse, Jignesh M. Patel, Martin Ester, and Zengjian Hu. Evaluating skylines in the presence of equijoins. In *International Conference on Data Engineering (ICDE)*, pages 249–260, 2010. 13, 14, 18, 19, 22, 23, 33, 69, 70, 73, 74, 75, 76, 97, 106, 140, 146, 147

[JTEH07] Wen Jin, Anthony K. H. Tung, Martin Ester, and Jiawei Han. On efficient processing of subspace skyline queries on high dimensional data. In *Statistical and Scientific Database Management (SSDBM)*, 2007. 190

[KFHJ04] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. The case for precision sharing. In *Very Large Data Bases (VLDB)*, pages 972–986, 2004. 15, 20

[KGM92] Ben Kao and Hector Garcia-Molina. An overview of real-time database systems. In *NATO Advanced Study Institute on Real-Time Computing*, 1992. 15

[Kie02] Werner Kießling. Foundations of preferences in database systems. In *Very Large Data Bases (VLDB)*, pages 311–322, 2002. 1, 2, 18, 30

[KLP75] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975. 87, 131

[KLTV06] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *Very Large Data Bases (VLDB)*, pages 199–210, 2006. 8, 13, 15, 19, 22, 23, 30, 56, 61, 69, 70, 73, 74, 75, 78, 97, 140, 146, 180

[KML11] Mohamed E. Khalefa, Mohamed F. Mokbel, and Justin J. Levandoski. Prefjoin: An efficient preference-aware join operator. In *International Conference on Data Engineering (ICDE)*, pages 995–1006, 2011. 14, 75, 76, 146, 147

[Kol08] J Koliha. *Metrics, Norms and Integrals: An Introduction to Contemporary Analysis*. World Scientific Publishing Company, 2008. 151

[KRR02] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Very Large Data Bases (VLDB)*, pages 275–286, 2002. 5, 12, 13, 16, 72, 73

[KS00] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems*, 25:43–82, 2000. 47, 192

[KWFH04] Abhijit Kadlag, Amol V. Wanjari, Juliana Freire, and Jayant R. Haritsa. Cardinality estimation using sample views with quality assurance. In *DAFSAA*, pages 594–605, 2004. 16

[LCIS05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD Conference*, pages 131–142, 2005. 2, 3

[LFW+11] Guoliang Li, Ju Fan, Hao Wu, Jiannan Wang, and Jianhua Feng. Dbease: Making databases user friendly and easily accessible. In *Conference on Innovative Data Systems Research (CIDR)*, 2011. 10

[LLN02] Danzhou Liu, Ee-Peng Lim, and Wee Keong Ng. Efficient k nearest neighbor queries on remote spatial databases using range estimation. In *Statistical and Scientific Database Management (SSDBM)*, pages 121–130, 2002. 2

[LQX06] Alexandros Labrinidis, Huiming Qu, and Jie Xu. Quality contracts for real-time enterprises. In *BIRTE*, pages 143–156, 2006. 111, 147

[Luo06] Gang Luo. Efficient detection of empty-result queries. In *Very Large Data Bases (VLDB)*, pages 1015–1025, 2006. 180

[LYZZ07] Xuemin Lin, Yidong Yuan, Qing Zhang, and Ying Zhang. Selecting stars: The k most representative skyline operator. In *International Conference on Data Engineering (ICDE)*, pages 86–95, 2007. 190

[LZR06] Bin Liu, Yali Zhu, and Elke A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD Conference*, pages 347–358, 2006. 193

[MK09] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *International Conference on Extending Database Technology (EDBT)*, pages 862–873, 2009. 8, 16, 180, 182

[MKZ08] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *SIGMOD Conference*, pages 499–510, 2008. 16, 170, 180, 182

[ML05] Ion Muslea and Thomas Lee. Online query relaxation via bayesian causal structures discovery. In *National Conference on Artificial Intelligence (AAAI)*, pages 831–836, 2005. 15

[MMY09] Xiangfu Meng, Z. M. Ma, and Li Yan. Answering approximate queries over autonomous web databases. In *WWW*, pages 1021–1030, 2009. 16

[MPK00] Stefan Manegold, Arjan Pellenkoft, and Martin L. Kersten. A multi-query optimizer for monet. In *BNCOD*, pages 36–50, 2000. 15

[MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002. 192

[Mus04] Ion Muslea. Online query relaxation. In *Knowledge Discovery and Data Mining (KDD)*, pages 246–255, 2004. 15, 180

[MZ93] Hla Min and Si-Qing Zheng. Time-space optimal convex hull algorithms. In *ACM Symposium on Applied Computing (SAC)*, pages 687–693, 1993. 2

[NCS+01] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *Very Large Data Bases (VLDB)*, pages 281–290, 2001. 2, 3

[NW11] Sivaramakrishnan Narayanan and Florian Waas. Dynamic prioritization of database queries. In *International Conference on Data Engineering (ICDE)*, pages 1232–1241, 2011. 15

[PJET05] Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Very Large Data Bases (VLDB)*, pages 253–264, 2005. 146

[PL07] John Pongsajapan and Steven H. Low. Reverse engineering tcp/ip-like networks using delay-sensitive utility functions. In *INFOCOM*, pages 418–426, 2007. 147

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction.* Springer, 1985. 4

[PTFS03] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, pages 467–478, 2003. 5, 8, 12, 13, 14, 72, 105, 106

[PTFS05] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005. 61, 73

[RCL01] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The oracle database resource manager. In *High Performance Transaction Systems*, 2001. 15

[RR09] Venkatesh Raghavan and Elke A. Rundensteiner. Skydb: Skyline aware query evaluation framework. In *SIGMOD Workshops*, 2009. 17

[RR10] Venkatesh Raghavan and Elke A. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In *International Conference on Data Engineering (ICDE)*, pages 733–744, 2010. 17, 140, 146, 147

[RRS11] Venkatesh Raghavan, Elke A. Rundensteiner, and Shweta Srivastava. Skyline and mapping aware join query evaluation. *Inf. Syst.*, 36(6):917–936, 2011. 17, 98, 101

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational

database management system. In Philip A. Bernstein, editor, *SIGMOD Conference*, pages 23–34, 1979. 187, 192

[SAL+96] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, 1996. 193

[SI93] Arun N. Swami and Balakrishna R. Iyer. A polynomial time algorithm for optimizing join queries. In *International Conference on Data Engineering (ICDE)*, pages 345–354, 1993. 192

[SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J. 6:3*, pages 191–208, 1997. 192

[SRR11] Shweta Srivastava, Venkatesh Raghavan, and Elke A. Rundensteiner. Adaptive processing of multi-criteria decision support queries. In *VLDB Workshops (BIRTE)*, 2011. 189

[SWLT08] Dalie Sun, Sai Wu, Jianzhong Li, and Anthony K. H. Tung. Skyline-join in distributed databases. In *ICDE Workshops*, pages 176–181, 2008. 13, 18, 19, 22, 23, 74, 75, 76, 78, 97, 101

[TB10] Vamsidhar Thummala and Shivnath Babu. ituned: a tool for configuring and visualizing database parameters. In *SIGMOD Conference*, pages 1231–1234, 2010. 59

[TcZ+03] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Very Large Data Bases (VLDB)*, pages 309–320, 2003. 193

[TEO01] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In *Very Large Data Bases (VLDB)*, pages 301–310, 2001. 8, 14, 72, 105, 106

[UF00] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000. 193

[VDP11] Akrivi Vlachou, Christos Doulkeridis, and Neoklis Polyzotis. Skyline query processing over joins. In *SIGMOD Conference*, pages 73–84, 2011. 14, 75, 146, 147

[VNB03] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Very Large Data Bases (VLDB)*, pages 285–296, 2003. 192, 193

[WOT10] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD Conference*, pages 651–662, 2010. 14, 15, 20

[XZH05] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *International Conference on Data Engineering (ICDE)*, pages 791–802, 2005. 147, 193

[YL94] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *International Conference on Data Engineering (ICDE)*, pages 89–100, 1994. 105

[YLL+05] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *Very Large Data Bases (VLDB)*, pages 241–252, 2005. 116, 122, 146