# Refinement Driven Processing of Aggregation Constrained Queries

Manasi Vartak [1,a], Venkatesh Raghavan [2,b], Elke Rundensteiner [3,c], Samuel Madden [4,a]

[a] *Massachusetts Institute of Technology,* [b] *Pivotal Inc.,* [c] *Worcester Polytechnic Institute*

[1] mvartak@mit.edu, [2] vraghavan@pivotal.io, [3] rundenst@cs.wpi.edu, [4] madden@csail.mit.edu

## ABSTRACT

Although existing database systems provide users an efficient means to select tuples based on attribute criteria, they however provide little means to select tuples based on *whether they meet aggregate requirements*. For instance, a requirement may be that the cardinality of the query result must be 1000 or the sum of a particular attribute must be $<$ \$5000. In this work, we term such queries as "Aggregation Constrained Queries" (ACQs). Aggregation constrained queries are crucial in many decision support applications to maintain a product's competitive edge in this fast moving field of data processing. The challenge in processing ACQs is the unfamiliarity of the underlying data that results in queries being either too strict or too broad. Due to the lack of support of ACQs, users have to resort to a frustrating trial-and-error query refinement process. In this paper, we introduce and define the semantics of ACQs. We propose a refinement-based approach, called ACQUIRE, to efficiently process a range of ACQs. Lastly, in our experimental analysis we demonstrate the superiority of our technique over extensions of existing algorithms. More specifically, ACQUIRE runs up to 2 orders of magnitude faster than compared techniques while producing a 2X reduction in the amount of refinement made to the input queries.

## 1. INTRODUCTION

Databases provide a number of ways to efficiently select tuples of interest to the user by constraining attributes of individual tuples, for instance, return tuples that meet the criteria price $<$ \$50, join results between tuples in table A and table B that match on attribute "id," etc. However, little effort has been focused on a means of selecting tuples based on *whether they satisfy aggregate constraints*. For instance, select tuples with average *price* $<$ \$10, number of tuples = 1000, etc. The ability to apply aggregate constraints along with constraints on tuples' individual attribute values is important in many applications as illustrated below.

- In advertising campaigns (such as Example 1), *the budget restricts the number of users that can be reached* [4]; as a result, the campaign manager must select users based not only on demographics but also whether the total number of users (i.e. the COUNT) is within the budget limit.
- In a supply chain application, *a requirement on the total number of parts* to be ordered from suppliers translates to a constraint on the sum of the number of parts available with each supplier (Example 2). As a result, queries must place constraints not only on part specifications but also the SUM of the parts available.
- When analyzing large data sets through aggregates [15], users often want to identify what input tuples produced outliers in aggregate values (e.g. select patients who had extremely high average cost). In this case, the user would like to place constraints on the AVG aggregate.

**Example** 1. *HighStyle Designers would like to run a Facebook* [1] *ad campaign to get more users to "like" their page. The campaign budget of* \$10,000 *will allow HighStyle to reach 1 million customers. Therefore, when the campaign manager, Alice, selects target users, she must not only constrain her search based on customer demographics but also based on the total number of customers who must be reached. This situation thus calls for an* **"Aggregation Constrained Query"** *(ACQ).*

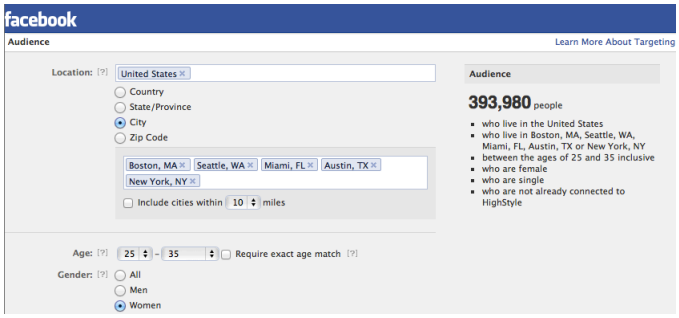



**Figure 1: Facebook Ad Creation Interface: Allows specifying demographic criteria and view estimate of audience size.**

Figure 1 shows the Facebook's Advertising Interface[2] that allows campaign manager Alice to select target users for her ad. In terms of SQL, Alice has to run the following query:

```
Q1: SELECT * FROM Users
WHERE location in ('Boston', 'New York',
'Seattle',  'Miami', 'Austin') AND
(gender = 'Women')  AND (25 <= age <= 35)
AND (education = 'CollegeGrad')
```

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

[2] https://www.facebook.com/ads/create/

```
AND (relationshipStatus = 'Single')
AND (interests IN {'Retail', 'Shopping'})
```

**Need for Query Refinement.** For Alice's above query, Facebook estimates the reach to be 393,980 users, i.e. only 40% of the required 1 million users. While the results of query Q1 precisely satisfy Alice's selection predicates, they are far from meeting her aggregate constraints. In fact, selection and aggregation constraints are orthogonal in most cases. As a result, we need to refine various query predicates in order to meet the aggregate constraints.

**Current Approach.** In existing systems Alice has to manually alter her criteria to encompass more users while ensuring that the semantics of her query are not altered. While some selection criteria (e.g. gender and shopping interest) may be fixed, Alice can try potentially infinite refinements of her predicates such as target consumers in additional cities; alter age range; relax relationship status; or any combination of the above. Repeatedly altering the original query and having its size estimated is not only inefficient for the backend, but the process is tedious and frustrating for Alice.

**Desired User Experience.** A much better user experience can be provided if Alice was allowed to specify her (1) demographic criteria (query), and (2) aggregate constraints, and the database engine can then execute variations of the input query such that the aggregate constraints are met. The output of such a search would be a set of refined queries that change Q1 as little as possible while meeting the aggregate constraints (in our case the audience size). Alice would then simply pick the query that best meets her selection criteria.

In this paper, we encode ACQs by introducing two SQL keywords CONSTRAINT and NOREFINE, where CONSTRAINT captures the aggregate constraint and NOREFINE specifies whether the predicate should not be refined. The encoded Query Q1 is:

```
Q1': SELECT * FROM Users
CONSTRAINT COUNT(*)=1M
WHERE location in ('Boston', 'New York',
'Seattle', 'Miami', 'Austin') AND
(gender = 'Women') NOREFINE  AND (25 <=age<=35)
AND (education = 'CollegeGrad')
AND (relationshipStatus = 'Single')  AND
(interests IN {'Retail', 'Shopping'}) NOREFINE;
```

Running Q1' will automatically generate alternate queries that produce 1M customers and alter Q1 as little as possible.

**Example** 2. *HybridCars Co. would like to place an order for 100,000 units of a burnished steel part having specific size, wholesale price less than* $1000*, and from suppliers who have a low account balance. On the TPC-H benchmark, HybridCars runs query Q2 to find the suppliers with whom to place the order.*

```
Q2: SELECT * FROM supplier, part, partsupp
WHERE (s_suppkey = ps_suppkey) AND
(p_partkey = ps_partkey) AND
(s_acctbal < 2000)
AND (p_retailprice < 1000) AND (p_size = 10)
AND (p_type = 'SMALL BURNISHED STEEL')
```

As in Example 1, this situation calls for an ACQ as we would like to constrain the total number of available parts, i.e. *sum of the number of parts available per supplier* (i.e. SUM(ps_availqty)) in addition the select predicates. We can encode the ACQ as Q2' to produce alternate refined queries. As before, the NOREFINE keyword associated with p_type and p_size indicate that these predicates cannot be altered.

```
Q2': SELECT * FROM supplier, part, partsupp
CONSTRAINT SUM(ps_availqty) >= 0.1M
WHERE (s_suppkey = ps_suppkey) NOREFINE AND
(p_partkey = ps_partkey) NOREFINE AND
(p_retailprice < 1000) AND (s_acctbal < 2000)
AND (p_size = 10) NOREFINE AND
(p_type = 'SMALL BURNISHED STEEL') NOREFINE
```

Building a system to execute ACQs is challenging because the number of possible refined queries is exponential in the number of predicates. Hence an exhaustive search of all possible queries is prohibitively expensive. Moreover even for aggregates such as COUNT, finding a query that meets its constraint is an NP-Hard problem [1]. In this paper, we limit ourselves to ACQs with numerical select and join predicates, and aggregates that satisfy the *optimal substructure* property (Section 2). Additionally, we focus on the problem of *expanding predicates* to meet constraints, rather than the inverse problem of shrinking queries returning too many tuples.

**Contributions.** We propose a technique to efficiently execute ACQs and our contributions are summarized as follows:

- We introduce and define semantics of a new class of queries called an **Aggregation Constrained Query** (ACQ). These special purpose queries are of value in real-world applications and are amenable to clever execution techniques.
- We propose a technique called ACQUIRE to execute ACQs via query refinement. ACQUIRE auto-generates alternative refined queries that minimize changes to the original query while meeting aggregate constraints.
- We combine the building blocks of breadth-first-search and dynamic programming in a novel way to elegantly and efficiently re-use query results. We call this *Incremental Aggregate Computation* (Section 5).
- We propose sensible default query refinement scoring and aggregate error functions. The design principle of ACQUIRE is general and therefore we allow user defined predicate refinement scoring and aggregate error functions. The functions used in this work are merely sensible defaults.
- Our experimental analysis on TPC-H dataset demonstrates that ACQUIRE consistently out-performs extensions to current techniques by up to 2 orders of magnitude. Moreover, queries recommended by ACQUIRE are on average closer to the original query by a factor of 2X more than the compared techniques (Section 8).

# 2. PRELIMINARIES

## 2.1 SQL extension for ACQs

We propose to capture ACQs by using two keywords: CONSTRAINT to describe the aggregate constraint and NOREFINE to indicate that a predicate should not be refined. By default, we assume that all predicates can be refined.

```
SELECT * FROM Table1, Table2 ...
CONSTRAINT AGG(attribute) Op X
WHERE Predicate1 AND Predicate2 ...
AND Predicate_i NOREFINE AND Predicate_j
AND ...Predicate_n NOREFINE
```

The aggregate constraint is of the form $AGG(attribute)$ $Op$ $X$, where AGG is a standard (COUNT, SUM, MIN, MAX, AVG) or user defined aggregate function, $X$ is a positive number and $Op$ is a comparison operator $(=, \leq, <, \geq, and >)$. In this work, we focus on the problem of expanding predicates to meet constraints, rather than the inverse problem of shrinking queries returning too many tuples, we therefore limit the comparison operation to $=, \geq, and >$. Henceforth for illustrative purposes only we assume that the aggregate constraint has an equality condition.

## 2.2 Query Representation

In this work, we focus on queries with numeric select, project and join predicates of the form $Q = P_1 \wedge \ldots \wedge P_d$, where $P_i$'s is a predicates on relations $R_1 \ldots R_k$. To illustrate consider query Q3 with one select and one join predicate.

```
Q3: SELECT * FROM A, B
    WHERE A.x=B.x AND B.y < 50
```

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 relations $R_1 \ldots R_k$ 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, if the minimum value of $B.y$ is 0, the predicate $(B.y < 50)$, in $Q3$ 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, $(B.y > 10) \wedge (B.y < 50)$. This enables the refinement of one or both sides of the range predicate. 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 $\Delta(({P_i}^{\mathcal{F}})_1, ({P_i}^{\mathcal{F}})_2)$, where $({P_i}^{\mathcal{F}})_1$ and $({P_i}^{\mathcal{F}})_2$ are separate predicate functions and $\Delta$ is the function measuring distance between them. Therefore, join predicate $A.x = B.x$ in $Q3$ is decomposed into $({P_i}^{\mathcal{F}})_1 = A.x$ and $({P_i}^{\mathcal{F}})_2 = B.x$. ${P_i}^{\mathcal{I}} = (0, 0)$ signifies that values of the two functions must match exactly. For each predicate $P_i$, we also store a boolean value indicating whether the predicate can be refined. Recall that ACQ's contain an aggregate function that specifies the target value of an aggregate over the output result. We denote the target, or expected, aggregate value as $A_{exp}$ and actual aggregate value returned by the query as $A_{actual}$.

## 2.3 Measuring Query Refinement Quality

We define a query refinement score to measure the change that has been made to the original query to obtain the refined query. 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 the **predicate refinement score**, denoted as $PScore_i(Q, Q')$, is measured as the percent departure of $({P_i}^{\mathcal{I}})'$ from ${P_i}^{\mathcal{I}}$ (Equation 1). Note that if $({P_i}^{\mathcal{I}})_{min} = ({P_i}^{\mathcal{I}})_{max}$, $PScore_i(Q, Q') = 0$. 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 2).

$$PScore_i(Q, Q') = \frac{|({P_i}^{\mathcal{I}})_{min} - ({P_i}^{\mathcal{I}})'_{min}| + |({P_i}^{\mathcal{I}})_{max} - ({P_i}^{\mathcal{I}})'_{max}|}{|({P_i}^{\mathcal{I}})_{max} - ({P_i}^{\mathcal{I}})_{min}|} \cdot 100 \quad (1)$$

$$\overline{PScore(Q, Q')} = (PScore_1(Q, Q') \ldots PScore_d(Q, Q')) \quad (2)$$

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 [7] to calculate $QScore(Q, Q')$. Equation 3 shows the calculation of $QScore(Q, Q')$ using the default $L_1$ norm.

$$L_1 : \; QScore(Q, Q') = \Big( \sum_{i=1}^{d} PScore_i(Q, Q') \Big) \quad (3)$$

**Example** 3. *Consider the following refinement to Q3.*

```
Q3': SELECT * FROM A,B
     WHERE A.x = B.x AND B.y < 60
```

The refined query $Q3'$ expands the range of acceptable values for $B.y$ from (0, 50) to (0, 60). Therefore, $Q3'$ is represented as $\overline{PScore(Q3, Q3')}$= (0, $\frac{60-50}{50-0} \cdot 100$) and has $QScore(Q3, Q3')$=20 for the $L_1$ norm.

## 2.4 Refining Join Predicates

The advantage of representing predicates as functions $({P_i}^{\mathcal{F}})$ and intervals $({P_i}^{\mathcal{I}})$, and defining refinement as the change in the predicate interval, is that join refinement can be expressed and operated on in the same way as select predicates. For instance, a query with $\overline{PScore(Q3, Q3'')}$= (10, 20) indicates that the join predicate in Q3 has been refined by 10 to become $\|A.x - B.x\| \leq 10$ and that the $B.y$ predicate has been refined by 10 units. Thus, the algorithm can be applied unchanged for select as well as join queries.

## 2.5 Measuring Aggregate Error

To measure the difference between the expected aggregate value $A_{exp}$ and the actual aggregate value $A_{actual}$, we use a relative error measure defined as:

$$Err_A = \frac{\|A_{exp} - A_{actual}\|}{A_{exp}} \quad (4)$$

This measure is appropriate for aggregates such as COUNT or AVG; however, a hinge-function that only penalizes errors on one side is appropriate for SUM, MIN and MAX.

$$Err_A = \begin{cases} (A_{exp} - A_{actual}) & if\ A_{exp} > A_{actual} \\ 0 & otherwise \end{cases}$$

## 2.6 Optimal Substructure Property

In this work, we limit ourselves to aggregate functions that either (a) have the **o**ptimal **s**ubstructure **p**roperty (OSP), or (b) can be broken down into functions that satisfy the OSP. Consider any two queries Q1 and Q2 such that all the results of query Q2 are also results of query Q1 (Q1 *contains* Q2). An aggregate is said to satisfy the OSP if the value of the aggregate for the results of Q1 can be computed without re-executing part or whole of the query Q2.

For instance, the COUNT aggregate is said to satisfy the OSP because given queries Q1 and Q2 as defined above, the value of COUNT for Q1 can be computed by adding the value of COUNT for Q2 to the value of COUNT for the query (Q1-Q2). SUM, MIN

and MAX similarly satisfy the OSP, and can be addressed by our technique. AVG, another common aggregate, can be broken down into two aggregates SUM and COUNT which have the optimal substructure property in turn, and therefore AVG can also be addressed by our technique. STDDEV, on the other hand, does not satisfy the OSP because even if the STDDEV for Q2's results are known, the results of Q2 must be re-analyzed to compute STDDEV for Q1.

## 2.7 Problem Definition

Given a query $Q$ and a desired aggregate value $A_{exp}$, the problem of **Aggregation Constrained Query Execution** consists of refining $Q$ to produce alternate queries $Q'$ that produce the aggregate value $A_{exp}$ while changing $Q$ as little as possible. Formally, we can state it as follows:

DEFINITION 1. *Given database $\mathcal{D}$, query $Q$, desired aggregate value $A_{exp}$, an aggregate error threshold $\delta$, and refinement threshold $\gamma$, ACQ finds a set of refined queries $Q'$ s.t. (a) the actual aggregate value for $Q'$, $A_{actual}$, satisfies: $Err_A \leq \delta$, and (b) $\|QScore(Q, Q_i) - QScore_{opt}\| \leq \gamma$, where $Q_i \in Q'$, $QScore_{opt}$= min $\{QScore(Q, Q'_j) | \forall$ valid query refinements $Q'_j$ s.t. $(Err_A \leq \delta)\}$.*

Since the problem of attaining the required aggregate value is NP-hard, we cannot provide formal guarantees about constraint (a) in Definition 1. However, as demonstrated in our experiments (Section 8), our algorithm ensures that the constraint is met practically every time. Our proximity-driven refinement technique guarantees that ACQUIRE will always meet constraint (b) in Definition 1.

We now turn our attention to evaluating these ACQs. As noted in the introduction, in this paper, our major focus is on queries that undershoot the aggregate constraint, however, we show in Section 7 how ACQUIRE can be extended to handle queries that overshoot the constraint. Furthermore, although we use COUNT as the aggregate of choice for all discussions, it is straightforward to support other aggregates with our technique and we note any changes to the algorithm that are required for doing so.

# 3. ACQUIRE: AN OVERVIEW

Given a query, the desired aggregate value, and acceptable result thresholds, ACQUIRE produces a set of refined queries that minimize changes to the original query but also satisfy the aggregate constraint. In formulating this set of refined queries, ACQUIRE adopts the strategy of *Expand and Explore* to iteratively $expand$ the original query and to $explore$ refined queries with respect to aggregate values. The $expand$ phase ensures that refined queries satisfy the refinement threshold and that queries with smaller refinements are produced before those with larger refinements. Thus, once ACQUIRE finds a query satisfying the aggregate constraint, it need not examine queries with larger refinements. The $explore$ phase on the other hand efficiently computes aggregate values for refined queries via an incremental aggregate computation algorithm. We delegate all actual query execution tasks to an *evaluation layer*, which in this case is Postgres. However, the evaluation layer is modular and can be replaced with other techniques such as estimation, and/or sampling. Our incremental aggregate computation algorithm exploits dependencies between refined queries and the optimal substructure property so that for each query, ACQUIRE must only execute a small sub-query and then simply use our recursive model to combine results from previous queries. Together, these two techniques ensure that once a query Q has been executed, any query Q' that *contains* Q will not have to re-execute Q. As a result, ACQUIRE can evaluate a large number of refined queries at a cost that is a fraction of the execution time for a single query. Figure 2 shows the system architecture described above.

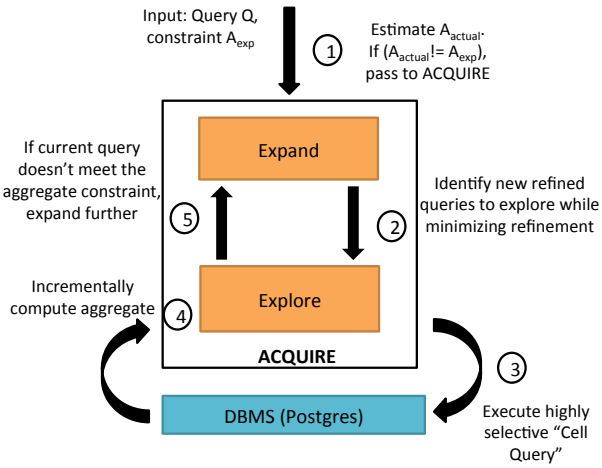



**Figure 2: System Architecture of ACQUIRE**

# 4. PHASE I: EXPAND

As described in the previous section, the *Expand* phase of ACQUIRE 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.

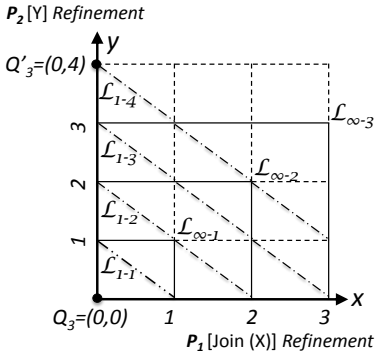



**Figure 3: Refined Space and Generation of Refined Queries**

To meet the above query generation goals, ACQUIRE uses an abstraction called the **Refined Space** to represent all refined queries. Given an original query $Q$ having $d$ predicates, 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 $L_1$ norm is used to compute $QScore$. $Q'$ would then be represented in $RS(Q)$ as $(u_1, u_2, \ldots, u_d)$ where $u_i = (PScore_i(Q, Q'))$ $\forall i = 1, \ldots, d$, making $QScore(Q, Q')$ = $(\sum_{i=1}^{d} u_i)$. Conversely, every point in the refined space $(u_1, u_2, \ldots, u_d)$ corresponds to some query $Q'$ with $PScore_i(Q, Q') = u_i$. Therefore, any d-dimensional hyper-rectangle on $RS(Q)$ also corresponds to a query.

ACQUIRE divides $RS(Q)$ into a multi-dimensional grid with step-size $\frac{\gamma}{d}$ to avoid an exhaustive search of $RS(Q)$ and to stay within the proximity threshold, as illustrated by Theorem 1. Each query on the multi-dimensional grid is called a *grid query*.

**Theorem** 1. *Suppose the original query is $Q$ and $Q_{opt}$ is the optimal query meeting the aggregate 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 $Q_{opt} = \{u_1, u_2, \ldots, u_d\}$ lie in some 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:
$|{u_1}^p - {u'_1}^p| + |{u_2}^p - {u'_2}^p| + \ldots + |{u_d}^p - {u'_d}^p| \leq \frac{\gamma}{d} \cdot d = \gamma$

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

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

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

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

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

Figure 3 depicts the refined space abstraction for query Q3 assuming $\gamma = 10$. Since Q3 has two predicates, step-size=5 and $RS(Q3)$ is a 2-dimensional space with the axes respectively measuring the refinements along the select and join predicates. A refined query like Q3' having $\overline{PScore(Q3, Q3')} = (0, 20)$ is represented as (0, 4) in $RS(Q3)$.

The second goal of the *Expand* phase is to generate refined queries in order of increasing refinement. ACQUIRE achieves this goal by producing queries close to the origin in $RS(Q)$ before those far from it. In particular, the *Expand* phase uses breadth-first search to generate refined queries in layers where queries in a given *query-layer* have the same $QScore$. Consequently, for all $L_p$ norms except $L_\infty$, 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 $L_\infty$, however, query-layers are L-shaped and intersect each axis at $k^p$. Figure 3 shows query-layers for Q3 assuming the $L_1$ and $L_\infty$ norms. Beginning with the query-layer with refinement 0, ACQUIRE generates all grid queries in the current query-layer. If no query from the current layer satisfies the aggregate constraint, ACQUIRE 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, ACQUIRE can stop immediately after a query is found to meet the required constraint, thus reducing the number of queries examined by ACQUIRE. Algorithms 1 and 2 respectively describe the pseudo code for generating queries using the $L_p$ and $L_\infty$ norms. The $L_p$ algorithm generates query-layers using a breadth-first search while the $L_\infty$ norm sequentially enumerates queries in the given layer.

**Algorithm 1** GetNextQuery(Queue *queryQue*)

```
1: int[] Q_curr = queryQue.Pop()  //Indexed from 1
2: for i = 1, ..., d do
3:     Q_next ← GetNextNeighbor(i)  //Increment i-th dimension
       of Q_curr by stepsize
4:     if (!queryQue.Contains(Q_next)) then
5:         queryQue.Push(Q_next)
6: return Q_curr
```

**Algorithm 2** GetNextQuery(Queue queryQue, int currRef)

```
1: if (!queryQueue.Empty()) then
2:     return queryQue.Pop()
3: else
4:     Query Q_new = 0
5:     for i = 1, ..., d do
6:         Q_new[i] = currRef; queryQue.Push(Q_new)
7:         while Q_new != null do
8:             IncrementQuery(Q_new, i, currRef) // enumerate
               queries with i-th dim fixed at currRef and others < cur-
               rRef
9:             queryQue.Push(Q_new)
```

**Theorem** 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 $L_\infty$ 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 $|E| = d \cdot |V|$.

# 5. PHASE II: EXPLORE

The *Explore* phase of ACQUIRE is responsible for efficiently computing the aggregate values of queries produced in the *Expand* phase. For this purpose, we introduce a light-weight query execution methodology based on a novel, efficient incremental query execution algorithm that 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 aggregate by intelligently combining partial results from previous queries. ACQUIRE guarantees that a query is executed at most once, irrespective of how many queries contains it.

## 5.1 Incremental Aggregate Computation

The principle underlying our query execution algorithm is that refined queries often share results. Therefore, once a query result has been evaluated 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** 3. *If refined query $Q'$ is contained within refined query $Q''$: (1) all results of $Q'$ also satisfies $Q''$. (2) $Q'$ is guaranteed to be generated before $Q''$ in the Expand phase.*

*Proof:* Let tuple $\tau$ satisfy $Q'$. (1) By Equation 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

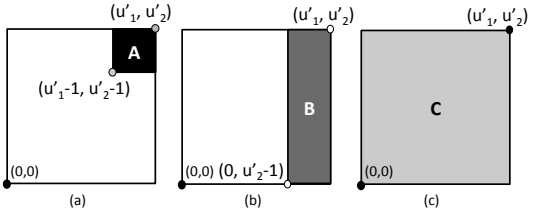



**Figure 4: Sub-queries of a 2-D query**

(2), from the definition of contained queries, $QScore(Q', Q) \leq QScore(Q'', Q)$. Therefore, by Theorem 2, the *Expand* phase will produce $Q'$ before $Q''$. ■

Since all contained queries are produced and executed before those containing them, ACQUIRE can extensively use previously-generated query results. In particular, ACQUIRE exploits the concept of query containment by 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.

### 5.1.1 Query Decomposition

Consider query $Q'$ with $d$ predicates, represented as point $(u'_1, \ldots, u'_d)$ in the refined space. In addition to $Q'$, ACQUIRE defines $d$ specialized sub-queries contained within it, giving $d+1$ queries in all. Figure 4 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 $RS(Q)$ 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 5, 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**, respectively.

In a d-dimensional refined space, the $d+1$ sub-queries, called $O_1$, $O_2$,..., $O_{d+1}$, can be formally defined as shown in Equations 5-8. 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 a unit length away from $(u'_1, \ldots, u'_d)$ on all dimensions (Equation 5). 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 6). In general, the lower bound of the $j^{th}$ 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 $O_i(u'_1, \ldots, u'_d)$.

$$O_1 = ((u'_1 - 1, \ldots, u'_d - 1), (u'_1, \ldots, u'_d)) \quad (5)$$

$$O_2 = ((0, u'_2 - 1, \ldots, u'_d - 1), (u'_1, \ldots, u'_d)) \quad (6)$$

$$O_j = ((0, 0, \ldots, 0, u'_j - 1, \ldots, u'_d - 1), (u'_1, \ldots, u'_d)) \quad (7)$$

$$O_{d+1} = ((0, \ldots, 0), (u'_1, \ldots, u'_d)) \quad (8)$$

By decomposing a query into the sub-queries defined above, we can reuse previously obtained results. To illustrate, consider Figure 6.a where the 2-D query is 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 6.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

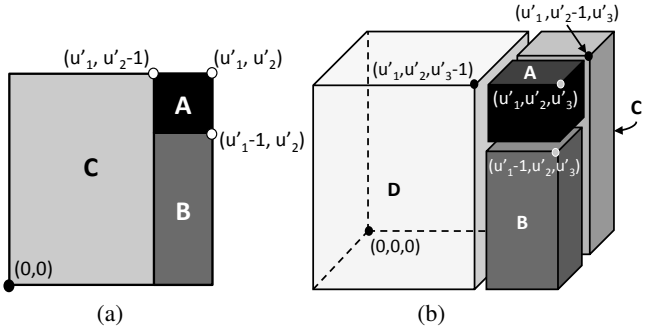



**Figure 6: Query Decomposition: (a) 2-D (b) 3-D**

$Pillar(u'_1 - 1, u'_2, u'_3)$, 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:

$$\mathbf{2-predicate\ Query}: \quad (9)$$

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

$$\mathbf{3-predicate\ Query}: \quad (10)$$

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

$$\mathbf{d-predicate\ Query}: \quad (11)$$

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

Thus, if the aggregates for the $(d+1)$ sub-queries have been pre-computed, the aggregate of query $Q'$ is the mere addition[1] of these sub-aggregates. We must store only the aggregate *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 3 guarantees that these queries would have been produced and hence executed before investigating $Q'$. As a consequence, ACQUIRE must only execute the cell sub-query and can directly reuse aggregates of the remaining sub-queries.

### 5.1.2 Recursive Aggregate Computation

Query decomposition assumes that the aggregates for the $d+1$ sub-queries have already been computed. But independently determining aggregates of these sub-queries is redundant. Instead, we present a recursive strategy to calculate the aggregates of the sub-queries in constant time. Reconsider Figure 6 focusing now on the relationship between sub-queries. We observe that for 2-predicate sub-queries (Figure 6.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 6.b, we have three similar recurrences as shown below.

[1]For aggregates like MIN/MAX, addition is replaced by the corresponding MIN/MAX function, while AVERAGE = SUM/COUNT. SUM and COUNT aggregates are computed and stored separately. AVERAGE is computed from these values as required.

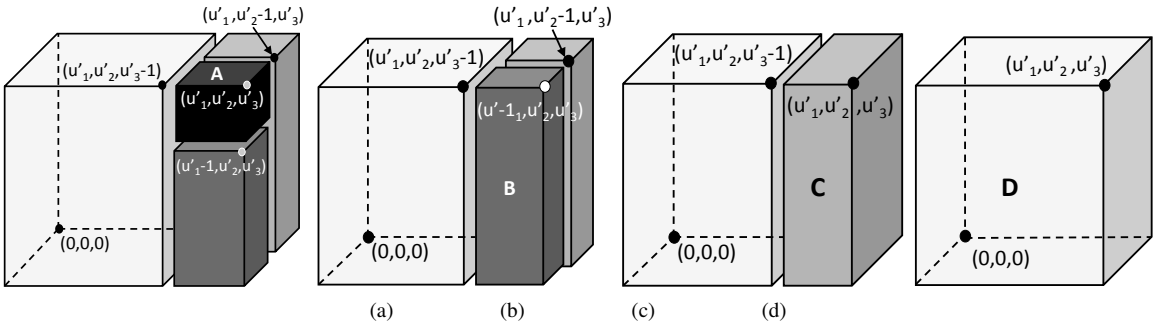



**Figure 5: Sub-queries of a 3-predicate query**

$$\mathbf{2-Recurrences}: \quad (12)$$

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

$$Wall(u'_1, u'_2) = Pillar(u'_1, u'_2) + Wall(u'_1, u'_2 - 1) \quad (13)$$

$$\mathbf{3-Recurrences}: \quad (14)$$

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

$$Wall(u'_1, u'_2, u'_3) = Pillar(u'_1, u'_2, u'_3) + Wall(u'_1, u'_2 - 1, u'_3) \quad (15)$$

$$Block(u'_1, u'_2, u'_3) = Wall(u'_1, u'_2, u'_3) + Block(u'_1, u'_2, u'_3 - 1) \quad (16)$$

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

$$O_i(u'_1, \ldots, u'_d) = O_{i-1}(u'_1, \ldots, u'_d) + \quad (17)$$
$$O_i(u'_1, u'_2, \ldots, u'_{i-1} - 1, \ldots, u'_d) \; where \; i = 2, \ldots, d+1$$

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

### 5.1.3 Aggregate Computation Algorithm

Algorithm 3 takes as input the query $Q'(u'_1, \ldots, u'_d)$ being investigated and produces its aggregate. For this, Algorithm 3 first computes the aggregate of the $Cell(u'_1, \ldots, u'_d)$, and then iteratively applies the recurrence in Equation 17 to compute aggregates of the remaining sub-queries. The function `ExecuteCellQuery` is used to compute the aggregate over a single input cell by issuing a query to the evaluation layer.

**Algorithm 3** ComputeAggregate(Query $Q_{curr}$, int $d$)

1: int[$d+1$] $A_{curr}$ // All arrays are indexed from 1
2: $A_{curr}[1]$ = ExecuteCellQuery($Q_{curr}$)
3: **for** $i = 2, \ldots, d+1$ **do**
4: $Q_{prev} \leftarrow$ GetPreviousNeighbour($i$-1) // decrement the $(i-1)^{th}$ dimension of $Q_{curr}$ by stepsize
5: int[] $A_{prev}$ = GetAllAggregates($Q_{prev}$)
6: $A_{curr}[i] = A_{curr}[i-1] + A_{prev}[i]$
7: StoreAllAggregates($Q_{curr}$, $A_{curr}$)
8: **return** $A_{curr}[d+1]$

# 6. PUTTING IT ALL TOGETHER

Algorithm 4 presents the pseudo code for the ACQUIRE framework. Given an initial query $Q$ and the refinement threshold $\gamma$, ACQUIRE begins to iteratively *Expand* and *Explore* refined queries, starting at the origin of the refined space and sequentially traversing queries in subsequent layers. For each refined query, ACQUIRE calculates the aggregate using the Incremental Aggregate Computation technique described in Algorithm 3. Once the aggregate value $A_{actual}$ has been determined, it is compared to $A_{exp}$. If the aggregate is within the error threshold $\delta$, the query is stored in the answer list ($\mathcal{A}$). 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 constraint by more than $\delta$, ACQUIRE explores the next higher layer. Lastly, if any query overshoots the expected aggregate value by more than $\delta$, we repartition the cell corresponding to the given query and examine queries lying within. We repeat the repartitioning process for $b$ iterations, where $b$ is a tunable parameter. If, at the end of repartitioning, no query is found to satisfy the aggregate constraint, ACQUIRE returns the query attaining the closest aggregate value.

**Algorithm 4** ACQUIRE(Query $Q_{original}$, double $A_{exp}$, int $\delta$, double $\gamma$)

1: $\mathcal{A}$ =[] // Set of refined Queries
2: Queue *queryQueue* = [] // Data structure for traversal
3: $d \leftarrow$ Flexible predicates in $Q_{original}$
4: int[d] $Q_{curr}$ = {0,..., 0} // Origin represents $Q_{original}$
5: *queryQueue*.push($Q_{curr}$)
6: int *minRefLayer* = MAX_INTEGER_VALUE
7: int *currRefLayer* = 0
8: **while** (*currRefLayer* $\leq$ *minRefLayer*) **do**
9: double $A_{actual}$ =ComputeAggregate($Q_{curr}$, $d$) // Algorithm 3
10: **if** ( $\mid A_{exp}$ - $A_{actual} \mid \leq \delta$) **then**
11: $\mathcal{A}$.add($Q_{curr}$)
12: *minRefLayer* = *currRefLayer*
13: **else if** ( $A_{exp} > A_{actual}$) **then**
14: $\mathcal{A}$.add(Repartition($Q_{curr}$))
15: $Q_{curr}$ = GetNextQuery(*queryQueue*) // Algorithm 1
16: *currRefLayer* = QScore($Q_{curr}$)
17: **return** $\mathcal{A}$

# 7. EXTENSIONS

In this section, we present extensions to the framework that accommodates some of the limitations of our approach.

## 7.1 Preferences in Refinement

Along with the NOREFINE keyword used to identify and preserve rigid constraints, ACQUIRE allows users to set preferences

on which predicates should be refined. This can be easily done by specifying a $L_{W_p}$ norm which sets appropriate weights on various predicates. Similarly, users can also supply maximum refinement limits on predicates. While we provide several avenues for user control, user intervention is not required and each tunable parameter is provided an appropriate default setting.

## 7.2 Contracting Queries With Too Many Results

ACQUIRE with minor modifications handles queries that generate *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 $Q'_{min}$. ACQUIRE now traverses the refined space to find queries that meet the cardinality constraint, this time minimizing refinement with respect to $Q$ instead of $Q'_{min}$.

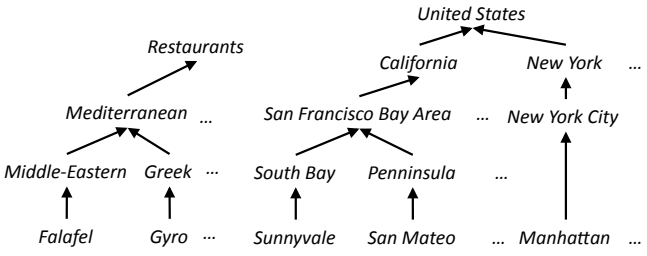



**Figure 7: Ontology for Categorical Data**

## 7.3 Non-numeric Predicates

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. However, ACQUIRE 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 7 depicts sample ontology trees related to *food* preferences and *location*. The refinement distance between the original query desiring 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. Given this meta-information from the ontology tree and a distance metric, the ACQUIRE framework can be used to refine categorical predicates.

## 7.4 Exploiting Indexes and Data Distribution

The algorithms discussed so far make no assumptions about the underlying data distribution or presence of indexes on the data. Moreover, experiments in Section 8 indicate that ACQUIRE is already 2 orders of magnitude faster than the state-of-the-art techniques. However, if required, we can further boost the efficiency of ACQUIRE by employing a specialized bitmap-like index structure on the tables. To construct this index, we divide each attribute dimension into equi-width parts and create a multi-dimensional grid on the table. We then examine the records in the table to determine which grid cell each record belongs to. In our index, each cell is assigned a corresponding bit, which is set to 1 if the cell contains some tuple and 0 otherwise (storing the number of tuples may be easier for keeping the index up-to-date but requires more space). Once constructed, this simple index structure can be used in the $Explore$ phase to determine if a given cell query is empty without actually executing the query. If the query is found to be empty, we can safely skip it and proceed to the next, thus avoiding unnecessary query execution costs.

# 8. EXPERIMENTAL EVALUATION

## 8.1 System Implementation

The ACQUIRE framework is built on top of Postgres. ACQUIRE sits outside the DBMS where it performs the tasks of exploring the refinement space, formulating queries and applying our aggregate computation algorithm. To make ACQUIRE portable across multiple database systems, and to aid in proper comparison with competing techniques, all query execution tasks are delegated to the DBMS. We similarly implemented the compared techniques on top of Postgres.

## 8.2 Alternative Techniques

We compare ACQUIRE to three extensions of existing techniques that address the ACQ problem to varying degrees. First, we compare it to Top-k which, although unable to produce *refined queries*, is suited to ranking tuples in order of refinement. While it is straightforward to translate a COUNT constraint to Top-k, translating other aggregate constraints (e.g. AVERAGE) is difficult if not impossible. As a result, we only study Top-k ranking for COUNT constraints. We use existing DBMS capabilities of ORDER BY and LIMIT to implement Top-k, as demonstrated on generic queries (Q and corresponding Top-k-Q) below.

```
Q = SELECT COUNT(*) from table1
    WHERE x <= 10 and y <= 20;

Top-k-Q = SELECT * FROM table1 ORDER BY
(case when  (x <= 10) then 0
else (x - 10)/(x.max - x.min)) +
(case when (y <= 20) then 0
else (y - 20)/(y.max - y.min)) LIMIT A_exp
```

We also compare ACQUIRE to the TQGen [11] and a simple binary search (BinSearch) technique [11]. Our experiments uses the TQGen parameters reported in [11]. To allow for uniform comparisons across all methods, we do not employ sampling techniques for TQGen. However, our experiments demonstrate that our results hold even for small sample-size datasets (see Figure 10.a). The final point to note is that, unlike ACQUIRE, (a) none of the above techniques addresses aggregates other than COUNT, and (b) even for COUNT, none of the above techniques are capable of refining join predicates.

## 8.3 Methodology

To study the robustness of ACQUIRE we vary (1) *dimensionality* of refinement space, i.e., number of refinable predicates, and combination of attributes in these predicates, (2) *magnitude of aggregate value discrepancy*, i.e., ratio $\mathcal{A}_{actual}/\mathcal{A}_{exp}$ between the actual aggregate value and the desired aggregate value, (3) *dataset size*, (4) *aggregate types*, and (5) *data distributions*. To study the efficiency gained by the ACQUIRE system, we evaluated the net decrease in query execution time for various data sizes and dimensionality. Finally, we evaluated the performance of ACQUIRE under various settings of refinement and aggregate thresholds as well as presence of join refinement. For each experimental setting, we measure the time needed to return the set of refined queries, $\mathbf{Q}_F$, amount of refinement (refinement score), and relative aggregate error = $Err_A$.

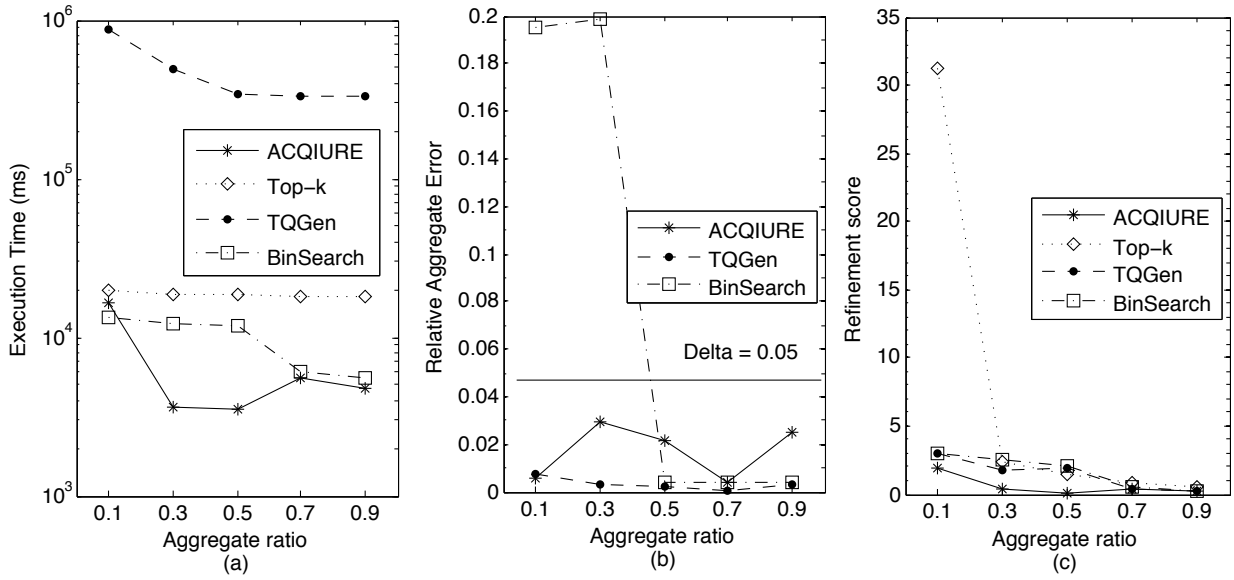



**Figure 8: Performance Comparison Under Varying Aggregate Ratios: ACQUIRE, Top-k, TQGen and BinSearch**

All algorithms were implemented in Java. Measurements were obtained on AMD 2.6GHz Dual Core CPUs, and Java heap of 2GB. We utilized the TPC-H datasets of varying sizes (1K - 10M tuples). Since the standard TPC-H data is uniformly distributed (i.e., $Z = 0$), we used [3] to also generate skewed data with $Z = 1$. Our test queries are TPC-H queries which have been adapted to include only numeric range and join predicates. Query Q2 (in Example 2) provides skeleton query that was used to evaluate the SUM aggregate. For each dataset, query, and ACQUIRE settings, we define the original aggregate $A_{actual}$ and the aggregate ratio $\frac{A_{actual}}{A_{exp}}$.

## 8.4 Performance Comparisons

### *8.4.1 Effect of $\frac{A_{actual}}{A_{exp}}$ Ratio*

We first examine the effect of aggregate ratio, $\frac{A_{actual}}{A_{exp}}$, on the execution time, error rate and refinement scores. A small $\frac{A_{actual}}{A_{exp}}$ ratio implies that the original query is highly selective and needs large refinements, while a large $\frac{A_{actual}}{A_{exp}}$ implies that the original query is close to the desired query and needs only small refinements. These experiments were carried out on a 1 million tuple dataset and a query with 3 flexible predicates. The aggregate ratio was varied between 0.1 - 0.9.

As shown in Figure 8.a, the execution time for ACQUIRE increases with decreasing expansion ratio, i.e., the greater the need to expand the query, longer it takes for ACQUIRE to reach the required aggregate ratio. While Top-k requires the same execution time (the ranking function is unchanged and all records need to be sorted), its execution time however is on average 3.7X more than ACQUIRE. TQGen and BinSearch both need to explore the same number of queries each time and hence their execution time remains constant. ACQUIRE does consistently as well as all the other methods, and is on average 2X faster than BinSearch and 2 orders of magnitude faster than TQGen (Y-axis is in log scale). Although BinSearch shows promise with respect to execution time, we show next that it is not robust with respect to aggregate errors.

Figure 8.b shows the relative error (average relative error for BinSearch) for each of the queries with changing aggregate ratio. We do not compare Top-k because a Top-k query explicitly specifies the number of tuples to return and hence has no aggregate error by definition. The BinSearch line in the graph shows that BinSearch is extremely unstable and has high variance in aggregate errors. The underlying reason is that BinSearch is very sensitive to the order in which predicates are refined; even a single change to the order can change the error by a factor of 100. To illustrate, one ordering of predicate refinement in BinSearch produces a refinement error of 0.19 or 20% whereas another ordering produces an error of 0.002 or 0.2%. Attempting to refine the query by attempting all orderings of predicates is computationally expensive. ACQUIRE, on the other hand, not only produces queries consistently within the threshold ($\delta$ = 0.05), but also does so efficiently. TQGen, in fact, produces lower error rates than ACQUIRE. However, this reduction comes at the cost of a 100X increase in execution time. Since both error rates are acceptable, we prefer ACQUIRE. Lastly, in Figure 8.c we compare the refinement scores obtained by each method. We see that the refinement score for queries generated by other methods are 2-3X larger than those from ACQUIRE.

### *8.4.2 Effects of Dimensionality*

Next, we discuss the effects of increasing dimensionality, i.e. increase in the number of query predicates. We used the same dataset as before, used expansion ratio = 0.3 and varied the number of predicates in the query. In execution time, we see the same trend as before where the execution time increases with increasing dimensionality of the query. However, for ACQUIRE, the increase is largely linear and not exponential. For Top-k, the execution time remains largely constant since only the ranking function changes. For TQGen, we see an exponential increase in the execution time (as number of queries executed is exponential in number of dimensions) with the method taking 500X more time than ACQUIRE for high dimensional queries. Thus, ACQUIRE is a much better alternative to the state-of-the-art on queries of varying dimensions. Figure 9.b once again demonstrates that BinSearch is extremely unstable with respect to aggregate error. While some queries obtain an error rate of 0.6%, some obtain an unacceptable error rate of 45%. This large variance in error values produced by BinSearch indicates that the method is unpredictable and not-robust. As a result, it cannot guarantee any threshold on the error rate.

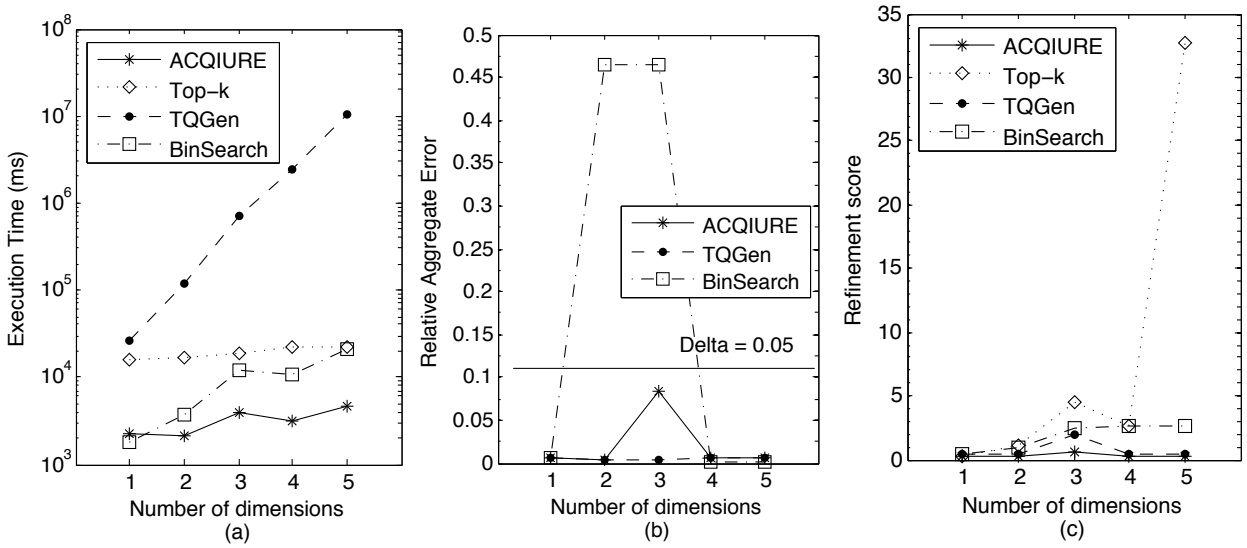



**Figure 9: Performance Comparison Under Varying Number of Predicates Ratios: ACQUIRE vs. Top-k vs. TQGen vs. BinSearch**

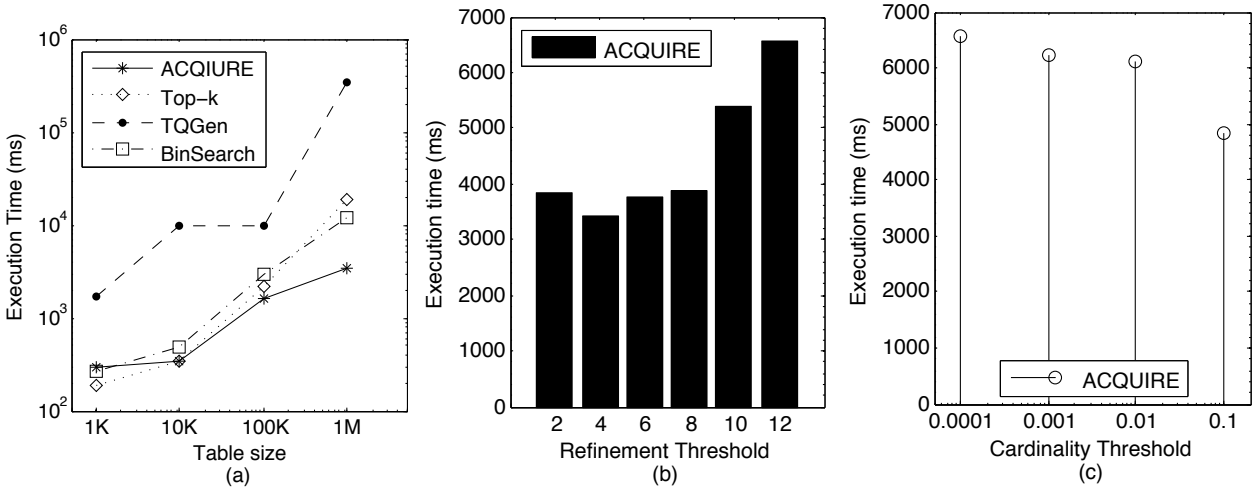



**Figure 10: Performance Comparison Under Varying (a) Table Size, (b) Refinement Threshold and (c) Cardinality Threshold**

In Figure 8.c exemplifies the trends in query refinement score seen with all methods. The refinement scores of ACQUIRE are consistently the lowest across all methods – meaning fewer changes to the original user query and therefore more desirable. Top-k produces higher refinement than ACQUIRE. This figure also shows that TQGen and BinSearch can have high variance in refinement scores. Since the goal of these techniques is only to meet the aggregate constraint and not to minimize refinement, this is expected. BinSearch queries have, on average, 4.8X more refinement than ACQUIRE queries.

### *8.4.3 Varying Table Size*

For datasets of varying size, beginning with a 1k-tuple dataset (to mimic a sample based approach) to a 1M-tuple dataset. As shown in Figure 10.a the execution time for ACQUIRE and all compared techniques increases proportionally to the size of the dataset. Relative error and refinement scores show the same trends as before.

### *8.4.4 Effect of Varying Data Distributions*

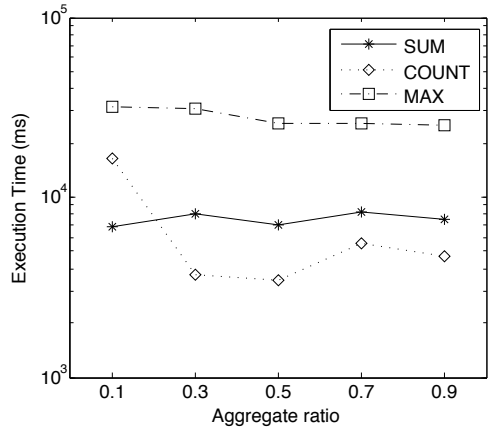
To study the robustness of our method, we re-ran experiments on data with Zipfian skew = 1. Trends in results were same as above.

### *8.4.5 ACQUIRE Parameter Studies*

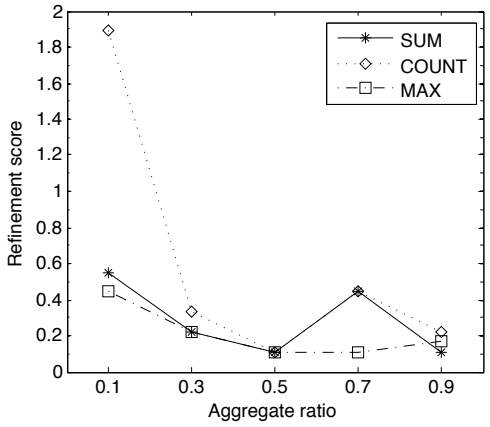
In Figure 10.a and Figure 10.c, we report the performance of ACQUIRE with respect to its internal parameters, namely the aggregate threshold, the number of steps in the grid and the depth of the search. As expected, a stringent cardinality and refinement threshold produces proportional increases in the ACQUIRE execution time as more queries need to be explored.

### *8.4.6 Varying Aggregate Types*

ACQUIRE framework is general and can be applied to different types of aggregates satisfying the optimal substructure from Section 2.6. We tested the technique for other aggregates too. Figure 11 shows the results for the SUM, COUNT and MAX aggregates. We omit MIN since this can be written as the MAX(-1 *




(a)




(b)

**Figure 11: ACQUIRE's Performance on Different Aggregates**

attribute). We find that ACQUIRE successfully minimizes refinement and reaches the aggregate thresholds in all the above aggregates.

## 8.5 Summary of Experimental Conclusions

1. ACQUIRE is consistently 2 orders of magnitude faster than TQGen and on average 2X faster than *BinSearch*.

2. In all experimental conditions, ACQUIRE's aggregate error is well below the aggregate error threshold. In contrast, BinSearch has very high variance in error rates, reaching up to 45% of the expected aggregate value.

3. Although, Top-k can be efficient at small-sized datasets, it quickly becomes inefficient as data size increases. In general Top-k is about 3.7 times slower than ACQUIRE.

4. ACQUIRE generates queries that on average have 2X better *refinement scores* than a query produced by either TQGen or BinSearch.

# 9. RELATED WORK

In this section, we discuss two areas of related work namely, (1) *set-based queries* [6, 5, 16, 14, 13] and (2) solving the *empty result problem* [9, 1, 11]. Existing set-based query evaluation techniques differ from our work fundamentally because they are solving a different problem than the one addressed in this work. For instance, techniques proposed in [13] address the problem of recommending "satellite items" (e.g., car charger, case) for a given item that the customer is currently shopping for (e.g. smart phones). Alternatively, [16, 5] solve the generalized the Knapsack problem [6] of making composite recommendations of a set of items. That is, recommend the Top-k sets of items with the total cost below a given budget and preferring the set with higher ratings. In contrast, [14] focused on finding users (e.g. tourists) sets of results (e.g. a set of places of interest) given a set of constraints (e.g. budget). This is identical to the current behavior of the Facebook Ad Creation Interface [4]. However, this approach is less than desirable (as described in Section 1) as it would force Alice to go through hundreds of iterations to find a meaningful query that meet the aggregate constraints. To summarize, techniques for set-based queries focus on returning tuples or sets of tuples that meet a constraint. In large scale database systems since the users are mostly unfamiliar with the characteristics of the underlying data, they usually construct queries that are either too strict or too broad [9]. In such scenarios, execution techniques designed for set-based queries could potentially return no results or all tuples in the database.

To the best of our knowledge, we are the first work to address the question of recommending refined user queries that meets their aggregate constraints. Existing query refinement techniques can be classified into two categories namely, (1) *tuple-oriented approaches*, and (2) *query-oriented approaches*. Table I summarizes the key related work, and whether they support all aggregate constraints and / or a proximity criteria.

| Techniques | Aggregates Supported | Proximity | Card. | Query |
|---|---|---|---|---|
| Tuple-Oriented: Skyline [8], Top-k [2], | COUNT | ✓ | ✓ | |
| Query-Oriented: BinSearch [11], IQR [10] | COUNT | | | ✓ |
| Query-Oriented: TQGen [11], Hill-Climbing [1] | COUNT | | ✓ | ✓ |
| **ACQUIRE** | COUNT, SUM, MIN, MAX, AVG, UDA[2] | ✓ | ✓ | ✓ |

**Table 1: Summary of the Related Work**

**Tuple-Oriented Techniques.** Result refinement techniques [12, 8] focus only on generating the required number of results and ignore the problem of generating refined queries that explain how the result tuples were selected. The refinement criteria are crucial in scientific and business applications. Similarly Top-k algorithms, such as [2], while useful in many instances cannot correctly address the ACQ problem since they can only handle COUNT aggregates. To illustrate, consider a query that selects patients based on income, blood pressure, and the amount of weekly exercise. A Top-k based approach will obtain the required number of patients, but these patients will likely be skewed in certain predicate dimensions and will not be representative of the population. Thus pure Top-k and its variations are inadequate to address the ACQ problem; clearly demonstrated in our experiments (see Section 8).

**Query-Oriented Approach.** More recently in the context of database testing [1, 11, 10] have started to focus on the problem of

[2]User Defined Aggregates that either satisfy the optimal substructure property (OSP) or can be broken into functions that satisfy OSP

generating refined predicates. [10] proposed a framework 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 ACQUIRE does. For select-only queries, [11] seeks only to attain the desired cardinality and disregards proximity. Consequently, it cannot guarantee that the refined query has the least refinement. The BinSearch algorithm [11] is heavily influenced by the order in which predicates are refined; some orders produce accurate results whereas others produce large errors. Unlike ACQUIRE, these techniques don't generate a set of alternative refined queries for the user to choose from. To summarize, ACQUIRE is the first technique to refine select and join queries to meet the dual constraints of proximity to the original query and the desired aggregate constraint.

# 10. CONCLUSION

We introduce *Aggregation Constrained Queries* that constrain not only the tuples produced by the query, but also aggregates on these tuples. We argue that algorithms targeting ACQs must combine efficient query execution and query refinement. We propose ACQUIRE to tackle ACQs. ACQUIRE adopts the *Expand and Explore* strategy where it iteratively *expands* the original query to minimize refinement and efficiently *explores* refined queries via a novel incremental aggregate computation technique. The general principle of ACQUIRE allows us to support user defined predicate refinement scoring and aggregate error functions. ACQUIRE guarantees that each query is executed at most once, regardless of the number of queries it is contained within thereby exploiting work sharing. This enables ACQUIRE to consistently perform up to 2 orders of magnitude faster and produce queries with 2X smaller refinement than extensions to existing techniques.

# 11. REFERENCES


[1] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE TKDE*, 18(12):1721–1725, 2006.

[2] S. Chaudhari and L. Gravano. Evaluating top-k selection queries. In *VLDB*, pages 397–410, 1999.

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

[4] V. Goel. How facebook sold you krill oil. *The New York Times*, August 2014.

[5] S. Guha, D. Gunopulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003.

[6] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.

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

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

[9] G. Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, 2006.

[10] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

[11] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510, 2008.

[12] I. Muslea. Online query relaxation. In *SIGKDD*, pages 246–255, 2004.

[13] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.

[14] Q. T. Tran, C.-Y. Chan, and G. Wang. Evaluation of set-based queries with aggregation constraints. In *CIKM*, pages 1495–1504, 2011.

[15] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[16] M. Xie, L. V. S. Lakshmanan, and P. T. Wood. Breaking out of the box of recommendations: from items to packages. In *RecSys*, pages 151–158, 2010.