# Efficient Skyline Computation over Ad-hoc Aggregations

Shyam Antony, Ping Wu, Divyakant Agrawal, Amr El Abbadi

University of California at Santa Barbara
{shyam, pingwu, agrawal, amr}@cs.ucsb.edu

## Abstract

Aggregation is among the core functionalities of OLAP systems. Frequently, such queries are issued in decision support systems to identify interesting groups of data. When more than one aggregation function is involved and the notion of interest is not clearly defined, skyline queries provide a robust mechanism to capture the potentially interesting points where (i) users do not need to specify a ranking function and (ii) the result is independent of the dimension scales. For providing better exploration functionalities in the OLAP system, in this paper, we propose to use skyline queries over aggregated data to identify the *most interesting groups*. Since the aggregation function has to be ad-hoc to cover a wide variety of user interests, the skyline over the aggregates has to be computed on the fly. Hence any algorithm to compute such a skyline must be fast and be able to progressively produce the result set with potential skyline groups being produced as early as possible. We explore a family of algorithms which try to consume only as many data records as are necessary to compute the skyline and design an optimal algorithm. We further refine the algorithm by taking into account systems issues such as disk behavior which are often ignored but have strong impact on real system performance. Experimental results validate the performance and progressive benefits of our algorithm.

## 1 INTRODUCTION

Multi-Objective optimization (**MOO**) has been an important subject of decision making for many decades. It deals with problems that seek to simultaneously optimize multiple objectives. For such problems, often there does not exist one single optimal solution in the traditional sense. Instead, a number of alternative solutions may coexist involving various trade-offs in different objectives. A pareto optimal set yields the feasible solutions that are not worse than any other alternatives with respect to all objectives and hence defines an "efficient frontier" over the answer space.

In recent years, the notion of multi-objective optimization over large data sets has been a subject of study in the database community. In particular, the skyline query operator has attracted considerable amount of attention after its first introduction in 2001 [4]. Such interest is mainly driven by the fundamental need for MOO in decision making and data analysis scenarios. A classical example is the hotel selection decision in which one would like to optimize both the price and the distance to the beach. In fact, skyline queries can be seen as an instance of the MOO where each tuple in the database is considered a feasible solution in a discrete solution space and attributes of interest are considered as the optimization goals.

Unfortunately, existing research on skyline queries have been mainly restricted to the problem of selecting "interesting"‘ objects/tuples from OLTP (On-line Transaction Processing) databases. However, current enterprise decision making systems primarily benefit more from OLAP (Online Analytical Processing) than OLTP. In the past 15 years, OLAP has grown into a full-fledged market and become the core technology in the broader category of business intelligence. OLAP exposes multi-dimensional views over historical data with different levels of detail and equips analysts with exploration operations such as drill-down and roll-up to effectively explore the often complex data spaces, enabling quick response time for ad-hoc analytical queries over large data warehouses.

We believe that in order to further realize the potential of both OLAP systems and the skyline operations, the skyline operation must be applied to the OLAP context where it rightly belongs. New functionalities have been constantly pushed into OLAP engines to automate data exploration and accelerate the process of knowledge discovery, eg., [25], [26]. Due to the often large and complex data spaces in OLAP, analysts need the system to prioritize the cube cells and highlight interesting subspaces. This functionality calls for essentially two components. One component is that users should be able to express their customized objective functions effectively or in other words the system should support *user defined ad-hoc queries*. The other component is to avoid information overload by separating the most interesting sub-space from the less interesting ones. Top-$k$ query provides global order over the result set. However, in many scenarios, users often do not know how to quantify the trade-offs between different goals and just wish to quickly grasp what can be potentially interesting. Furthermore, the different objective functions can

be conflicting in nature, therefore a global order can hardly produce sensible results for everyone. This essentially calls for a new functionality which skyline operations can readily offer. First, skyline query does not require any specific ranking functions from the user and nevertheless its result set contains records that maximize some ranking functions. Second, skyline query is defined over the orders within the attribute domains and thus is robust to the scale mismatch among different domains.

Another benefit of supporting the skyline operation into OLAP systems is that skyline results usually convey useful *trade-off information* in the aggregation space which can lead to executable business operations. For example, suppose that a vehicle manufacturer wants to make decisions about adjusting sales volume and profit across all the dealers in California. A useful guiding factor would be the skyline which is computed by aggregating the profit per-car-sold and car sales volume for each dealer. The reason is that dealers with high volume but less profit have the potential to increase the profit per car, while dealers with low volume but high profit-per-car have the potential to raise the sales volume. By adjusting the business strategy accordingly, manufactures can move further towards their "efficient frontier".

In order to introduce the skyline operation into existing OLAP systems, the main challenge is to efficiently compute *skyline over aggregation*, which although useful, has not been studied in previous literature. To illustrate this problem, we give an example.

**Example 1**. For the car dealer application scenario above, we can express it in a SQL-like syntax as follows:

**SELECT DEALER, AVG(SALE_PRICE-COST_VALUE) AS** $OBJ_1$**,**
**SUM(QUANTITY) AS** $OBJ_2$
**FROM CAR_SALES_TABLE**
**GROUP BY DEALER**
**SKYLINE ON** $OBJ_1$ **MAX,** $OBJ_2$ **MAX**

As we can see, this problem exposes two unique challenges for query processing. First, unlike traditional skyline problem where the attribute values from individual tuples are considered, the input now becomes the aggregated values from group of tuples defined by the group-by predicate and the *objective functions*. Second, to make the problem even more challenging, these objective functions are defined by the user in an ad-hoc manner. Such a dynamic nature essentially makes existing index-based query processing methods such as BBS [22] no longer applicable (Another reason is that not a single OLAP system provides R-Tree support). On the other hand, methods that computes skyline groups after aggregation is finished will effectively consume all the tuples in the fact table and therefore can be prohibitive in large data warehouses. In this paper, we address these challenges by pushing the skyline computation inside the aggregation operation and thereby limit the I/O costs to provide the final answers. Our methods have provable properties as well as taking into consideration practical issues.

To further motivate the problem, we give another two application scenarios as follows. Consider an executive who explores the potential future locations to open new branches: she may wish to find the locations where the expected profits are maximized and the expected operational costs are minimized. Both values involve aggregations over the collected data of existing shops at each candidate location. As a final example, in a customer relationship warehouse, the analysts want to make series of decisions about marketing strategies for the coming holiday season. With a fixed marketing budget, she wishes to see a quick summary on some conflicting factors at different levels of locations (such as state, county, zip code and so on). Skyline operations provide a good candidate for such data summarization since it often captures such conflicting or unrelated concerns over historical data such as the total volume of sales and the customer complaint behavior.

As a first step towards multi-objective data exploration in OLAP, this paper makes the following key contributions. First, this is the first paper on supporting skyline aggregations within OLAP systems. Second, we identify the major technical challenges, formulate the problem framework, study properties and develop algorithms that allows us to effectively "push" the skyline operator into the aggregation phase thereby improving both performance and progressiveness. Finally, we conduct a thorough experimental study to validate and further explore the proposed solutions.

## 2 PRELIMINARIES

### 2.1 Query Model

We first normalize our query model. Consider an OLAP fact table $FT$ with $p$ dimensional attributes $D_1, D_2, ..., D_p$ and $k$ measure attributes $M_1, M_2, ..., M_k$. A skyline aggregation query consists of two basic components: group-by predicates and the objective functions. Each objective function $OBJ_i$ maps a group of tuples to a scalar value and can be defined by users in an ad-hoc manner. We shall discuss more about objective functions later. Such query can be formulated as follows:

**SELECT** $D_1$**,...,**$D_p$**,**$OBJ_1, ..., OBJ_d$
**FROM FT**
**GROUP BY** $D1$**,...,**$D_p$
**SKYLINE ON** $OBJ_1, OBJ_2$**,...,**$OBJ_d$

where $OBJ_1, ..., OBJ_d$ are $d$ objective functions defined over the measure attributes $\{M_1, M_2, ..., M_k\}$. Without loss of generality, we assume that users always prefer smaller values on each objective function. Conceptually, the $d$ objective functions map each group to a $d$-dimensional space, which we denote as *objective space*. And the query results are a set of groups that are not dominated by any other groups in this objective space.

### 2.2 Feature Expressions, Objective Functions and Skyline Groups

Intuitively an *objective function* $OBJ$ represents one particular goal of the user. $OBJ(G)$ over a group of tuples $G$ has two components $F$ and $A$, where $F$ is a *feature expression* and $A$ is an aggregation function such as SUM. A feature expression $F$ is defined over each tuple $t$ in $G$ by mapping the measure attributes of $T$ to an intermediate scalar value. For example, on a sales table, a feature expression can be the number of items sold multiplied by the sale price for each sales transaction. Essentially $OBJ$ takes a group of tuples as input and applies the feature expression $F$ to each of the tuples to emit a set of intermediate values which are then aggregated by applying aggregation function $A$ to produce a final scalar value.

Given a fact table $FT$ with measures $M_1, M_2, M_k$ and dimensions $D_1, D_2..., D_p$. A skyline aggregation query first partitions the tuples in $FT$ into different groups based on the *grouping attributes*, and for each the resulting group, $d$ values are computed based on the $d$ objective functions $\{OBJ_1, OBJ_2,..., OBJ_d\}$ defined in the **SKYLINE ON** clause. A group $g_1$ is said to be dominated by another group $g_2$ if for every objective $F_i$, $F_i(g_2) \leq F_i(g_1)$ and for at least one objective $F_j$ we have $F_j(g_2) < F_j(g_1)$. The skyline groups are those groups that are not dominated by any other group. We assume minimum to be "better" for domination but adapting the solutions for maximum involves only minor changes.

***//Ping will redraw the figure***

**Example 2:** Table 1 shows a base table having three dimensions $\{D_1, D_2, D_3\}$ and two measures $\{M_1, M_2\}$. Query $Q_1$ groups by two dimensions $D_1$ and $D_2$ and for each group computes respectively the AVERAGE and SUM for measures $M_1$ and $M_2$. Therefore the objective functions are $OBJ_1 = A_1(F_1)$ and $OBJ_2 = A_2(F_2))$ where $F_1$ and $F_2$ can be seen as linear functions: $1 \times M_1 + 0 \times M_2$ and $0 \times M_1 + 1 \times M_2$ respectively in this simple example and $A_1$ and $A_2$ are SUM and AVERAGE respectively. Table 2 shows the aggregate values of the different combinations of $D_1$ and $D_2$. The goal is to minimize $OBJ_1$ and $OBJ_2$. The skyline groups $\langle c, e\rangle$ and $\langle b, f\rangle$ are highlighted in bold. □

### 2.3 Assumptions

As defined above a query consists of a grouping component and an objective component. We assume that the grouping component is pre-computed. Pre-computing the grouping component is not difficult as the grouping component is just a sub-set of the dimensional attributes and therefore not ad-hoc. However space limitations may prevent us from pre-materializing all possible dimensional attribute combinations. Since our focus is on data exploration sessions, if a query is posed that involves a Group-By that has not been materialized, it can be computed once and used at least for the rest of the session.

Next we specify the class of objective functions we support. An objective is determined by two components, an aggregate function $A$ and a feature expression $F$. Among these two components, feature expressions tend to be user defined while aggregate functions tend to be pre-defined system functions. The reasons behind this claim include the difficulty in constructing a large number of useful aggregation functions by composing a small set of pre-defined aggregation functions and also in the effort necessary to integrate user defined aggregate functions into extensible database systems. Feature expressions, on the other hand, allow easy composition of pre-defined expressions and hence can be easily integrated into user interfaces that spare the user from having to directly interact with the execution engine of database systems. Hence we should attempt to support a wide class of feature expressions and commonly used aggregate functions.

We allow arbitrary monotonic feature expressions. Monotone feature expressions are those whose value does not decrease when the measure values increase. As far as aggregate functions are concerned, the solutions we develop are correct for monotone aggregate functions and efficient for self-maintainable aggregate functions and functions which can be expressed in terms of self-maintainable aggregate functions. Let $A$ be an aggregate function that on the input tuple group $G = \{t_1, t_2, ..., t_n\}$ outputs $A(G)$ as the aggregate. Let $G'$ be the set obtained by replacing one element in $G$ say $t_i$ with $t'_i$. Then $A$ is monotone if $A(\{t'_i\}) \leq A(\{t_i\})$ implies $A(G') \leq A(G)$. $A$ is self-maintainable if we can calculate $A(G')$ from only $A(G), A(\{t_i\})$ and $A(\{t'_i\})$. More rigorously there exists a known function $U$ such that $A(G') = U(A(G), A(\{t_i\}), A(\{t'_i\}))$. SUM is an example aggregate function that is monotone and self-maintainable. SUM is self-maintainable because we can calculate the new summation by adding the difference between $a'_i$ and $a_i$ to the old result. Average is example of an aggregate function that can be expressed in terms of self-maintainable aggregate functions (COUNT and SUM). Median and many other holistic functions [11] are not self-maintainable. Holistic functions are usually difficult to handle efficiently in OLAP settings [1] and are left as future work.

Figure 1: Example

Table 1: Base

| d1 | d2 | d3 | m1 | m2 |
|---|---|---|---|---|
| a | e | x | 16 | 8 |
| a | e | y | 6 | 3 |
| a | e | y | 2 | 5 |
| a | e | z | 8 | 4 |
| a | f | x | 6 | 2 |
| a | f | z | 9 | 2 |
| a | f | y | 3 | 2 |
| b | e | x | 6 | 0 |
| b | e | x | 6 | 3 |
| b | f | z | 5 | 1 |
| b | f | z | 5 | 2 |
| c | e | y | 3 | 5 |
| c | f | z | 4 | 6 |

Table 2: Aggregate

| d1 \ d2 | e | f |
|---|---|---|
| a | 8, 20 | 6, 6 |
| b | 6, 3 | **5, 3** |
| c | **3, 5** | 4, 6 |

**Query Q1**

```
Select d1, d2, AVG(m1) as o1,
       SUM(m2) as o2,
From Base_Table
Group By d1, d2
Skyline on o1 MIN, o2 MIN.
```

### 2.4 Problem Analysis

Aggregation queries if computed on demand result in bad response times since most aggregation queries have to make at least one pass over large data sets. This well known problem is the major technical challenge in attempting to incorporate skyline operator into OLAP systems. To be truly effective, we should allow the user to specify ad-hoc objectives rather than a pre-determined set of objectives. Two approaches have been suggested in the past for this problem. The first is to essentially materialize the answers to a pre-determined set of queries either by building appropriate views [7] or even by materializing the entire (or portion of the) data cube [11]. This approach is not applicable when dealing with ad-hoc queries. The second approach is to keep the user engaged during the course of the execution of a long aggregation query by providing running aggregates with confidence bounds [13]. But when dealing with skyline queries the aggregation is not the end-product but rather an intermediate phase in query processing and hence presenting the aggregates is of little value. However this approach is useful if we can adapt it to show potential skyline points as the query is being executed. In this paper we follow a third approach which is to push the skyline operator into the aggregation phase. The basic idea is to prune away partially aggregated groups as soon as it becomes clear that they cannot make it into the final skyline, and therefore reduces significant amount of time which may otherwise be spent on a full aggregation. This allows us to take advantage of the selectivity of the skyline operator to avoid a full aggregation and also output skyline points as early as possible.

## 3 SOLUTION FRAMEWORK

The baseline solution first aggregates all groups. Since the grouping part is pre-computed, the aggregation phase can be accomplished with a single pass over the data. The aggregated groups are then input into a skyline computation algorithm such as LESS [10] which outputs the skyline groups. The prime target for optimization is the aggregation phase since its input is much larger than the input to the skyline computation phase. How can we optimize the aggregation phase when it makes only a single pass over the data? Since the aggregation phase reads every tuple, if we can design an algorithm which reads only as many tuples as are necessary to compute the skyline, we can potentially come up with a solution faster and more progressive than the baseline solution.

### 3.1 Bounds in Multi-dimensional Objective Space

To motivate the intuition behind the proposed solutions we start with a contrived and simple example. Consider a sorted list of positive numbers (1,2,3,4). The length of the list is known. Suppose we want to determine if the summation of the numbers in the list exceeds a threshold, say 6. We can read through the list until the partial aggregate exceeds the threshold. This is analogous to the full aggregation mentioned above. But we have auxiliary information about the list. Because we know that the list has only positive numbers, even before reading one tuple we know that the summation is at least zero. When the first tuple is read, because the list is sorted and the length is known we know that the summation is at least 1+3*1=4. Similarly when we read the second tuple, 2, we know that the summation is at least 1+2+2 *2=7 and hence we can conclude that the sum will exceed the given threshold 6. Thus we have avoided scanning through the entire list by essentially coming up with a lower bound on the value that the aggregation function can eventually attain. We would like to use this idea of using lower bounds of unread tuples to lower bound aggregate functions in computing the skyline groups. The idea of bounding unread tuples has been used in a variety of contexts for top-k problems [17, 5, 8] and also for computing the skyline in vertically partitioned data sets [2].

**Bound Definition:** The example above deals with a list of numbers. But our goal is to compute the skyline groups in a $d$-dimensional objective space. So we start by defining the multi-dimensional equivalent of a lower bound. Given a set of $d$ objective functions $OBJ = \{OBJ_i, 1 \leq i \leq d\}$ which define an objective space for each group $g$. We define a bound of $g$ to be an estimate $\overline{OBJ}(g)$ of the position of $g$ in the objective space such that if $OBJ(g)$ is the actual position of $g$ in the objective space, then for every objective $OBJ_i$, $\overline{OBJ}_i(g) \leq OBJ_i(g)$.

Bounds of a $d$-dimensional point are essentially points which are either identical to the point or located in the region that dominates the point. For example, in a two dimensional objective space if $\langle 3, 3\rangle$ is the actual position of a group $g$, then $\langle 1, 2\rangle$, $\langle 2, 3\rangle$, $\langle 3, 3\rangle$, etc are possible bounds for $g$. Note that by this definition if some group $g_1$ dominates the bound of another group $g_2$ then $g_1$ also dominates $g_2$ in the objective space. Also note that if $g_1$ is not dominated by the bound of another group $g_2$ then $g_1$ cannot be dominated by $g_2$. Let $b_1$ and $b_2$ be two bounds of a group $g$. Then we say $b_1$ is tighter than $b_2$ if $b_2$ dominates $b_1$. The tightest bound is that which is identical to the actual position of the group in the objective space.

### 3.2 Bounding Strategy

How do we determine a bound for a group before any tuples of that group is read? How do we tighten that bound whenever a tuple (or set of tuples) is read? And finally, in what order should we read the different tuples of a group? We call an answer to this three part question as a *bounding strategy*.

In the above one dimensional example, the bounding strategy is very simple. The initial bound of 0 is derived by using the fact that all the tuples are positive integers. The different tuples are read in increasing order of their one dimensional value. And, the bound is tightened every time a tuple is read by essentially using the newly read value as the lower bound for every unread tuple. We develop general $d$-dimensional bounding strategies in the next section. For now, we take a bounding strategy as a black box which

takes the current bound of a group as input, reads one or more tuples from that group and provides a tighter bound for that group. Each time a group is passed to a bounding strategy it consumes some tuples of the group. This gives rise to a family of algorithms which differ primarily in the order in which they pass different groups to the bounding strategy, i.e. *group ranking strategies*.

### 3.3 Lower Bounding Skyline Groups Algorithms

We consider the family of all algorithms that use the same bounding strategy and use only the information provided by the bounds in determining the skyline groups. We term this family of algorithms as the *lower bounding skyline groups algorithms (LBSGA)*. In restricting the algorithms to use only the information provided by bounds we do limit the optimizing power of the algorithms. We could add additional power to the algorithms, for example, by developing an equivalent notion of upper bound in addition to the lower bound notion developed above. However even under this restriction, as we show later, we were able to arrive at efficient algorithms. Next we state some properties of algorithms in the LBSGA family and identify one which is optimal in terms of the total number of tuples read. These properties may be viewed as an adaptation to the skyline context of some of the properties developed for the top-$k$ problem in [17].

The first property defines a lower bound on the number of tuples from a given group that any algorithm in the LBSGA family must read. Intuitively, we must read at least as many tuples as are necessary for the bounding strategy to tighten the bound enough for the bound to be dominated by the actual position of some other group in the objective space. As already mentioned, if a group $g'$ dominates a bound of $g$, then $g'$ also dominates the actual position of $g$ in the objective space.

**Property 1.** *Let $S$ denote the set of skyline groups and let $g$ be a given non skyline group. Then the minimum number of tuples to be read from $g$ before $g$ is discarded by any algorithm in the LBSGA family is the minimum number of tuples needed by the bounding strategy followed to produce a bound of $g$ that is dominated by at least one group in $S$.*

*Proof.* For any group in the final skyline all tuples have to be retrieved since we need the actual scores in the feature space for such groups. Let g be any group not in the skyline. Suppose an algorithm prunes away g before retrieving $H_g^{min}$ tuples where $H_g^{min}$ is as defined above. Then suppose the unseen tuples of g are such that the objectives of g matches its bound. Then no group can dominate g and hence by definition g should have been declared as part of the skyline thereby proving that the algorithm is incorrect. □

**Corollary 1.** *In order to declare a group $g$ as a skyline group, any algorithm in the LBSGA family has to read from every other group $g'$ at least as many tuples as are necessary such that the bound the algorithm can determine for $g'$ using those tuples is guaranteed not to dominate $g$.*

Intuitively, the fundamental problem is that from the definition of a bound it is possible that a bound could be identical to the actual position of the group it bounds in the objective space. Since we do not allow algorithms the power of "look-ahead", any correct algorithm has to treat a bound as potentially being the same as the actual position. The corollary above provides an interesting insight. There is a minimum amount of "work" that any algorithm in the LBSGA family must perform before it can output the first skyline group. Since this minimum work itself could be pretty large for large datasets, we should consider algorithms which have the ability to at least identify potential skyline groups early and hence can be first presented to the user.

At some intermediate point of an algorithm, suppose $S_c$ is the skyline based on the bounds of all groups, then we call $S_c$ the current skyline and the skyline based on the actual objective values of the groups as the final skyline set $S$. The next property says that any algorithm in LBSGA has to necessarily read more tuples from members of $S_c$ if they have unread tuples.

**Property 2.** *Let $g_c$ be any group with unread tuples in the current skyline set $S_c$. Then no correct algorithm in LBSGA can terminate without reading more tuples from $g_c$. Furthermore, if for every group in $S_c$ all tuples have been read, then $S_c = S$.*

*Proof.* For any group $g_s \in S_c$ further processing is necessary. Otherwise $g_s$ will continue to belong to the current skyline thereby preventing the actual skyline with exact features being materialized. Given any such $g_s \in S_c$ and $g_{ns} \notin S_c$, whatever score $g_{ns}$ can achieve in the feature space $g_s$ can possibly do better. Therefore we have to prioritize $g_s$ over $g_{ns}$. When all the groups in S are fully evaluated we can declare S to be the final skyline because for every group $g_{ns}$ not in S there is atleast one point in S which can dominate the bound and hence by transitivity of dominance the actual position of $g_{ns}$ in the feature space. □

Since $g_c$ is a member of the current skyline, it is not dominated by the bounds of any other group and hence is also not dominated by the actual position of other groups in the objective space. Thus no algorithm in LBSGA can discard $g_c$ without reading more tuples. On the other hand an algorithm cannot declare $g_c$ to be a member of the skyline either, since any algorithm in LBSGA has to read all tuples from a group before declaring it as a skyline group. Since it is necessary to read more tuples from every member of $S_c$ that has unread tuples, any algorithm which at every point reads more tuples only from some member of $S_c$ has to be optimal.

### 3.4 The MOOLAP Algorithm Framework

The analysis so far immediately suggests a simple algorithm. The algorithm works as follows. Compute the current skyline based on the initial bounds. Then for every

group in the current skyline tighten the bound using the bounding strategy. Recompute the current skyline. Repeat this process until for every group in the current skyline we have read all tuples. The disadvantage of this algorithm is obvious. If $c_{max}$ denotes the number of times the bounding strategy has to be invoked before it consumes all the tuples of some group in the final skyline, then the algorithm will have to perform at least $c_{max}$ skyline computations. Thus while this simple algorithm is optimal in terms of the number of tuples read, its worst case time complexity (which is $O(c_{max} * |groups|^2)$) is not good. Therefore the challenge is to come up with an algorithm which avoids the repeated recomputation of the skyline and still finds a way to draw tuples only from the groups which belong to the current skyline.

To achieve this we essentially use a best-first strategy which has been used in a variety of contexts, e.g., [21], [14]. We exploit the fact that the bound with the minimum score with respect to some monotone function, say the $L_1$ norm, always belongs to the current skyline. We term the monotone function the group ranking function. We arrange the bounds in an external memory priority queue on the basis of their group ranking. The final skyline set $S$ is initially empty. The algorithm (called the MOOLAP algorithm) is given in `Algorithm 1`. At each step we retrieve the bound with minimum group ranking function score from the priority queue (`Line 14`) and check if it is dominated by any group in the final skyline set $S$. If it is dominated it is immediately discarded (`Line 14-18`). Otherwise we use the bounding strategy to read more tuples and tighten its bound (`Line 23-24`). The bound is then again checked for dominance and if it is dominated by some group in $S$, it is discarded(`Line 25-29`). If the bound is fully materialized, i.e., it does not have another unread tuple it is added to the final skyline set $S$ (`Line 19-22`). Otherwise it is reinserted into the priority queue $Q$ on the basis of its adjusted group ranking function score(`Line 30`). Finally, we would like to reiterate here that the MOOLAP framework so far treats the Bounding Strategy as a black box, which will be discussed in detail in next section.

**Lemma 1.** *Every group in the priority queue $Q$ in the MOOLAP algorithm will make it to the head of the priority queue in finite time.*

**Theorem 1.** *If the algorithm is run to completion, then every skyline group is fully materialized. Furthermore, the skyline groups are materialized in non-decreasing order of their group ranking function score.*

**Theorem 2.** *The MOOLAP algorithm is an optimal algorithm in the LBSGA family where the cost is the number of tuples read.*

*Proof.* Let $A$ be an algorithm which for some group $g$ draws t tuples before discarding the group which is less than t' the number of tuples drawn by moolap algorithm. Since both algorithms use the same bouding strategy the first t tuples drawn by both algorithms from $g$ are the same. From principle 1 it follows that at the instant $A$ decided to discard g it should have guaranteed that there exists another group $g'$ which dominates the bound of $g$ at that depth. Furthermore, since the algorithms can use only the best bound it follows that for $g'$ every tuple has been drawn by $A$. The L1-norm of the bound of $g$ at the depth t is greater than the L1-norm of the actual score of $g'$. Hence when $g$ makes it to the head of the priority queue at depth t $g'$ would have been fully materialized and hence is either present in the skyline set $S$ or there exists some other group $g''$ in S which dominates the actual feature space position of $g'$ and hence the bound of g. Therefore moolap algorithm will also discard $g$ at depth t contradicting our assumption. Hence the moolap algorithm is optimal among algorithms satisfying the criteria laid down in the above theorem when the cost is determined by the number of tuples drawn per group. □

**Algorithm 1** MOOLAP Algorithm

```
1: BS: the underlying bounding strategies;
2: G: candidate groups;
3: Q: priority queue that stores bounds and corresponding group references;
4: S: the set of skyline groups;
5: for each group g ∈ G do
6:    BS.getInitialBound(g);
7:    g.unseeTuples = g.numTuples
8:    Q.insert(L1Norm(g.bound), g)
9: end for
10: S = ∅;
11: LOOP:
12: while Q! = ∅ do
13:    g = Q.getMin();
14:    for each s ∈ S do
15:       if s dominates g.bound then
16:          discard g and goto LOOP
17:       end if
18:    end for
19:    if g.unreadTuples == 0 then
20:       S = S ∪ {g}
21:       goto LOOP
22:    end if
23:    BS.tightenBound(g)
24:    g.unreadTuples -= bs.numTuplesUsedForTightening
25:    for each s ∈ S do
26:       if s dominates g.bound then
27:          discard g and goto LOOP
28:       end if
29:    end for
30:    Q.insert(L1Norm(g.bound), g);
31: end while
```

The next property specifies the interplay between the bounding strategy followed and the final skyline set $S$. For any non-skyline group $g$, the bounding strategy and $S$ together determine the number of tuples to be retrieved from $g$. For the same group $g$ and fixed $S$ different bounding strategies will consume different number of tuples. Similarly for the same group $g$ and fixed bounding strategy, different $S$s result in the bounding strategy consuming different number of tuples.

**Property 3.** *Consider the MOOLAP algorithm developed above. The number of tuples, $N$, consumed from any non-skyline group, $g$, depends only on the bounding strategy followed and the final skyline set $S$. In particular, the number of tuples read from $g$ is equal to the number of tuples necessary for the bounding strategy to tighten the bound enough to be dominated by at least one member of $S$. $N$*

*is independent of the number of tuples that have to be read for any other non-skyline group $g'$.*

*Proof.* $H_g^\alpha$ cannot be more than such a depth d because the L1 norm score of g's bound at that depth is more than that of the skyline point in S which dominates it and hence when g makes it to the head of the priority queue it would fail the dominance check. $H_g^\alpha$ cannot be less than such a depth because then when g makes it to the head of the priority queue the algorithm would dictate that another tuple be drawn from g. □

## 4 Bounding Strategies

The design space for potential bounding strategies is pretty large. In this section we explore part of this space. In addition to arriving at an efficient bounding strategy our goal is to identify the different factors that should be considered in designing a bounding strategy. As already mentioned, prior research in top-$k$ query processing over aggregates also used the idea of bounding unseen tuples. We start our attempt to develop an effective bounding strategy on the basis of the bounding strategy developed for top-$k$ queries in [17] and use the inadequacies of the strategy as the motivation for developing alternate strategies. In order to leverage the ideas developed in top-$k$ let us first consider the skyline groups problem with only one objective and then the skyline query becomes essentially the top-1 query.

### 4.1 One Objective Bounding Strategy

Let $OBJ = A(F)$ be the one objective function w.r.t. which the skyline groups are to be calculated, where $A$ stands for the aggregation function and $F$ denotes the feature expression. Let $\theta$ denote the top-1 (minimum) $OBJ$ value among all groups. Then while executing the MOOLAP algorithm, for each non-skyline group $g$ we have to draw exactly as many tuples as necessary for the condition $\overline{OBJ(g)} > \theta$ to be satisfied. The basic assumption of the one objective bounding strategy (OOBS) is that it is possible to organize the tuples of a group in such a way that at any step it is possible to draw the tuple with the minimum $F$ (feature expression) value. OOBS also assumes that the number of tuples associated with each group is known. At each step OOBS draws the tuple with minimum $F$ value, i.e., it reads the tuples in $F$-ascending order and uses the $F$-value of the last read tuple as a lower bound on the $F$ value of the unread tuples. The number of unread tuples is known since the total number of tuples in a group is known. This effectively bounds the objective value $OBJ$ since the aggregation function $A$ is monotone. Thus at each step the bound improves because the estimate of the $F$-value of one of the unread tuples is replaced with its actual value and also the estimate of the $F$-value of the other unread tuples improves.

Even for the one objective case there is a fundamental problem with OOBS. It assumes that it is possible to retrieve the tuples of a group in $F$-ascending order and also assumes that we know the count of each group. Assuming that count is known is a reasonable assumption since it can be pre-computed. However to retrieve efficiently in $F$-ascending order there has to be an index on the tuples on the basis of their $F$-value. This is clearly not possible for ad-hoc feature expressions in which case we have to fall back on sorting the tuples on the basis of their $F$-values. Our original goal is to avoid a full scan of all the tuples and this goal is obviously not achievable if we sort the tuples of each group.

Another fundamental problem with OOBS arises when we try to generalize OOBS to the general case where the skyline is computed over multiple objective functions. Can we order the tuples in some manner such that for all monotonic feature expressions the order returns a close approximation of the respective $F$-ascending order? The answer is unfortunately no. The counter-argument is as follows. If one tuple dominates (in the measure space) another then clearly for any monotonic function the former should be retrieved first to approximate the $F$-ascending order. But consider a set of $n$ tuples in which none of the tuples dominate any other. In this case it is possible to define $n!$ monotonic functions such that for each of the functions the $F$-ascending order corresponds to a distinct order among the $n!$ possible orderings among $n$ tuples. Thus an order which is good for one feature expression can be arbitrarily bad for another even though both are monotonic. Hence trying to approximate the $F$-ascending order for multiple objective functions is not a good strategy.

The two problems cited above has to do with the fact that drawing tuples in $F$-ascending order is inherently difficult when the feature expression $F$ is an adhoc function. Obviously in developing an alternative solution we should try to overcome this problem. Is this the only problem which we should try to overcome in designing a generalized bounding strategy? Prior research provides no answer to this question and hence in the next subsection we investigate a class of aggregate functions for which drawing the tuples in $F$-ascending order is a non-issue. This allows us to isolate issues other than those arising from the $F$-ascending order problem.

### 4.2 Virtual Tuples Bounding Strategy

We distinguish between two classes of objective functions. In one class the feature expression involves only one measure of the base table, e.g., $2 \times M_1$. We call such objective functions *simple*. In the other case is called *composite* objective functions where the feature expression involves multiple measures, e.g., $M_1 + M_2$.

In this section we investigate an OOBS like bounding strategy when all the objective functions involved are simple. We note in passing that simple objective functions are the only kind of objective functions possible when the base table has only one measure. In this case the user might still want to compute multiple aggregates, e.g. the total and average sale per customer. The reason we can sidestep the two fundamental limitations of OOBS in simple objective

functions is as follows. Consider all the objective functions whose feature expressions involves a certain measure $M_i$. Then drawing the tuples in ascending order of $M_i$ satisfies the $F$-ascending order for all these objective functions since they are monotone and hence the increasing order of $M_i$ corresponds to the increasing order of the $F$-values. But how do we reconcile the $F$-ascending order simultaneously for objective functions over different measures? For example, if $\langle 1, 9\rangle$ and $\langle 9, 1\rangle$ are two tuples of a group and the aggregate functions are SUM on measure $M_1$ and $M_2$ respectively, then reading tuple $t_1$ after $t_2$ guarantees $F$-ascending order for the first aggregation function and not the second. Similarly drawing the second tuple after the first, will guarantee $F$-ascending order for the second aggregation function and not for the first. We can solve this problem if we can create tuples which are reordered as $\langle 1, 1\rangle$ and $\langle 9, 9\rangle$. Since the feature expression of every objective involves only one measure, such reordering preserves the correctness of aggregation. We call such reordered tuples as *virtual tuples*.

In general the virtual tuples bounding strategy (VTBS) works as follows. In a pre-processing step, we create virtual tuples by sorting each measure while fixing the other measures in place. Thus virtual tuples are created such that the tuple order is in ascending order for every measure. Then whenever the MOOLAP algorithm demands that another tuple of a group be read we read the next virtual tuple.

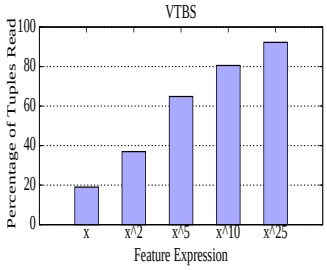
**Example 3:** As a more detailed example, consider the group {a,e} from Example 2. It has four tuples {(16,8), (6,3), (2,5) and (8,4)}. The corresponding virtual tuples are obtained by sorting independently each measure and are respectively {(2,3), (6,4), (8,5) and (16,8)}. □

**Observation 1.** *Aggregating over the virtual tuples preserves the correctness of aggregation and guarantees $F$-ascending order for every objective function.*

The above observation follows from the assumptions that the feature expressions are monotonic and also that the aggregation functions are simple, i.e. their output is affected only by one measure and is independent of other measures. VTBS consumes one tuple on every invocation from the MOOLAP algorithm. While this is the best strategy to minimize the total number of tuples consumed, it has I/O problems since the granularity of disk access is a page (block). Once a tuple is consumed, the group ranking function score ($L_1$Norm or such) of the corresponding group increases and hence the group may descend deeper into the priority queue $Q$ maintained by the MOOLAP algorithm. This could result in the same page being read from the disk multiple times unless there is a buffer hit. A simple solution to avoid this problem is to consume immediately and aggressively a page worth of tuples every time VTBS is invoked.

To make more observations we evaluated VTBS experimentally. The details of the experimental setup are provided in the experimental section later. But now we present representative results which allow us to draw some conclusions about VTBS. Note that we present results for only one workload here but they show representative behavior observed in many different workloads. Briefly, the workload description is as follows: there are 10000 groups in the workload. The number of tuples per group is normally distributed with an average of 5000. There are four measures $M_{1-4}$ and four objective functions which are $OBJ_1 = SUM(m_1^i)$, $OBJ_2 = SUM(m_2^i)$, $OBJ_3 = SUM(m_3^i)$ and $OBJ_4 = SUM(m_4^i)$. The value of $i$ was varied in the experiments and the results are shown in Figure 2. The $y$ axis plots the percentage of tuples drawn by the MOOLAP algorithm. For VTBS, the savings are impressive when the feature expression is linear but shows steady degradation as higher degree polynomial functions are used as the feature expression. This problem arises in VTBS (and hence also in OOBS) because the $F$-value of the last read tuple in a group is used to bound all the unread tuples. For functions of smaller rate of change such as the linear function the difference between the bound and the actual value of unread tuples is not as high as in functions which show larger rate of change such as $x^{10}$. As a result the obtained bound is very loose and hence a large number of tuples have to be retrieved before the bound has been tightened enough to discard the group. This high sensitivity to the nature of the feature expression is clearly undesirable since the solution should be efficient for a wide class of feature expressions.

Figure 2: Sensitivity to Feature Expression




### 4.3 Index Bounding Strategy

Based on our discussion of prior work and our experimental observation above we lay down the following four criteria for a useful bounding strategy. First, it should not rely on building an index or other such structures which depend on the query parameters. Second, it should be general enough to be applied to objective functions with both composite and simple feature expressions. Third, it should preferably use pages rather than individual tuples as the granularity of access. Fourth, it should not bound a large number of unseen tuples with a single value.

#### 4.3.1 Basic Idea

The basic idea is that if we precompute and store some succinct statistics about the tuples of a group, we may be able to come up with tight bounds. Our new strategy, which we call the *Index Bounding Strategy (IBS)*, works as follows. Partition the tuples of a group into different pages.

Precompute and store meta-information necessary to lower bound all the tuples in each page. Then the initial bound for the tuples is obtained by reading just the meta information. Subsequently whenever a group makes it to the head of the priority queue in the MOOLAP algorithm, its bound is refined by reading one unread page and replacing the estimate provided by the meta-information for that page with the actual information. The new strategy gives rise to a variety of questions. What kind of meta-information should be stored about each page? To improve the bound, in what order should different pages be read? How to decide which tuples are put together in the same page?

The meta-information stored about each page of tuples should be small in size so that it can be retrieved in a few I/Os as compared to reading all the data pages. If some of the bounds obtained using the meta-information turns out to be loose, it impacts only the corresponding pages of unread tuples and not the entire set of unseen tuples as was the case for VTBS. We divide the pages into index pages and data pages. Index pages hold the meta-data while data-pages store the corresponding tuples. The meta-data stored for each page is simple and small. For each data page, an entry in the index page stores the minimum value of each measure and the number of tuples in that data page. Since we assume that the feature expressions and aggregation functions are monotonic this information is sufficient to obtain a bound. For example, consider a group $g$ with one data page containing two tuples{(1,9), (9,1)}. The corresponding index page entry will hold (1,1) as the minimum measures for $g$ and 2 as the number of tuples. Suppose we have a query with one objective $OBJ_1 = A_1(F_1) = SUM(M_1 + M_2)$ then a lower bound for the objective is obtained as $SUM((1+1),(1+1)) = SUM(2,2) = 4$. Index pages are not created for groups having less than a page worth of tuples.

#### 4.3.2 Data Page Ordering Strategy

Initially, the index pages are read to obtain the initial bounds of groups. Then whenever the MOOLAP algorithm invokes the bounding strategy to read more tuples for a group, an unread data page of the group is read and the bound is tightened. In what order should the data pages of a group be read to ensure that only the minimum number of pages necessary are read? Consider the simple case where there is only one measure $M_1$ and one objective function, $OBJ_1 = SUM(M_1)$. Let $p$ be some data page with $n$ tuples $\{t_1, t_2, ..., t_n\}$ and let $t_{min}$ be the tuple with minimum $M_1$ value. Then the partial sum provided by the index pages for the data page $p$ is $t_{min} \times n$. Define $\delta$(p) to be $\Sigma_{i=1-n}(t_i) - t_{min} \times n$, i.e, $\delta$ denotes the error in the partial sum obtained from the index page entry for $p$. Then, in order to read only the minimum number of data pages of $g$, the data pages of $g$ should be retrieved in $\delta$-descending order. In order words, we can best refine the bound of $OBJ_1$ by fetching the data page with largest "bound estimation error" from the group.

But the intuition in this simple case is straightforward. Since we have only one objective function $SUM(M_1)$, every time a data page is read, the value of the bound increases by precisely the $\delta$ of the page. In reading the pages in $\delta$-descending order we ensure that after every page read we have the tightest bound possible. Bounds are improved by replacing estimates with their actual values. Therefore, intuitively, the tuples should be retrieved in the descending order so that the difference between the estimate and the actual $F$ values is maximized. But we cannot determine the $delta$ value of a page without actually retrieving all pages. Hence we need to approximate it with some statistics. Our strategy is to store, in addition to the minimum measure values, the average measure values with the meta-data associated with a page. Then we compute the difference between the $F$ score of the average values and the $F$ score of the minimum values and use it to approximate the $delta$-descending order.

When we have more than one objective function, we adopt a round-robin strategy. First the first objective function is used to choose the next block, then the second objective function is used to choose from among the remaining blocks and so on. This brings an element of fairness so that one objective function is not "deprived" of pages which improve its bound the best for long. Our implementation computes the schedule of data page ordering the first time the index pages are loaded. Then we use an in-memory cache to store this schedule and probe it whenever the corresponding group pops out from the priority queue from the MOOLAP framework.

#### 4.3.3 Tuple Grouping Strategy

Finally, which tuples should end up in the same page? Intuitively tuples which are geometrically close together should end up in the same page as it reduces the error in the estimation provided by the index pages. This is the strategy we adopt even though it penalizes objective functions which show sharp gradation in some narrow region. We divide the tuples into different pages by using $k$-dimensional equi-width partitioning, where $k$ stands for the number of measures. Pages corresponding to sparse buckets of the equi-width partitioning are combined together at the end of pre-processing. The rest of the issues discussed above are orthogonal to the partition strategy adopted and hence alternate partition strategies such as $k$-means clustering can be adopted easily. We explore this point further in the experimental section.

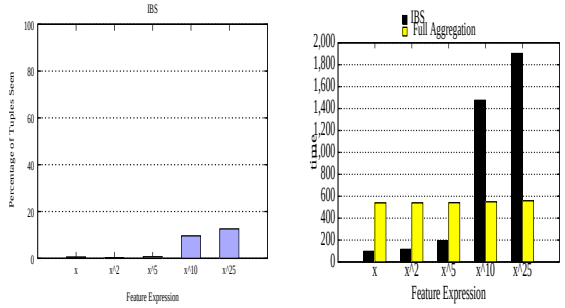
### 4.4 Sequential Index Bounding Strategy

#### 4.4.1 Observations from the Index Bounding Strategy

Figure3 shows the sensitivity of IBS to the feature expression. It can be seen that IBS shows much less sensitivity to the nature of the feature expression as the index pages together provide a fairly tight bound. IBS has satisfied all the four criteria we laid down above. The preprocessing partitions tuples based on their position in the measure space and is independent of the query. Both composite and sim-

ple functions are supported by IBS. Page is the fundamental unit of access and one value per page is used to bound the unseen tuples. Finally as it can be seen from $2\ b$, on an average IBS has to read only a small percentage of the tuples - an impressive savings in terms of the tuples not read. It would appear that we have designed a solution that has met all our goals.

Figure 3: Performance of IBS




Unfortunately, previous research in top-$k$ research [17, 5] and also the properties we developed assumed that savings in terms of number of tuples consumed is necessary and sufficient for savings in time and hence optimized only the number of tuples. Figure4 shows that other factors have come into play which has resulted in not only the savings in number of tuples not translating into savings in time but has also resulted in a large slow down. The "naive" method of computing all the aggregates followed by computing the skyline using the LESS [10] algorithm has not only beaten our supposedly "intelligent" method but has done so by an order of magnitude. Thus IBS has resulted in an algorithm which consumes only a small fraction of the tuples but results in an order of magnitude slow down. *Where did the time go*?

The main reason behind the above mentioned slow down is disk behavior. Recall that we assumed that the grouping part of the problem is pre-materialized. Our implementation of this part ensures that the tuples are stored sequentially on disk. This is achievable under most circumstances since it is a pre-processing step. Sequential read of a data page is orders of magnitude faster than the a read which involves a random seek. This disparity in access time offsets the advantage derived from consuming less tuples (data pages). We note in passing that pre-grouping the data followed by sequential access for full aggregation at query time was not one of the methods experimentally evaluated in the top-$k$ case [17] and hence this problem was not uncovered earlier.

### 4.4.2 Incorporating the Disk Behavior

Before developing a bounding strategy that takes into account these factors, we should first answer the question if it is even necessary to develop a more complicated method than the full aggregation method? We believe it is necessary for the following reasons. First of all, the workload used in the above experiment is a small workload. For larger workloads even with sequential access the full aggregation method takes considerable time to terminate and hence any significant savings in the time taken to terminate is useful. Secondly and more importantly remember that our goal is to allow the user to use skyline queries in an interactive manner. Hence in order to keep the user engaged we should output potential skyline points. Potential skyline groups are those groups which may not be actually be in the final skyline set but are still very good in terms of the user's objectives. The full aggregation method can also compute potential skyline groups by repeatedly computing the skyline over the groups whose aggregation has been completed. This has two problems. First it loses the sequential advantage since it has to perform a random seek to compute the skyline after the aggregation of every group is complete. Also the quality of potential skyline points may be very poor since in order to outwit this method all an adversary has to do is hide the groups which are good in terms of the user's objectives later in the file. Thus if we can develop a bounding strategy which takes considerably less time than the time taken by the full aggregation method while outputting both the final skyline aggregates and the potential skyline aggregates of very high quality early, it is worth the effort.

Now we develop a bounding strategy called the sequential index bounding strategy (SIBS) that takes into account the disk access time. Our strategy is not to outperform the advantage offered by sequential access but rather to use

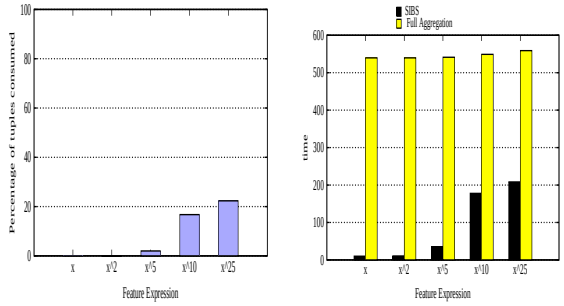
**Algorithm 2** Moolap Using SIBS

```
1: for each group g ∈ G do
2:    g.bound = calculateBound(g.minMeasures, g.numTuples)
3:    for each index page I of g do
4:       for each indexPageEntry ie of I do
5:          g.bound = updateBound(g.bound, ie.minMeasures, ie.numTuples)
6:       end for
7:    end for
8:    Q.insert(L1Norm(g.bound), g)
9: end for
10: S = PS = {}
11: LOOP:
12: while Q is not empty do
13:    g = Q.getMin();
14:    if g ∈ PS then
15:       PS.erase(g)
16:    end if
17:    if dominated(S, g) or dominated(PS, g) then
18:       discard g and goto LOOP
19:    end if
20:    if g.unreadTuples == 0 then
21:       S = S ∪ {g} and goto LOOP
22:    end if
23:    while (DataPage page = g.readNextPage()) != NIL do
24:       minMeasures = page.minMeasures
25:       for each tuple t in page do
26:          updateBound(g, minMeasures, t)
27:       end for
28:       if dominated(S, g) or dominated(PS, g) then
29:          discard g and goto LOOP
30:       end if
31:    end while
32:    for each e in PS do
33:       if g.bound dominates e then
34:          PS.erase(e)
35:       end if
36:    end for
37:    if PS.hasSpace() then
38:       PS.add(g)
39:    end if
       Q.insert(L1Norm(g.bound), g)
40: end while
```

it while still trying to consume less tuples. The complete MOOLAP algorithm using SIBS as the bounding strategy is listed in Algorithm 2. SIBS maintains a set of potential skyline groups in addition to the set of final skyline groups. Initially both sets are empty. In a pre-processing step, SIBS builds index pages similar to IBS but this time the index pages store only the minimum measures and not the average measures. Thus the index page entries are more compact in SIBS when compared to IBS. All the index pages are read in the beginning to compute the initial bound of all groups and the priority queue of the MOOLAP algorithm is organized as usual (`lines 1-9`). The head of the priority queue is popped and let $g$ be the corresponding group. If $g$ is dominated by some group in either the potential skyline set or the final skyline set, $g$ is discarded (`lines 14-20`). If it is not dominated then instead of inserting the group back into the priority queue, the sequentially next page of $g$ is read and the process is repeated (`lines 24-32`). When all the pages of $g$ are read, $g$ is inserted back into the priority queue and also into the potential skyline set if the potential skyline set has size less than its fixed capacity. When $g$ makes it to the head of the priority queue again and if it is still not dominated by either the members of the potential skyline set or the final skyline set it is removed from the potential skyline set and added to the final skyline set. Note that in this bounding strategy a group arrives at the head of the priority queue at most twice. The first time it is either discarded after reading a few pages or all its pages are read. The second time it is added to the final skyline set if it is not dominated. SIBS is so named since it draws the pages of a group by sequential access. Unlike IBS, SIBS does not intelligently schedule the data pages within the group but instead draws the pages in stored order. Clearly this would increase the percentage of tuples consumed which is shown in Figure4. But Figure4 also shows that this increase is offset by the time advantage due to the sequential accesses. From Figure4 it can be seen that SIBS is significantly faster than full aggregation.

Since SIBS and IBS are on the two opposite ends of trading sequential access against the number of data page transferred. It is worth noting that the benefits of sequential access used in SIBS can definitely be combined with the intelligent data page scheduling in IBS to harness the best of both worlds. In this paper, we do not explore further into such a hybrid solution and leave it as the future work.

Figure 4: Performance of SIBS




# 5 EXPERIMENTS

## 5.1 Experiment Setup

In this section we conduct extensive experiments to explore the design space of the algorithms. We have implemented the MOOLAP algorithm framework and the bounding strategies discussed in the paper in C++. Since the Sequential Index Bounding Strategy (SIBS) is the final, complete solution that we have arrived at in the paper, throughout this section, we use MOOLAP to refer to the MOOLAP framework that employs the SIBS as the underlying bounding strategy. The MOOLAP algorithm leverages on an external memory priority queue described in [24] with a cache size of 20MB. We have implemented the block-based storage manager from scratch. Each data page size is fixed at 4KBytes throughout the experiments. For comparison, we have also implemented the full aggregation algorithm and the LESS algorithm[10] as our baseline algorithm for the skyline aggregation computation. The experiments were conducted on a dual pentium-4 processor machine with 1GB of main memory and a 500GB IDE hard disk. A preemptible version of the 2.6.17 Linux SMP kernel was used as the operating system. Note that in all our experiments we read directly from disk to user space bypassing the kernel space and hence also the kernel level buffering. Neither the MOOLAP algorithm nor the baseline full aggregation read the same data page more than once in the same run and hence buffering has no impact. However given the sequential access pattern of MOOLAP (with the sequential index bounding strategy) and baseline solution, both could benefit from aggressive read-ahead. This is studied in a separate experiment and by default the aggressive read-ahead is turned off.

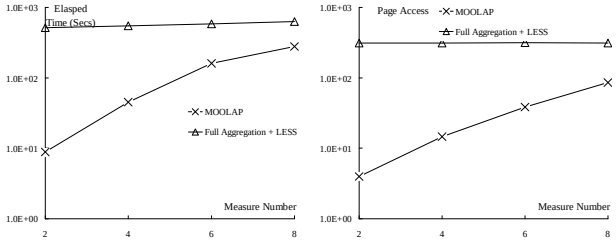
One of the fundamental problems in attempting to evaluate the performance of ad-hoc user-defined queries lies in designing representative queries that capture many different possibilities. Since the scope of this paper is novel, there does not exist a public query benchmark for our problem. Therefore we define our own objective functions. Recall that objective functions have two components, the aggregation function and the feature expression. For the aggregation functions, SUM and AVERAGE are used. For the feature expressions, we define various polynomials over the measure attributes. The advantage of using polynomials is that it allows us to define a large family of feature functions showing various rate of changes by changing weights and powers. For simple feature expressions, we use polynomials of the form $F_i = m_i^p$ where $i$ varies from 1 to the number of objectives and $p$ determines the rate of change of the feature expression. For the composite feature expressions, we use polynomials of the form $Fi = \Sigma m_i^p$ where $i$ varies from 1 to the total number of measures and $p$ varies from 1 to the number of objectives.

The experiment database consists of the group-by attributes and the measure attributes. The measure attributes of each data tuple are generated by the widely used generator [4] in the previous skyline literature. The total number of groups is fixed in each experiment and the number of

tuples belonging to each group follows the normal distribution. We use the following defaults for the parameters except in experiments where they are varied. The number of groups is 10000. The number of tuples per group is normally distributed with an average equal to 5000 and standard deviation of 0.2. The tuple size is fixed at 100 bytes irrespective of number of measures (using padding when necessary). In other words, our default database size consists of 50M records or 5G bytes in disk size. The default partition method used is the equi-width partition. The default maximum buckets number is 100 and the maximum allowed size of potential skyline groups in Sequential Index Bounding Strategy (SIBS) is 200. The default power of simple polynomials in feature expressions is 5. The default number of measures and objectives are both 4. All the experiments have been run over both uniform and anti-correlated measures. We show only the results for the anti-correlated case, since it is considered as the most challenging and closest to the real world scenario of optimizing conflict objectives (results are consistent over other distributions as well).

## 5.2 Experiment Results

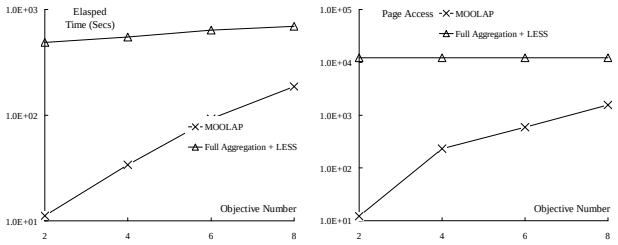
### 5.2.1 Impact of number of measures




(a) Elapsed Time (b) I/O

Figure 5: Effects of the Number of Measures (Anti-correlated Measures, 10K Groups, 5K Tuples Per Group on Average, 5GBytes)

Figure 5 shows the impact of the number of measures on the performance of MOOLAP in terms of both elapsed time and the I/O cost. Composite feature expressions are used here to combine all the measures in each objective function. Note that even though we vary the number of measures, the tuple size is fixed. Thus the primary impact of the increasing number of measures is the increasing error contributed to the bound obtained from the corresponding index pages. When the number of measures is less than 5, MOOLAP is order of magnitude faster than the baseline. For higher number of measures, MOOLAP is still around 50-60 percent faster. The correlation between the measures affects the performance when the number of measures are changed. But the effects are uneven. Note that the baseline method also shows a slight increase in time with increase in number of measures. That is due to the extra computational cost needed to process the extra measures.




(a) Elapsed Time (b) I/O

Figure 6: Effects of the Number of Objectives (Anti-correlated Measures, 10K Groups, 5K Tuples Per Group on Average, 5GegaBytes)

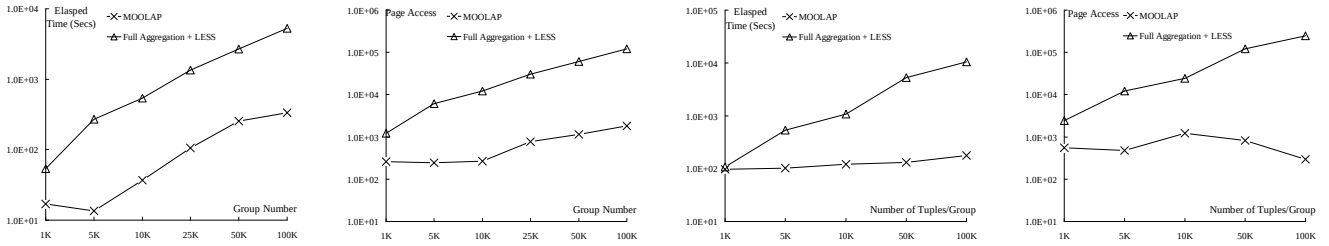
### 5.2.2 Impact of number of objective functions

The impact of the number of objectives on the performance of MOOLAP is very similar to that of number of measures and is shown in Figure 6. Again, if the number of objectives is less than 5, MOOLAP is much faster than baseline methods and for more than 5 objectives, MOOLAP is still considerably faster. The cost of the baseline increases with increasing objectives, again, due to the extra computational cost. With the increase of objectives, MOOLAP performance degrades due to the fact that more groups are in the final skyline results and hence more I/Os need to be performed to read pages from the extra groups.

### 5.2.3 Scalability of MOOLAP

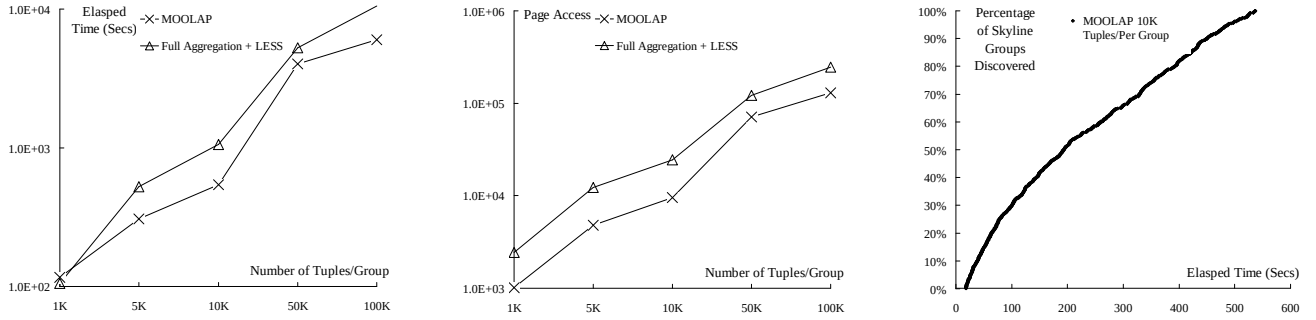
Figure 7 examines the scalability of the MOOLAP algorithm in terms of the number of groups and tuples respectively. By varying both the average group size and the group number from 1K to 100K, we have covered all sizes of databases up to 100GBytes. In both cases, the MOOLAP algorithm scales much better than the baseline aggregation method which has to consume every tuple. In all the four figures, MOOLAP consistently outperforms the baseline method by orders of magnitude. For example, on the 100GBytes databases, MOOLAP completes within 3 minutes whereas the baseline method takes several hours.

An interesting observation comes in Figure 7(c). With the increase of average group size (from 10K to 100K), MOOLAP even becomes faster. Although counterintuitive, this is quite reasonable. The number of final skyline groups does not increase with the group size since the total group number is fixed in this experiment. The only effect of a larger average group size is the greater variance in the objective space, i.e. the probability of a "killer group" being present increases. Such groups are good in terms of every objective and thus dominate a large region of the objective space. It should be also noted that the inability to exploit such killer groups is one of the major disadvantages of the baseline method. Finally, it is clear that anti-correlation of the measure attribute is insufficient to bring out the worst case behavior of the MOOLAP algorithm. So, the next experiment studies the anti-correlation of the objectives.



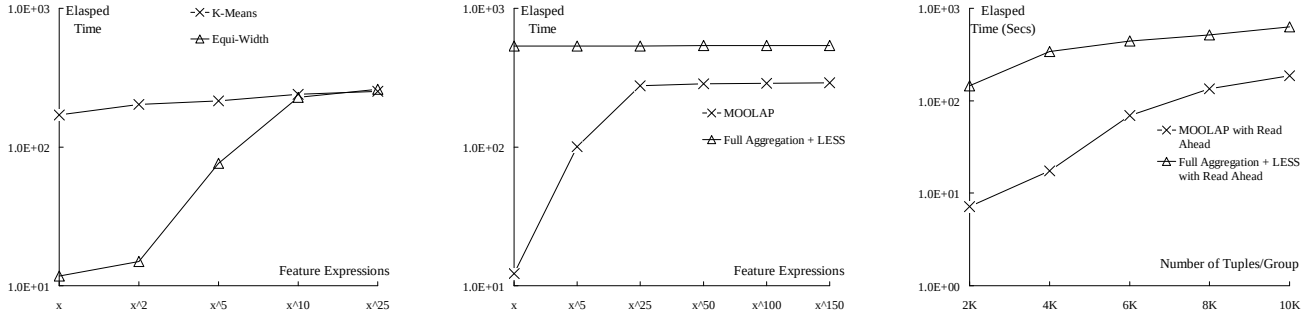

(a) Elapsed Time v.s. Number of Groups (b) I/O v.s. Number of Groups (c) Elapsed Time v.s. Average Number of Tuple Per Group (d) I/O v.s. Average Number of Tuple Per Group

Figure 7: Scalability of MOOLAP Algorithm (Anti-correlated Measures, 10K-100K Groups, Average 1K-100K Tuples/Group, 5-100GegaBytes)




(a) Elapsed Time (Anti-Correlated Objectives) (b) I/O (Anti-Correlated Objectives) (c) Demonstration of the Progressiveness of MOOLAP (10K Groups, Average 10K Tuples/Group, 10GegaBytes

Figure 8: Study of Anti-Correlated Objective Space




(a) Comparison of Data Partitioning Methods (b) Effects of Different Feature Expressions (c) Effects of System Read-Ahead

Figure 9: Other Experiments

### 5.2.4 Study of Anti-correlated Objectives

Note that just because the measures are anti-correlated does not mean the aggregates are anti-correlated since aggregation does not preserve correlation even if the aggregate functions used are simple. In this experiment, we study the performance of MOOLAP on data sets that are anti-correlated in the *objective space*. To simulate, anti-correlation in the objective space, we first generated the aggregate result and then split it into random parts. Note that this technique only works when the feature expression is linear and hence for this experiment only, a linear composite feature expression was used. As it can be seen from the curve in Figure 8(a) and 8(b), MOOLAP also suffers from the "curse of anti-correlation" which afflicts the skyline operator. As it can be seen from this experiment, unlike top-$k$, a query optimizer should be more careful while pushing the skyline operator into other operators. However, MOOLAP still outperforms the baseline algorithm by a great margin (50% in both time and I/O cost).

Furthermore, we would like to emphasize the progressive nature of the MOOLAP algorithm, i.e. most skyline groups have been identified much earlier than the whole process returns. To demonstrate, we show in Figure 8(c)

the percentage of skyline groups discovered over the time domain. While the whole MOOLAP process takes more than 500 seconds (which is already 50% faster than the baseline), most skyline groups are computed much earlier. For example, 20 percent of the skyline aggregates are discovered in around a minute with the first several skyline groups found almost instantaneously. Therefore, these aggregates can be quickly shown to the user by highlighting the corresponding cube cells, greatly improving the system usability.

#### 5.2.5 Impact of Partition Methods

Figure 9(a) shows the impact of the partition method used on performance. Partitioning methods matters since the bounds obtained in MOOLAP critically depends on the error rates of the the estimation computed from the index entries. We compare the default equi-width partition method with $k$-means clustering using different simple feature expressions. It can be seen that for smaller polynomials the equi-width method is order of magnitude faster than $k$-means clustering. This is probably due to the fact that $k$-means clusters tuples on the basis of the centroids of buckets while the way the index pages are built demands that tuples be as close as possible to the plane defined by the minimum measures. This disparity vanishes when feature expressions of higher rate of increase are used wherein $k$-means and equi-width perform equally bad since the error ratio begin to dominate. Nevertheless, we believe that feature expressions showing lower rate of change are much more realistic than higher rate of change such as $F(x) = x^{25}$

#### 5.2.6 Effects of Different Feature Expressions

Figure 9(b) shows the impact of the rate of change of the feature expression on performance. Initially, when the feature expression changes from $x^1$ to $x^{25}$ there is a sharp decrease in the performance of the MOOLAP algorithm. But, beyond this point to even higher rates of increase seems to have little effect. Thus, while the performance degrades as the rate of change of the feature expression increases, the performance stabilizes beyond a certain point. This trend is analogous to the results in Figure 9(a). The bound estimation degrades with the increase of the rate of changes in feature expressions, which causes MOOLAP to spend more time on fully fetching data pages from unpromising groups.

#### 5.2.7 Impact of Aggressive Read-ahead

Figure 9(c) shows the impact of implementing aggressive read-ahead on both the MOOLAP algorithm and the baseline algorithm. The performance of both algorithms improve significantly. This is mainly due to the fact that the full aggregate algorithm's I/Os are composed entirely of sequential reads while in MOOLAP with SIBS the I/Os are predominantly sequential interspersed with few random seeks. MOOLAP consistently outperforms the baseline by one order of magnitude.

It should be noted that these improvements were observed when the system was quiescent otherwise. When multiple queries were run concurrently the benefits of read-ahead rapidly disappeared.

## 6 RELATED WORK

Skyline operator was introduced in [4]. There is a significant amount of recent work exploring various aspects of the skyline operator. The computation of the skyline itself is discussed in [16, 21] etc. But none of them target an OLAP environment. The benefits of progresssiveness in skyline query processing was discussed and studied in [16, 21, 27]. There is a fairly large body of work, similar in spirit to this paper, that attempt to apply the skyline operator in contexts beyond the traditional relational database. Continuous skyline queries over data streams is studied in [19, 20]. Using approximation to counter the bad selectivity of the skyline operator in high dimensions is studied in [6]. Particularly relevant to a data warehousing environment is [30] which studies the problem of maintaining a skyline view in the presence of inserts and deletes. Huang et al. [15] study skyline query in mobile adhoc network environments. Skyline query in distributed settings is studied in [2, 28, 29, 31].

Prior research closest in spirit to our work are [17] and [18] both of which deal with the problem of top-k queries in OLAP settings. In particular the idea of lower bounding unread tuples was used in [17]. Work in iceberg queries and iceberg cubes [9, 3] also share similar motivations in trying to identify interesting parts of the data cube on the basis of user specified criterion. Our idea to output potential skyline groups early was motivated by online aggregation studied in [13, 12].

## 7 CONCLUSION

On-line analytical processing (OLAP) is crucial for the success of diverse futuristic decision making strategies. In general, OLAP operates on ever increasingly large data sets and is expected to provide on-line support for different adhoc criteria. In this paper, we identified and introduced the skyline operator for OLAP applications. The skyline operator is an instance of the multi-objective optimization problem, which generalizes many of the previous database work, including ranking queries such as top-k. The multi-objective optimization problem poses many novel and interesting challanges in a multi-dimensional space. In this paper, we investigated a family of algorithms to address this challenge. Using the properties of the algorithms, we were able to develop efficient algorithms that use succinct meta data to optimize the number of tuples retrieved to answwer the skyline query. The algorithms were implmented in a prototype using real storage settings. This prototype identified the interesting and realisitc problem, not previously identified in prior aggreation ranking work, of the

classic tradeoff between sequential and random access. As a result, we modified our algorithms to exploit sequential access when possible, even at the cost of sometimes retrieving more tuples. The final algorithm, MOOLAP with SIBS, is shown to be superior to all prior attempts using a variety of data sets and objective functions.

Various possible avenues of future work related to the skyline over aggregates problems exist. One interesting problem is to compute the skyline at the level of each cuboid in a data cube apriori and store it. This presents a much more compact view of the data cube and hence can be stored without the space overhead which prevents the full data cube from being materialized. Note that this is distinct from the skyline cube problem explored in [32, 23]. Another interesting avenue is to efficiently allow the user to add or remove objectives on the fly. While this can be achieved by repeatedly invoking the MOOLAP algorithm, we hypothesize that it should be possible to use the computed skyline efficiently in computing the skyline in the new objective space. A related problem is to integrate skyline with drill-down and rollup operations. A much more general direction is to investigate the incorporation of more multi-objective query operators rather than just the skyline which is but one specific instan of the multi-objective optimization problem. In this paper, we essentially pushed the skyline operator into the aggregation operator. It would be interesting to investigate when and if the skyline operator should be pushed into other operators while generating efficient query plans. Reference [6] would be a good starting point for studying query optimization issues related to the skyline operator.

## References


[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multi dimensional aggregates. In *Proc. of VLDB*, 1996.

[2] W. Balke, U. Guntzer, and X.Zheng. Efficient distributed skylining for web information systems. In *Proc. of EDBT*, 2004.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. of SIGMOD*, 1999.

[4] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of ICDE*, 2001.

[5] K. C.-C. Chang and S. won Hwang. Minimal probling: supporting expensive predicates for top-k queries. In *Proc. of SIGMOD*, 2002.

[6] S. Chaudhuri, N. Dalvi, and K. Raghav. Robust cardinality and cost estimation for skyline operator. In *Proc. of ICDE*, 2006.

[7] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM TODS*, 31(2), 2006.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of PODS*, 2001.

[9] M. Fang, N. Shivakumar, H. Garcia-Molina, and R. Motwani. Computing iceberg queries efficiently. In *Proc. of VLDB*, 1998.

[10] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proc. of VLDB*, 2005.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. of ICDE*, 1996.

[12] P. Hass and J. Hellerstein. Ripple joins for online aggregation. In *Proc. of SIGMOD*, 1997.

[13] J. Hellerstein, P. Hass, and H. Wang. Online aggregation. In *Proc. of SIGMOD*, 1997.

[14] G. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2), 1999.

[15] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *Proc. of ICDE*, 2006.

[16] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *Proc. of VLDB*, 2002.

[17] C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting adhoc ranking aggregates. In *Proc. of SIGMOD*, 2002.

[18] H. Li, H. Yu, D. Agrawal, and A. E. Abbadi. Progressive ranking of range aggregates. In *Proc. of DaWak*, 2005.

[19] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proc. of ICDE*, 2005.

[20] M. Morse, J. Patel, and W. Grosky. Efficient continuous skyline computation. In *Proc. of ICDE*, 2006.

[21] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proc. of SIGMOD*, 2003.

[22] D. Papadias, Y. Tao, F. Greg, and B. Seeger. Progressive skyline computation in database systems. *ACM TODS*, 30(1), 2005.

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

[24] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[25] S. Sarawagi. User-adaptive exploration of multidimensional data. In *Proc. of VLDB*, 2000.

[26] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *Proc. of EDBT*, 1998.

[27] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. of VLDB*, 2001.

[28] A. Viachou, C. Doulkeridis, M. Vazirgiannis, and Y. Kotidis. Skypeer: Efficient subspace computation over distributed data. In *Proc. of ICDE*, 2007.

[29] S. Wang, B. C. Ooi, A. K. Tung, and L. Xu. Efficient skyline query processing in peer-to-peer networks. In *Proc. of ICDE*, 2007.

[30] P. Wu, D. Agrawal, A. E. Abbadi, and O. Egecioglu. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *Proc. of ICDE*, 2007.

[31] P. Wu, C. Zhang, Y. Feng, B. Y.Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *Proc. of EDBT*, 2006.

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