CAQE: A Contract Driven Approach to Processing Concurrent Decision Support Queries

Venkatesh Raghavan
Pivotal, Inc.
vraghavan@gopivotal.com

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

ABSTRACT

Real-time analytical systems need to handle workloads comprised of expensive decision support queries with diverse quality of service requirements known as contracts. Contract driven multi-query processing, being an NP-hard problem, remains unaddressed to date. The traditional approach of blindly pipelining the entire input through a shared execution plan is not viable due to the diversity in query contracts. To tackle this challenge, we now develop a flexible model to express contracts and accompany it with an effective means to measure the run-time contract satisfaction. We propose our **C**ontract-**A**ware **Q**uery **E**xecution framework **CAQE**. In this work, we exploit the principle that "*different portions of the input contribute to disparate subsets of queries with varying degrees of progressiveness.*" Therefore, CAQE's processing of the input chunks is driven by how the different query contracts are being met at run-time. To maximize the contract satisfaction of the workload, CAQE leverages the dependencies of input chunks across the queries. This enables us to determine the impact of processing particular input chunks on improving the run-time contract satisfaction. Our experiments demonstrate the effectiveness of CAQE in increasing the overall contract satisfaction of the workload, specifically 2 fold better than existing multi-query processing techniques.

1. INTRODUCTION

Real-time decision support applications must handle workloads of queries with varying quality of service requirements, known as **contracts** [19, 26]. Users of such applications range between those who are willing to pay more for getting a high degree of responsiveness (known as progressive result generation [23, 29]), to cost conscious users that can tolerate some delay. In this work, we tackle the problem of handling workloads of queries with diverse contracts.

1.1 Motivating Real World Applications

Example 1. **Stock ticker applications** *on smart mobile devices enable (1) customers to watch real-time quotes on a list of stocks with a contract requirement of 15 seconds refresh, (2) compile trend analysis on an hourly basis, (3) aggregate the activity on news, blogs, twitter and other social media, about the stocks in customized watch lists, and lastly (4) recommend stocks to diversify the portfolio. Each user may vary in their progressiveness of the system they are willing to pay for.*

Example 2. **Internet aggregators** *access and combine data from several sources to produce complex results. For example, a travel planner enable searching the database of Hotels (H) and Tours (T) to find competing packages [32]. As observed by [26], users of such aggregators often have to wait for a long time for their request to finish, or be presented with stale results. The user expectations of such applications can vary from those that expect a near instantaneous response to those that prefer periodic alerts. Consider the workload defined below.*

- Q_1*: 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_2*: Student Jane Doe is searching for deals in Paris that are cheap and can compromise on distance from points of interest. She wishes to be alerted about attractive packages as soon as they are identified to facilitate immediate action.*
- Q_3*: ACME travel agency designs competitive European tours [32]. Their preference is to maximize ratings and number of sights while minimizing the cost to produce hourly reports about newly available tours.*

The above queries perform joins across the same base tables but differ in which dimensions are of interest. Most importantly, the degrees of progressiveness of the query execution are different.

1.2 Contract-MQP Problem

In this work, we address the problem of **Contract-driven Multi-Query Processing** (**Contract-MQP**). Given a workload of queries augmented with their *contracts* the objective is to develop an execution strategy that maximizes the workload contract satisfaction. For a real-time application to simultaneously meet the contracts of multiple consumers the application must scale. Therefore, exploiting the sharing of execution of computationally intensive queries without compromising on user contracts is crucial. While we focus on multi-criteria decision queries over joins, our proposed principles are general and can be extended to other classes of queries.

1.3 State-of-The-Art Techniques

Multi-query processing (MQO) [28] is typically solved by one of two methodologies. The *time-shared approach* [22] divides the

10.5441/002/edbt.2014.12

total available processing time into slices and allocates them 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. Alternatively, the *shared query plan approach* pipelines each tuple through a shared multi-operator plan [7,18,33]. Each query is viewed as a *subscriber* that consumes the results produced by a *producer* operator within this larger *integrated plan*. For computationally intensive queries the shared plan approach is superior to the time-shared approach due to its effectiveness in reducing the computation load [10,33].

Multi-criteria decision support (MCDS) queries are at the core of advanced analytics that extract value from the underlying datasets by analyzing them across multiple dimensions. A rich variety of such queries have been proposed including *Top-K* queries [8, 13], *convex hull* [20], *nearest neighbor* [1, 9] and *skyline* queries [3]. Many applications require an intuitive means to formulate user preferences over multiple criteria of interest, and return a set of partially ordered results that satisfy it. The ease of expressing multi-dimensional preferences has made skyline queries popular [14,31].

The time-shared MQP approach is not practical for processing 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 join filter with selectivity of 0.1 and skyline dimensions d=2 will generate $\approx$ 1 million join results and require $>$ 1 million pairwise comparisons [5]. Clearly, processing multiple such skyline over join queries individually is prohibitively expensive because it ignores critical optimization opportunities. To the best of our knowledge, we are the first to look into the problem of optimizing workload with multiple skyline over join queries.

1.4 Research Challenges in Contract-MQP

The Contract-MQP problem is NP-hard since its sub-problem, *multi-query optimization* of *select-project-join* queries, is a well established NP-hard problem [28]. 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 by nature. In the worst case scenario, to generate a single skyline result we may have to process all of the join results first [5]. This approach contradicts the high responsiveness (progressive) contracts of some users – who expect partial results to be reported as early as possible rather than waiting until the end of query processing. That is, the skyline operators in the *shared query plan approach* may be blocking the progressiveness of other operators in the plan that serve a different query. To increase progressiveness of a single query Q_i, we may be forced to first fully generate large portions of Q_i's intermediate tuples servicing other queries in the workload. This defeats the chief objective of Contract-MQP.

Although existing *shared query plan* approaches have been effective for the simpler *select-project-join* queries [7, 18], they rely on the queries being monotonic [11]. That is, they produce an append-only result stream assuming that the processing of 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, during the skyline evaluation a newly generated join result can potentially dominate several previously generated join results – making them invalid in the final output.

Moreover, current MQO techniques [7,10,18,33] tend to assume all queries have equal importance. Thereby they ignore the fact that user queries may have diverse and possibly conflicting contracts.

1.5 Our Proposed Approach: CAQE

To address the Contract-MQP problem, we propose our **C***ontract-***A***ware* **Q***uery* **E***xecution* (**CAQE**) framework. *CAQE* takes as input a set of *skyline over join queries* ($S_{\mathcal{Q}}$) and their associated set of contracts ($S_{\mathcal{C}}$). CAQE exploits the core principle that "*different portions of the input contribute to disparate subsets of queries with varying degrees of progressiveness.*" The traditional view of multi-query processing is to blindly pipeline the entire input tuples through a shared plan [10, 18]. This approach however is not viable when the workload queries have varying and sometime conflicting contracts. To overcome this drawback, we model the problem as evaluating small units of work on a shared execution plan. This execution framework allows CAQE's contract-driven optimizer to adaptively pick (at run-time) the next input chunk to be processed. The decision is driven by prioritizing the current subset of queries whose contracts are not yet being met and identifying the input chunks which can meet the run-time contracts of the affected queries. This approach enables CAQE to maximize the overall contract satisfaction of the workload. To the best of our knowledge, we are the first work to address **Contract-MQP** problem. Our contributions are:

- We develop a flexible model to express a rich diversity of progressiveness contracts. This is accompanied with an effective means to measure the run-time contract satisfaction.
- We design a *min-max-cuboid* shared query plan structure that facilitates sharing among queries.
- We propose a *contract-driven optimizer* that employs a cost benefit model to determine the order in which the different input chunks are processed on the min-max-cuboid shared plan – thereby maximizing the workload satisfaction metric.
- We build a *contract-aware execution* strategy that progressively outputs the results of the different queries. In addition, our executor provides continuous feedback to the optimizer about the run-time satisfaction of the queries and trigger corrective steps when contracts are not being met.
- Our experimental analysis over benchmark datasets demonstrates that CAQE consistently outperforms existing techniques. In many cases CAQE is 2 fold better in satisfying query contracts while generating 20 folds fewer join results and conducting 17 folds fewer skyline comparisons than existing techniques.

2. BACKGROUND

In this section, we review the preference model and the query algebra representing a skyline over join (SJ) query.

2.1 Preference Model for Skyline Operation

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 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* $(\forall(a_k \in \mathcal{D})\ (\tau_i[a_k] \leq \tau_j[a_k]) \wedge \exists(a_l \in \mathcal{D})\ (\tau_i[a_l] < \tau_j[a_l]))$.

Example 3. *Let Hotels table (with columns price [p], rating [r], distance [d], WiFi [w]) have following entries: h_1($200, 5,* *0.5, \$20), h_2(\$350, 5, 0.5, \$20), h_3(\$89, 2, 3, \$0). Here, hotel h_1 is cheaper than hotel h_2 for the same rating and distance. Thus h_1* **dominates** *h_2. In contrast, hotel h_1 and hotel h_3 each is better than the other in at least one dimension. Therefore, hotel h_1* **does not dominate** *h_3 ($h_1 \not\prec h_3$) and vice versa ($h_3 \not\prec h_1$).*

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

Example 4. *In Example 3, if the user is only interested in hotels with lower price and free WiFi capability, then hotel h_3 dominates the remaining two hotels. Therefore, in subspace $\mathcal{V} = \{p, w\}$: $h_3 \prec_{\mathcal{V}} h_1$ and $h_3 \prec_{\mathcal{V}} h_2$.*

2.2 Project and Skyline Operations

For a tuple τ, $\mathcal{F} = \{f_1, \ldots, f_k\}$ is a set of k mapping functions, where f_j takes as input a set of distinct attributes B_j to return a value x, i.e., $f_j : Dom(B_j) \rightarrow Dom(x)$.

• **Project** (PROJECT$_{[\mathcal{F},X]}(R)$) operator applies a set of k scalar mapping functions $\mathcal{F}$ to transform each d-dimensional object $\tau_i \in R$ 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 $f_i \in \mathcal{F}$.

Example 5. *If the total price of a ten day trip is the sum of the hotel nightly rate, WiFi charges, and air fare, then the mapping function $f_{total-price}$ is defined as $(price{+}WiFi){*}10{+}air\_fare$.*

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

• **Skyline over Join** (SJ$_{[\mathcal{JC},\mathcal{F},X,P]}(R, S)$) performs the following in order: (1) combines tuples in tables R and S based on the join condition $\mathcal{JC}$, (2) applies the set of scalar mapping functions $\mathcal{F}$ that operate on each join tuple to generate a transformed join tuple (with attributes X), (3) generates the skyline of such tuples based on the preference $P = (E, \succ)$ where $E \subseteq X$.

Q_1: SJ $_{[JC_1, \{f_1, f_2\}, X_1, P_1]}$(R, T); $P_1 = \{d_1, d_2\}$
Q_2: SJ $_{[JC_2, \{f_1, f_2, f_3\}, X_2, P_2]}$(R, T); $P_2 = \{d_1, d_2, d_3\}$
Q_3: SJ $_{[JC_1, \{f_2, f_3\}, X_3, P_3]}$(R, T); $P_3 = \{d_2, d_3\}$
Q_4: SJ $_{[JC_2, \{f_2, f_3, f_4\}, X_4, P_4]}$(R, T); $P_4 = \{d_2, d_3, d_4\}$

Figure 1: Running Query Workload

Example 6. *Figure 1 represents a sample workload S_Q considered in our work. Here, all queries access the same base tables R and T. They however each query differs the join condition specified $\{JC_1, JC_2\}$ and the scalar mapping functions $\{f_1 \ldots f_4\}$ employed on the join results before performing the skyline operation and lastly their respective sets of skyline dimensions $\{P_1 \ldots P_4\}$.*

3. SPECIFYING PROGRESSIVENESS REQUIREMENTS VIA CONTRACTS

In this section, we introduce a versatile model to express user *progressiveness contracts*. Based on this model we design a success metric called *progressiveness score* that measures how the execution strategy is meeting the contracts across the workload at runtime. We utilize this metric to formulate our optimization goal.

3.1 Progressiveness Contract

The progressiveness contract follows the micro-economic principle of the utility of the result tuple [19]. 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 3. **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 finishes.*

Definition 4. *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 0 (least useful) based on its usefulness.*

3.2 Contract Specification Models

We now present a sample of the alternative models supported in CAQE to specify contracts.

3.2.1 *Time Based Contract*

Commercial systems such as IBM DB2, Microsoft SQL Server, and Oracle, enable users to specify contracts based on *response time*. The time indicates the deadline by which all results need to be reported. Result tuples produced after time t_{hard} are useless to the application, that is, have utility zero.

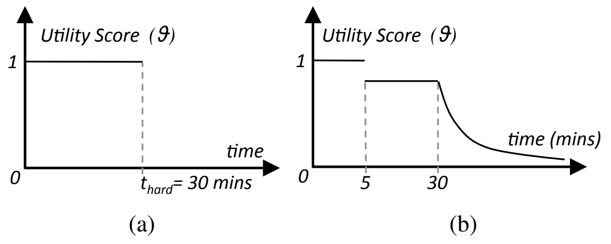


Figure 2: Time Based Utility Function

Example 7. *Figure 2.a depicts a time constraint where all tuples generated after 30 minutes have no use. We propose to model such time-based contract by utility functions:*

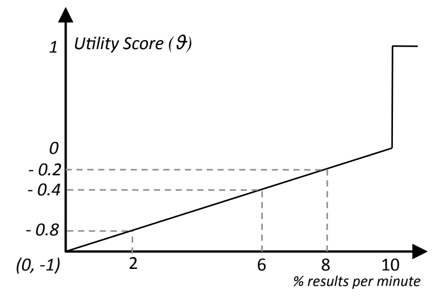
$$\vartheta_{time}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \leq 30 \\ 0 & \text{for} \quad \tau_k.ts > 30 \end{cases} \tag{1}$$

Example 8. *Alternatively, one can specify a decay function that decreases the result utility as query execution progresses. For example, the utility function that models decay in Figure 2.b. is:*

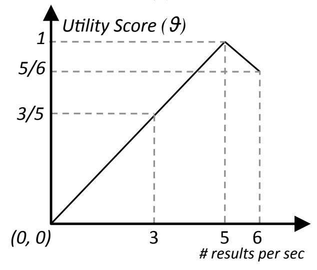
$$\vartheta_{time}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \leq 5 \\ 0.8 & \text{for} \quad 5 < \tau_k.ts \leq 30 \\ log(1/\tau_k.ts) & \text{for} \quad \tau_k.ts > 30 \end{cases} \tag{2}$$

3.2.2 *Cardinality Based Contracts*

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



(a)



(b)

Figure 3: Cardinality Based Utility Function

Example 9. *The requirement that 10% of total results be returned every minute is represented in Figure 3.a. Here the x-axis represents the % of the total results to be returned every minute. In Figure 3.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:*

$$\vartheta_{card}(\tau_k) = \begin{cases} 1 & \text{for} \quad n_{i,j}/N_{est} \geq 0.1 \\ n_{i,j}/(N_{est} * 0.1) - 1 & \text{for} \quad n_{i,j}/N_{est} < 0.1 \end{cases} \tag{3}$$

where N_{est} is the estimated final 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 10. *An example preference about the query output rate occurs when the user can handle at most 5 tuples/sec. The corresponding contract is depicted in Figure 3.b. Here, the x-axis represents the number of tuples generated every second and y-axis the utility of each tuple. The utility function of such a contract is:*

$$\vartheta_{card}(\tau_k) = \begin{cases} (n_{i,j}/5) & \text{for} \quad n_{i,j} \leq 5 \\ (5/n_{i,j}) & \text{for} \quad n_{i,j} > 5 \end{cases} \tag{4}$$

3.3 Hybrid Contracts

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

Example 11. *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 3 and 1 respectively).*

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

$$\vartheta(\tau_k) = \vartheta_{card}(\tau_k) * \vartheta_{time}(\tau_k) \tag{5}$$

3.4 The CAQE Optimization Goal

Definition 5. *Given a set of skyline over join queries $\mathcal{S}_Q$ 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,*

$$Maximize : \sum_{i=1}^{|S_Q|} pScore(Q_i, \mathcal{C}_i, \mathcal{E}_{shared}) \tag{6}$$

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

$$pScore(Q_i, \mathcal{C}_i, \mathcal{E}_{shared}) = \sum_{k=1}^{|Result(Q_i, \mathcal{E}_{shared}, t_{start}, t_{end})|} \vartheta_i(\tau_k) \tag{7}$$

where $\tau_k \in Result(Q_i, \mathcal{E}_{shared}, t_{start}, t_{end})$ (see Definition 3).

4. CAQE: AN OVERVIEW

We now present a brief overview of our **C***ontract-***A***ware* **Q***uery* **E***xecution* (CAQE) framework. The core principle exploited in this work is: "*different portions of the input contribute to disparate subsets of queries with varying degrees of progressiveness.*" By processing at different levels of data abstractions we expose and then exploit opportunities for fine-grained sharing among complex skyline over join queries. Given this overall approach, we address the following open questions:

1. Given a workload of skyline over join queries, how to effectively partition the total work into smaller units of work (chunks) that maximizes execution sharing without sacrificing progressiveness?
2. How does the processing of a given input chunk affect the run-time satisfaction of an individual workload query?
3. Results produced by processing one chunk can determine which subsets of the results produced by other chunks can contribute to the final output of workload queries. How do we exploit such output dependencies among chunks?
4. Lastly, how can the overall contract satisfaction across queries be maximized?

The CAQE framework as depicted in Figure 4 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 a given workload S_Q that maximizes the sharing of expensive operations (join and skyline operations) across queries (Section 4.1).

Multi-Query Output Look Ahead evaluates the workload at a coarse granularity over this shared min-max cuboid plan to build an

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

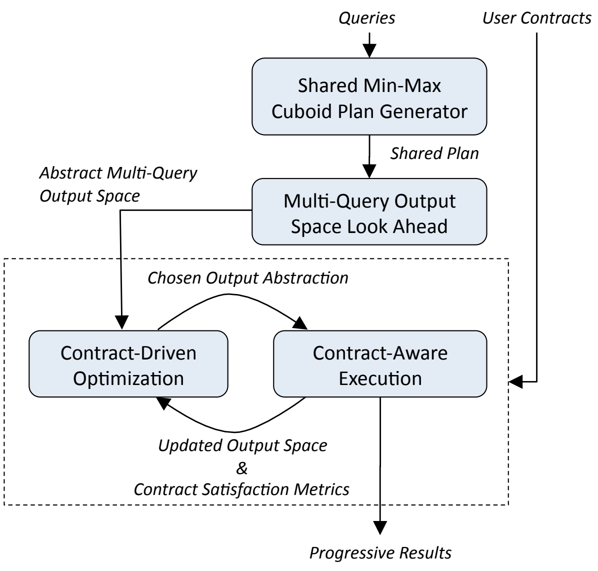

Queries
User Contracts
Shared Min-Max Cuboid Plan Generator
Shared Plan
Abstract Multi-Query Output Space
Multi-Query Output Space Look Ahead
Chosen Output Abstraction
Contract-Driven Optimization
Contract-Aware Execution
Updated Output Space & Contract Satisfaction Metrics
Progressive Results

Figure 4: Overview of the CAQE Framework

abstract multi-query output space. For each input data source, we form an d-dimensional abstraction where *d* is the total number of skyline dimensions used in the workload. In Section 5 we elaborate the methodology by which we perform coarse level query processing. Rather than directly diving into tuple-level processing, we look ahead into this multi-query output space to quickly identify groups of input tuples that contribute to multiple queries. This approach facilitates the sharing of common sub-expressions across queries.

Contract-Driven Optimization analyzes the abstract-level space to enable CAQE to determine the dependencies among regions in the output space across multiple queries that can be exploited to increase progressiveness for queries as designated by contracts. *Contract-driven optimization* employs a novel contract-based benefit model to determine the order in which the output regions are considered for tuple-level processing (Section 5.3).

Contract-Aware Execution iteratively processes the region selected by the *contract-driven optimizer* over the shared min-max cuboid plan. The executor exploits the dependency knowledge captured by our abstract multi-query output space to identify which subset of the join results generated thus far can be output to any of the workload queries. We continuously monitor the run-time satisfaction of the queries in meeting their respective contracts and adatively take corrective steps when necessary to maximally satisfy the contracts (Section 6).

4.1 Shared Min-Max Cuboid Plan

Next we generate a shared query plan that compactly represents the workload $\mathcal{S}_Q$. The objectives of shared plan are to (1) reduce unnecessary operations such as scans and skyline comparisons, and (2) minimize the total number of intermediate results. Given that skyline operation represent the most blocking query operation in this work, we henceforth describe our solution for workloads containing queries that differ in their skyline dimensions while the remaining query properties are identical. Generating shared plans for selects, joins and group by operations have already been discussed in literature [10, 18] and can be applied as is.

Consider the workload represented in Figure 1 (Section 2). If *Distinct Value Attributes* (DVA) *-property*[2] 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, we can still compute the skyline results over subspace $\{d_1, d_2, d_3\}$ from the results in subspaces $\{d_1, d_2\}$ and $\{d_2, d_3\}$. For a tuple τ_i to be in the skyline $SKY_{(d_1,d_2,d_3)}$, we need to only compare τ_i to those tuples in subspaces $\{d_1, d_2\}$ and $\{d_2, d_3\}$ with the same d_2 and/or d_3 attribute values. This allows us to perform dominance comparisons along dimensions d_1 and d_3 only once rather than separately for queries Q_2, Q_3 and Q_4.

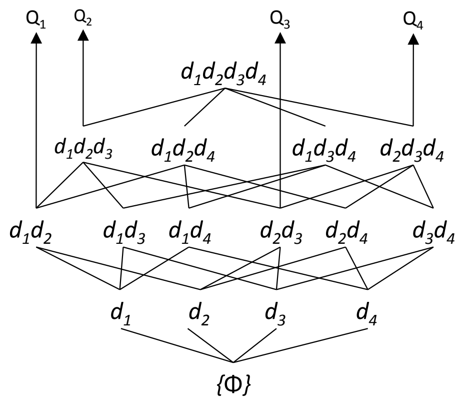


Figure 5: Full Skycube

For some workload of queries maintaining the entire $2^d - 1$ possible subspaces (known as *skycube* [36]) as in Figure 5 is unnecessary. We therefore prune the space to only contain subspaces that contribute to at least one query.

Definition 6. *For query $Q_i = SJ_{[\mathcal{JC}_i, \mathcal{F}_i, X_i, P_i]}(R, S)$, a subspace $\mathcal{U}$* **serves** *Q_i iff $\mathcal{U}$ is a subset of the skyline dimension specified in the Q_i's preferred dimensions P_i. The set of all queries in $\mathcal{S}_Q$ that $\mathcal{U}$ contributes to is denoted as $Q_{Serve}(\mathcal{U}, \mathcal{S}_Q)$.*

Example 12. *In Figure 5, the results in subspace $\{d_2, d_3\}$ contribute to queries Q_2, Q_3 and Q_4, whereas the 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 child subspaces $\{d_2, d_3\}$ and $\{d_2, d_4\}$ does help other queries.

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 only maintain the root subspace $\mathcal{V}$. We therefore propose **min-max-cuboid** as our structure to represent the shared plan. The min-max-cuboid plan is guaranteed to contain the minimal subset of subspaces while maximizing sharing (see Definition 7).

Definition 7. *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:*

1. $(|\mathcal{U}| = 1) \vee (|Q_{Serve}(\mathcal{U}, \mathcal{S}_Q)| > 1)$
2. $\nexists \mathcal{V}$ s.t. $[(\mathcal{U} \subset \mathcal{V}) \wedge (Q_{Serve}(\mathcal{U}, \mathcal{S}_Q) \subseteq Q_{Serve}(\mathcal{V}, \mathcal{S}_Q))]$

[2]*DVA-property states that no two tuples share the same value for any given skyline dimension [36].*

3. $\mathcal{U}$ is the complete set of skyline dimensions of one $Q_i \in \mathcal{S}_Q$

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

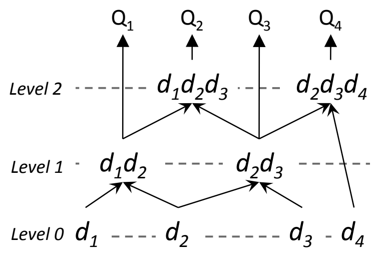


Figure 6: Min-Max Cuboid

Example 13. *In Figure 6, all subspaces in level 0 meet condition 1 of Definition 7 since $|\mathcal{U}| = 1$. Subspaces in level 1 service queries Q_1 and Q_3. Lastly, subspaces in level 2 are the skyline dimensions for queries Q_2 and Q_4.*

5. MULTI-QUERY OUTPUT LOOK AHEAD

The objective of the **M***ulti*-**Q***uery output* **L***ook* **A***head* (MQLA) step is to perform the query evaluation over the shared min-max cuboid plan at a coarser granularity of values rather than at the level of individual tuples. MQLA benefits CAQE's query processing capability in the following ways:

1. **Modular Execution.** Establish mapping between the coarse abstractions of the input space containing input tuples and *output regions* containing the results of different queries that are generated during execution. This enables CAQE to chop the total work into smaller chunks.
2. **Contract Driven Processing of Input Chunks**. Identify where the skyline results for the different queries lie in the abstract multi-query output space. This enables CAQE to prioritize the processing of the input chunks based on the run-time satisfaction of the different queries.
3. **Advanced Execution Sharing.** The mapping between input and output spaces facilitates CAQE to quickly identify groups of input tuples that contribute to multiple queries and thereby facilitating execution sharing.
4. **Avoid Redundant Work.** By aggressively pruning output regions that are guaranteed to not generate even a single skyline result for any workload query.

During the actual tuple-level query execution (Section 6) we populate this multi-query output space with the actual skyline results as they are being generated.

5.1 Coarse-Level Join Operation

We now perform the join execution for all queries at a coarser granularity of values rather than at the level of individual tuples. To facilitate this coarse grained processing, we assume the input data sets are *partitioned* into a d-dimensional quad tree. Now for a pair of input cells, one from each table $L_a^R \in R$ and $L_b^T \in T$, we first determine if tuples in these cells will produce even a single join result for any of the queries. To facilitate this coarse-grained join evaluation, each cell maintains a signature for each join predicate that captures the domain values of its member tuples.

Example 14. *Consider a supply chain application over the tables $RETAILER$ and $TRANSPORTERS$. Join predicate for Q_1 is r_country = t_country while that of Q_2 is r_part = t_part. To illustrate coarse-level join processing, consider two leaf cells one from each table. L_i^R includes suppliers from {Brazil, China, Mexico} that supply {Tires, Iron Ore, Brass Sheets}, and (2) L_j^T contains transporters from {Brazil, China, Germany, Mexico} that specialize in transporting {Dairy Products, Medical Supplies}. In otherwords, L_i^R[country] = {Brazil, China, Mexico}, L_i^R[part] ={Tires, Iron Ore, Brass Sheets}, while L_i^T[country] = {Brazil, China, Germany, Mexico}, L_i^R[part] ={Dairy Products, Medical Supplies}*

Notation	Meaning
$L_i^T(l_i, u_i)$	An **leaf cell** in table T defined by its $d-$dimensional lower and upper bounds
$\mathcal{L}^T$	Set of all **leaf cells** for the table T
Sig_i	**Signature** of a given cell for the join predicate JC_i
R_i	A d-dimensional **region** in the **output space** that contains results of one or more queries in the workload
$REG(Q_j)$	Set of non-dominated regions that contribute to Q_j
$RQL(R_i)$	Set of queries that output region R_i contributes to

Table 1: Notations Used In Section 5

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.

Example 15. *Consider the Example 14 of a supply chain application over $RETAILER$ and $TRANSPORTERS$ tables. Join predicate for Q_1 is s_country = t_country while that of Q_2 is s_part = t_part. From the signatures of the cells we can determine that the output regions resulting from $L_i^R \bowtie L_j^T$ will satisfy query Q_1 since L_i^R[country] $\cap L_j^T$[country] = {Brazil, China} $\neq \phi$. In contrast for Q_2, L_i^R[part] $\cap L_j^T$[part] $= \phi$ and thus will not contribute to Q_2. Therefore, the output region need $\underline{\text{only}}$ to be considered for skyline-level processing $\underline{\text{for}}$ Q_1 (and $\underline{\text{not}}$ $\overline{Q_2}$).*

5.2 Coarse-Level Skyline Operation

Next, we perform the abstract-level skyline operations rather than conducting expensive pairwise tuple comparison for each query in the workload. This effectively determines which of output regions generated in the previous step are guaranteed to not contribute to a single workload query. Therefore such output regions can be safely eliminated from further processing.

In other words, for each query Q_i we identify output regions that can potentially contribute to the skyline results depending on the actual result distribution determined during tuple-level processing. 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** $RQL(R_i)$. The comparison between attribute values of two different output regions R_i and R_j is meaningful only if they serve the same subset of queries i.e., $RQL(R_i) \cap RQL(R_j) \neq \phi$.

Definition 8 **(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 them 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 1. *Given subspaces $\mathcal{V}$ and $\mathcal{U}$ s.t. $\mathcal{U} \subset \mathcal{V}$, if R_i is a non-dominated region in the subspace $\mathcal{U}$, then R_i is guaranteed to not be dominated in subspace $\mathcal{V}$*[3].

Proof: Proof by contradiction. For subspace $\mathcal{U} \subset \mathcal{V}$ and a pair of output regions $\{R_j, R_i\}$, let $(R_i \not\prec_{\mathcal{U}} R_j) \wedge (R_i \prec_{\mathcal{V}} R_j)$ hold. Given that the DVA property [36] 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 $(R_i \prec_{\mathcal{U}} R_j)$. 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 1. *Given subspaces $\mathcal{U}_1$, $\mathcal{U}_2$ s.t. $\mathcal{U}_1 \subset \mathcal{V}$ 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 $\{R_i, R_j\} \in SKY_{\mathcal{V}}$.*

In CAQE we perform abstract-level dominance comparisons in a *bottom-up fashion* starting at subspaces in Level 0 of the *Min-Max Cuboid*. By utilizing Theorem 1 and its Corollary 1, we first populate the *Min-Max Cuboid* ($\mathbb{M}$) to determine the list of queries a region R_i contributes to.

Example 16. *Consider three output 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 Figure 6, R_1 belongs to $SKY_{(d_1)}$, and R_3 belongs to $SKY_{(d_2)}$, $SKY_{(d_3)}$, and $SKY_{(d_4)}$. For level 1 by Theorem 1, we deduce that $SKY_{(d_1,d_2)} = \{R_1, R_3\}$ and $SKY_{(d_3,d_4)} = \{R_3\}$. Next, we check 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 $SKY_{(d_2,d_3)} = \{R_2, R_3\}$.*

5.3 Contract-Driven Optimization

A naive technique for query execution is to blindly pipeline the tuples mapped to input cells associated with each output region over the shared execution plan. In this work, we estimate the impact of tuple-level processing of each output region on the contract satisfaction metric of each query, to maximize the cumulative satisfaction of the workload (see Definition 5).

In real-time applications, it is not practical to find the optimal ordering by which the output regions are sent for tuple-level processing since the cardinality estimation is very error prone, especially with respect skyline queries [14]. In this work we instead take the approach of iteratively picking the next region best estimated to improve the overall satisfaction of the workload. This feedback-driven iterative approach in-turn identifies the impact of each region selection decision on the overall contract satisfaction metric, and thus to take immediate corrective actions whenever a poor choice is being made.

5.3.1 Contract Satisfaction Metric

We identify "*the current best*" candidate among all regions for tuple-level processing as the region with the highest contract satisfaction metric at the given time instance t_{curr}. Let t_c be time required by region R_c to complete its tuple-level processing and is estimated to progressively output $N^i_{est}(t_c)$ 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:

$$CSM(R_c, \mathcal{E}_{shared}, \mathcal{C}, t_c) = \sum_{Q_i \in S_Q} w_i * \sum_{j=1}^{N^i_{est}(t_c)} \vartheta_i(\tau_j) \quad (8)$$

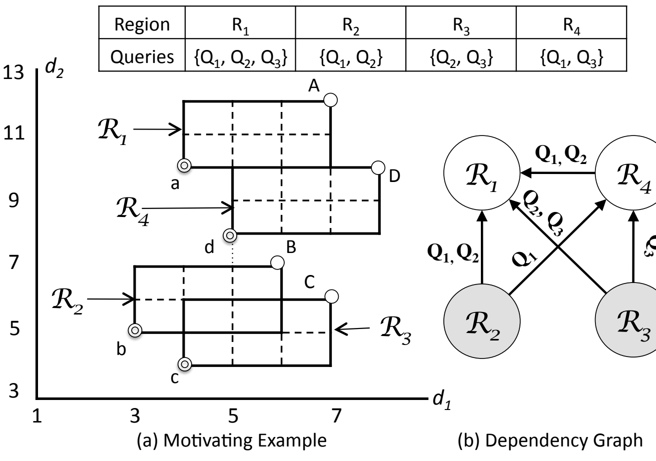
[3]Under the DVA assumption.

where $N^i_{est}(t_c)$ is the progressiveness estimate for query Q_i at time $t_{curr} + t_c$ (see Definition 10 and Equation 10) 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.

5.3.2 Progressiveness Based Benefit Model

Next, we describe our cardinality estimation model to compute the progressiveness benefit of a region. We introduce the concept of **dependency graph** to capture the output dependencies among the different pairs of output regions.



Region	R_1	R_2	R_3	R_4
Queries	$\{Q_1, Q_2, Q_3\}$	$\{Q_1, Q_2\}$	$\{Q_2, Q_3\}$	$\{Q_1, Q_3\}$

Figure 7: Dependency Graph

Definition 9. *A directed* **dependency graph** *$DG(V, E)$, where (1) V is set of vertices (regions); (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 partially dominates one or more output cells in R_j.*

Example 17. *In Figure 7, 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 $\mathbb{W}_{2,1} = \{Q_1, Q_2\}$.*

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

Definition 10. *The* **progressiveness estimate** *of region R_c for query Q_i at time t_c is the fraction of all the results produced by R_c that are guaranteed to be in the final skyline at time $t_{curr} + t_c$ and is denoted as $ProgEst(R_c, Q_i, t_c)$.*

Let L^R_a and L^T_b 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 [4] as:

$$Cardinality(R_c, Q_i) = ln(\sigma_i \cdot n^R_a \cdot n^T_b)^{d-1}/(d-1)! \quad (9)$$

where $n^R_a = |L^R_a|$ and $n^T_b = |L^T_b|$.

Definition 11. *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 18. *In Figure 7.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 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 11 and Equation 9 the progressiveness estimate of R_c for query Q_i can defined as follows:

$$\begin{aligned} ProgEst(R_c, Q_i, t_c) = &\Big(\frac{ProgCount(R_c, Q_i, t_c)}{CellCount(R_c, Q_i)}\Big) \\ &* Cardinality(R_c, Q_i) \end{aligned} \quad (10)$$

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

5.4 Putting It All Together

Algorithm 1: *Contract-Driven Optimization*

```
Input  : $\mathfrak{R}$ (Region Collection); input partitions ($\mathcal{L}^R$, $\mathcal{L}^T$);
         query workload $\mathcal{S}_Q$ contracts $\mathcal{S}_C$
Output: Iteratively pick the next region $R_{next}$ to process.
Build the initial dependency graph, $DG$;
for each $R_c$ in $DG_{root}$: do
    computeCSM($R_c, \mathcal{S}_Q, \mathcal{S}_C$);
    Add $R_c$ to 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 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 $R_f$'s CSM scores.
        $DG_{root'} \leftarrow$ new root nodes due to removal of $e_{c,f}$;
    for each $R_g \in DG_{root'}$ do
        compute CSM($R_g, \mathcal{S}_Q, \mathcal{S}_C$)(Definition 10);
        Add $R_g$ to $PQueue$;
    $DG_{root} \leftarrow DG_{root} \cup DG_{root'}$;
    Remove $R_c$ from $\mathfrak{R}$;
return;
```

The pseudo-code of the *contract-driven optimization* is listed in Algorithm 1. The *progressive benefit model* (Equation 10) estimates the number of tuples that a region is likely to output after its evaluation. Our cost model 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 8) 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 contract satisfaction metric of each query Q_i are being met during tuple-level processing, we update the CSM-based benefit model by adjusting its weight (see Section 6). This process is repeated until all regions have either been considered for tuple-level processing or have been dominated by newly generated tuple(s).

6. CONTRACT-AWARE EXECUTION

Given a particular region selected for processing by the optimizer, CAQE's *contract-aware execution* engine process the tuples in the input cells associated with the chosen region 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, project and skyline) over the shared min-max cuboid plan.
2. **Progressive Result Reporting.** For each query $Q_i \in \mathcal{S}_Q$ determine the subset of the generated result tuples that are guaranteed to be in the final skyline.
3. **Run-time Satisfaction Metric and Optimizer Feedback.** Based on contracts and the skyline results generated so far, we update the run-time satisfaction metric of each query $Q_i \in \mathcal{S}_Q$. Use this run-time metric to update the benefit model used by the contract-driven optimizer to pick the next region.

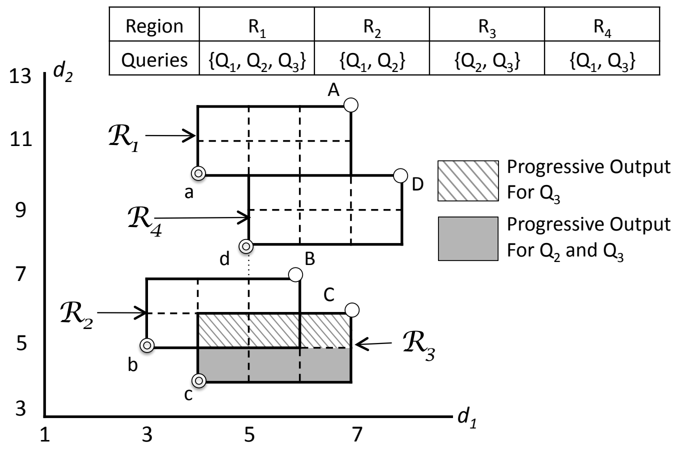
Tuple Level Processing. For the chosen region $\mathcal{R}_c$, we first evaluate the join conditions between the tuples in the input cell L_a^R and those in L_b^T. Join results are then mapped to their output cells by applying the mapping functions. For each output cell O_x we maintain the **cell query-lineage** (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 DG (see Section 5.3.2) 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 $RQL(R_c) \cap RQL(R_f)$.

Progressive Result Reporting. To support progressive result reporting, we push the decision making from the individual tuple-level to the coarser granularity of output cells. More precisely, we translate the problem of determining which result can be progressively output into a problem of determining output cells with the following properties: (1) no future results are guaranteed to map to the output cell, and (2) the tuples in output cell are guaranteed to not be dominated by future tuples that map to any other output cell.

Example 19. *In Figure 8, 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*



Region	R_1	R_2	R_3	R_4
Queries	{Q_1, Q_2, Q_3}	{Q_1, Q_2}	{Q_2, Q_3}	{Q_1, Q_3}

Figure 8: Multi-Query Progressive Output

$O[(4,4)(5,5)]$, $O[(5,4)(6,5)]$ 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.

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 $\mathcal{C}_i$. For query Q_i and its associated contract C_i, the run-time contract satisfaction metric at a given time instance t_j, is the average utility score of all the results being reported at time t_j (denoted as $v(Q_i, t_j)$). Based on this metric we adjust Q_i's weight w_i in Equation 8 for the next iteration of query 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:

$$w_i' = w_i + \frac{v_{curr-max} - v(Q_i)}{\sum_j^N (v_{curr-max} - v(Q_j))} \tag{11}$$

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

Example 20. *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 11 the new weights are $\{1.43, 1, 1.13, 1.43\}$*[4]*. In other words, we bump up the priorities of Q_1, Q_2 and Q_3 since they have not yet meet their respective contracts.*

7. EXPERIMENTAL STUDY

7.1 Experimental Settings

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.

Contract Models. As described in Section 3.2, progressiveness contracts in CAQE follow the micro-economic principle to determine the utility of a result tuple [19]. We tested CAQE's effectiveness under different classes of contracts, namely time-based (C1, C2 and C3), cardinality-based (C4) and hybrid (C5) contracts. Table 2 summarizes the contract models used in this study where t_{C1}

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

	Utility Functions
C1	$\vartheta_{C1}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \le t_{C1} \\ 0 & \text{for} \quad \tau_k.ts > t_{C1} \end{cases}$
C2	$\vartheta_{C2}(\tau_k) = 1/log(\tau_k.ts)$
C3	$\vartheta_{C3}(\tau_k) = \begin{cases} 1 & \text{for} \quad \tau_k.ts \le t_{C3} \\ 1/(\tau_k.ts - t_{C3}) & \text{for} \quad \tau_k.ts > t_{C3} \end{cases}$
C4	$\vartheta_{C4}(\tau_k) = \begin{cases} 1 & \text{for} \quad n_{i,j}/N \ge 0.1 \\ n_{i,j}/(N * 0.1) - 1 & \text{for} \quad n_{i,j}/N < 0.1 \end{cases}$
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 2: Progressive Contracts Used in the Experimental Study

and t_{C3} are tunable parameters for contracts $\{C1\}$ and $\{C3\}$ respectively, while $n_{i,j}$ is the time interval used in contracts $C4$ and $C5$. In Table 2 N is the total of output tuples for query Q and $\tau_k.ts$ is the output time of the result tuple τ_k.

Data Sets. We conducted our experiments using the *de-facto* standard datasets used to stress test skyline algorithms [3]. 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 portion 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 number 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 $|R| = |T| = N$.

Query Workload. We focus on queries similar to the motivating examples in Section 1. That is, queries that perform join, project and 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.

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* [17], *Skyline-Sort-Merge-Join* (*SSMJ*) [14] and ProgXe+ [27]. In all systems, while queries are processed in the order of the priority pr_i, 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 Section 4.1).

Evaluation Metrics. In our analysis 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 utility of each result tuple for each workload query, (2) the total execution time to return the complete result set, (3) memory usage (number of join results), and (4) CPU usage (number of skyline comparisons needed). Lastly, we calculate the average satisfaction metric of each workload query.

7.2 Contract Satisfaction Metric

We now analyze the performance of the algorithms under varying contract and data distribution models. In this set of experiments (Figure 9 and Figure 10), we vary the priority of the queries such that for contract models $\{C1, C2\}$ queries with a larger number

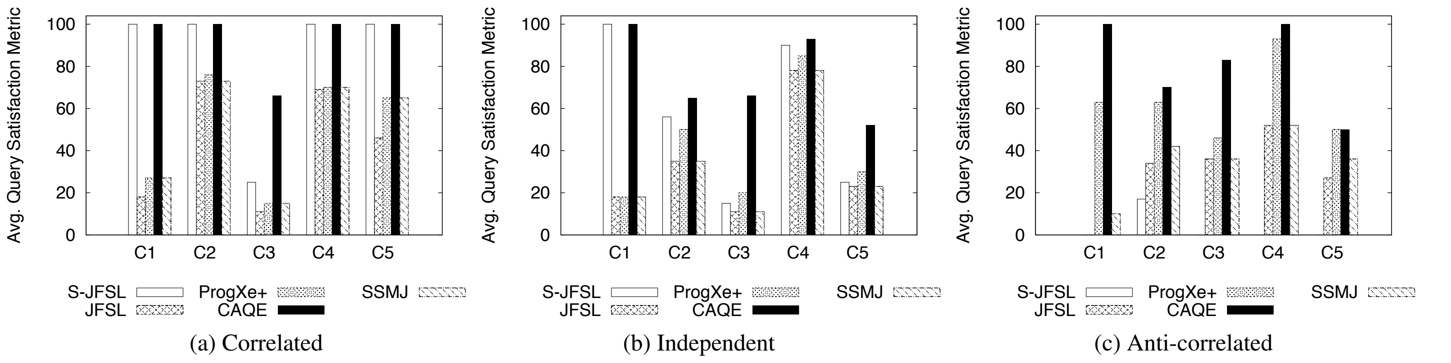


Figure 9: Comparing the Avg. Contract Satisfaction Metric for CAQE, S-JFSL, JFSL, ProgXe+ and SSMJ; $|\mathcal{S}_Q| = 11$ $N = 500K$

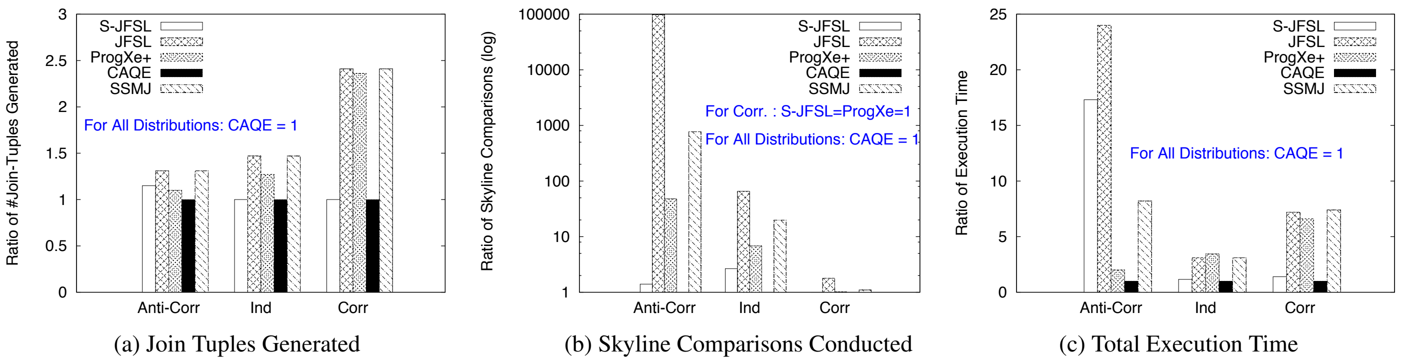


Figure 10: Comparing the Statistics Measured for S-JFSL, JFSL, *ProgXe+* and SSMJ Against CAQE ($|\mathcal{S}_Q| = 11, N = 500K, C2$)

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.

Correlated datasets are tailor made for skyline algorithms since a handful of join tuples can dominate the entire result space [5]. Therefore, for such datasets we set the contract parameters t_{C1} = t_{C3} = $10s$ and $n_{i,j} = 1s$. In Figure 9.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 of 12 seconds has a 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 an 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 9.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.

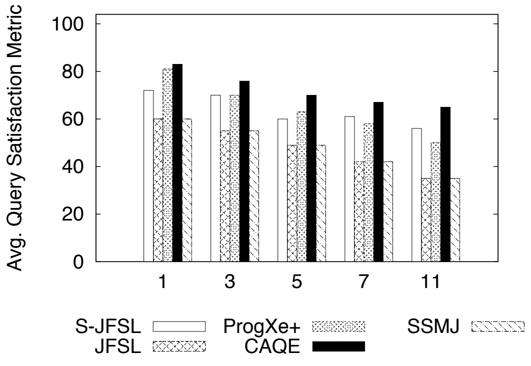
The anti-correlated data distribution is the most resource intensive dataset for a skyline algorithm. This is evident from the fact that a 4-d skyline has 75K+ join tuples in the final skyline. Given the expensive nature of this dataset we set t_{C1} = t_{C3} = 30 minutes and $n_{i,j}$ = 10 minutes. For all contract models except $\{C2\}$, skyline results returned by JFSL have no value to users of the workload queries. In Figure 9.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.

7.3 Comparing CPU and Memory Utilization

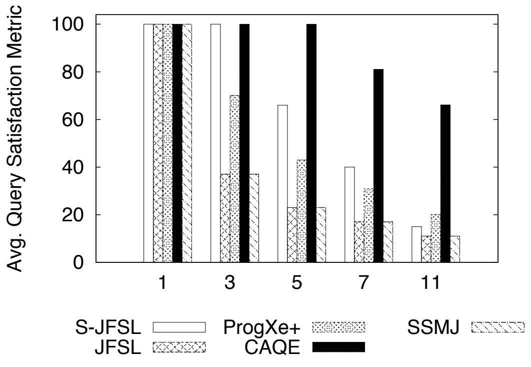
The CPU and memory utilization of the skyline over join algorithm is directly related to the number of intermediate tuples generated by the join operation as well as the expensive pairwise dominance comparison needed to evaluate the final skyline. In Figures 10.a - 10.c we illustrate that employing a 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 10.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.



(a) Contract Model: C2



(b) Contract Model: C3

Figure 11: Increasing Number of Queries in the Workload

7.4 Increasing Size of Workload

Next we measure the effectiveness of the techniques for varying workload sizes. Due to space limitations, we restrict the discussion to independent data distribution datasets and to contracts $\{C2, C3\}$ which are the strictest contract models presented in Table 2 (Section 7.1). In Figures 11 and 11.b, as the number of workload queries increases the average satisfaction of each query in the workload drops proportionally. In Figure 11.a and all workload sizes, due to the nature of the logarithmic decay function of contract $\{C2\}$, none of the techniques can achieve the optimal 100% contract satisfaction. However, as the workload size increases, CAQE's adaptive execution strategy enables it to have the smallest drop in performance of 20% in comparison to the 36% and 38% drop for ProgXe+ and SSMJ respectively. In Figure 11.b for the contract $\{C3\}$ 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 11.b that existing techniques suffer from a steep drop in performance (up to 85% regression). In contrast, CAQE's novel execution strategy enables it to only experience a relatively smaller drop of 30% in query satisfaction.

8. RELATED WORK

Skyline Algorithms over Single Relation. The majority of research on skylines has focused on the efficient computation of a skyline over a single relation [2, 3, 6, 16, 23]. This can be broadly categorized as *non-index* and *index-based* solutions. *Block nested loop* (BNL) [3] 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) [6] improves on BNL by first sorting the input data by a monotonic function. Nearest Neighbor (NN) [16] and Branch & Bound Search (BBS) [23] are index-based algorithms.

Subspace Skylines over Single Relation. A *Skyline Cube* is defined as containing skylines results for all combinations of skyline dimensions [24, 36]. This is reminiscent of the precomputed *data cube* technique in data warehousing [12]. Each combination of dimensions is termed as *subspace* [24, 34, 36] and for a given set of d dimensional objects, there are $2^d - 1$ subspaces representing the preferences of various users. [36] presented efficient algorithms that can compute skylines over all $2^d - 1$ subspaces in the *skycube*. Alternatively, [24] studied the problem of capturing the semantics of subspace skylines by identifying decisive subspaces. To provide an indexed based solution to the skycube problem, [30] proposed the *SUBSKY* technique by using a single B-Tree. To efficiently support subspace skyline queries in databases that receive frequent updates [34] presented an alternative *compressed skycube* approach. However, these techniques ignore (1) multi-relational skylines, and (2) do not support QoS sensitive query evaluation – both now tackled by our work.

Skylines over Join Queries. Existing techniques [3, 14, 15, 21, 27, 31] process a *single* skyline over join query, while ours is the first effort at processing *multiple skyline over join* query. Table 3 summarizes the differences between our approach versus the state-of-the-art skyline techniques.

	Skyline-Over Join	Multiple Queries	Progressive	Supports User QoS
SkyCube [36]		✓	✓	
BUS, TDS [24]		✓	✓	
PruningJoin$^+$ [17]	✓			
SAJ [17]	✓			
Sort-Based [14,31] [15,21]	✓		✓	
ProgXe+ [27]	✓		✓	
✠ **Our Approach**	✓	✓	✓	✓

Table 3: Summary of the related work

Quality of Service. In *computer networking*, QoS defines varying levels of services for applications and types of data. Applications such as Voice over IP and streaming multimedia must ensure a certain level of user experience by reducing packet loss. This is accomplished by reserving network capacity based on bandwidth, delay, and error rates [25]. In *streaming databases*, to provide real-time responses, [19, 35] 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, [35] sheds data from incoming streams to handle load and meet the desired QoS. Second, they do not support the more complex non blocking queries such as skyline over join queries.

9. CONCLUSION

In this work, we introduce an NP-Hard Contract-MQP problem that aims to optimize the processing of concurrent decision support queries each augmented by a quality of service contract. In this effort, we design a rich model to express progressiveness contracts and accompany it with an effective measure for determining the run-time satisfaction of these contracts. We propose **C***ontract*-**A***ware* **Q***uery* **E***xecution* (**CAQE**) framework that unblocks query processing by using a multi-granular execution strategy. CAQE's execution model enables us to expose and then exploit previously ignored opportunities for fine-grained sharing among complex queries. Our feedback driven execution strategy continuously monitors the run-time satisfaction of the workload and aggressively takes corrective steps to maximally satisfy the contracts. We demonstrate the superiority of CAQE over existing multi-query processing techniques by showing that in many cases CAQE is 2 fold more effective in satisfying the QoS contracts.

Acknowledgment

This work is supported by the National Science Foundation under Grant No. IIS-0633930, CNS-305258 and CRI-0551584. We thank David Sampson, Dr. Lyublena Antova, Dr. Rhonda Baldwin, and Dr. Florian Waas for their valuable feedback.

10. REFERENCES

[1] C. C. Aggarwal. Towards meaningful high-dimensional nearest neighbor search by human-computer interaction. In *ICDE*, pages 593–604, 2002.

[2] I. Bartolini, P. Ciaccia, and M. Patella. Salsa: computing the skyline without scanning the whole sky. In *CIKM*, pages 405–414, 2006.

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

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

[5] S. Chaudhuri, N. N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE*, pages 64–73, 2006.

[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816, 2003.

[7] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *PODS*, pages 59–70, 2001.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[9] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *ICDE*, pages 503–511, 2001.

[10] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[11] L. Golab and M. T. Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD*, pages 658–669, 2005.

[12] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.

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

[14] W. Jin, M. D. Morse, J. M. Patel, M. Ester, and Z. Hu. Evaluating skylines in the presence of equijoins. In *ICDE*, pages 249–260, 2010.

[15] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Prefjoin: An efficient preference-aware joinoperator. In *ICDE*, pages 995–1006, 2011.

[16] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[17] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[18] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004.

[19] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *BIRTE*, pages 143–156, 2006.

[20] H. Min and S.-Q. Zheng. Time-space optimal convex hull algorithms. In *SAC*, pages 687–693, 1993.

[21] M. Nagendra and K. S. Candan. Skyline-sensitive joins with lr-pruning. In *EDBT*, pages 252–263, 2012.

[22] S. Narayanan and F. Waas. Dynamic prioritization of database queries. In *ICDE*, pages 1232–1241, 2011.

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

[24] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[25] J. Pongsajapan and S. H. Low. Reverse engineering tcp/ip-like networks using delay-sensitive utility functions. In *INFOCOM*, pages 418–426, 2007.

[26] H. Qu, J. Xu, and A. Labrinidis. Guiding personal choices in a quality contracts driven query economy. In *PersDB Workshop, SIGMOD Conference*, 2009.

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

[28] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *TKDE.*, 2: 2:262–266, 1990.

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

[30] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, page 65, 2006.

[31] A. Vlachou, C. Doulkeridis, and N. Polyzotis. Skyline query processing over joins. In *SIGMOD*, pages 73–84, 2011.

[32] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *PVLDB*, 2(1):898–909, 2009.

[33] S. Wu, B. C. Ooi, and K.-L. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*, pages 651–662, 2010.

[34] T. Xia and D. Zhang. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In *SIGMOD*, pages 491–502, 2006.

[35] Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[36] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.