Interactive Data Exploration Using Semantic Windows

Alexander Kalinin
Brown University
akalinin@cs.brown.edu

Ugur Cetintemel
Brown University
ugur@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

We present a new interactive data exploration approach, called *Semantic Windows* (SW), in which users query for multidimensional "windows" of interest via standard DBMS-style queries enhanced with exploration constructs. Users can specify SWs using (i) *shape-based* properties, e.g., "identify all 3-by-3 windows", as well as (ii) *content-based* properties, e.g., "identify all windows in which the average brightness of stars exceeds 0.8". This SW approach enables the interactive processing of a host of useful exploratory queries that are difficult to express and optimize using standard DBMS techniques.

SW uses a sampling-guided, data-driven search strategy to explore the underlying data set and quickly identify windows of interest. To facilitate human-in-the-loop style interactive processing, SW is optimized to produce online results during query execution. To control the tension between online performance and query completion time, it uses a tunable, adaptive prefetching technique. To enable exploration of big data, the framework supports distributed computation.

We describe the semantics and implementation of SW as a *distributed* layer on top of PostgreSQL. The experimental results with real astronomical and artificial data reveal that SW can offer online results quickly and continuously with little or no degradation in query completion times.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

Keywords

Data exploration; Query processing

1. INTRODUCTION

The amount of data stored in database systems has been constantly increasing over the past years. Users would like to perform various exploration tasks with interactive speeds.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593666.

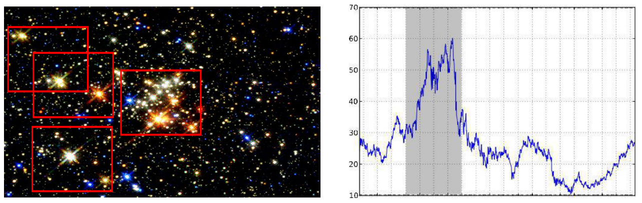

Figure 1: Exploration queries searching for bright star clusters (left) and periods of high average stock prices (right). Some results are highlighted.

Unfortunately, traditional DBMSs are designed and optimized for supporting neither interactive operation nor exploratory queries. Users typically have to write very complex queries (e.g., in SQL), which are hard to optimize and execute efficiently. Moreover, users often have to wait until a query finishes without getting any intermediate (*online*) results, which is often sufficient for exploratory purposes.

Consider the following data exploration framework. A user examines a multidimensional data space by posing a number of queries that find rectangular regions of the data space the user is interested in. We call such regions *windows*. After getting some results, the user might decide to stop the current query and move to the next one. Or she might want to study some of the results more closely by making any of them the new search area and asking for more details. Let us look at two illustrative examples in Figure 1.

Example 1. A user wants to study a data set containing information about stars and other objects in the sky (e.g., SDSS [1]). She wants to find a rectangular region in the sky satisfying the following properties:

- The shape of the region must be 3° by 2°, assuming angular coordinates (e.g., right ascension and declination).
- The average brightness of all stars within the region must be greater than 0.8.

The example above describes a spatial exploration case, where windows of interest correspond to two-dimensional regions of the sky. However, the framework can be used for other cases, e.g., for one-dimensional time-based data:

Example 2. A user studies trading data for some company. The search area represents stock values over a period of time. The user wants to find a time interval (i.e., a one-dimensional region) having the following properties:

- The interval must be of length from 1 to 3 years.
- The average of stock prices within the interval must be greater than 50.

By defining windows of interest in terms of their desired properties, users can express a variety of exploration queries. Such properties, which we call *conditions*, can be specified based on the shape of a window (e.g., the length of a time interval) and an aggregate function of the data contained within (e.g., the average stock value within an interval).

Traditional SQL DBMSs cannot deal with the type of queries presented here in an efficient and compact way. Available constructs, such as `GROUP BY` and `OVER`, do not allow users to define and explore all valid windows. Although it is possible to express such queries with a SQL query involving a number of recursive Common Table Expressions (CTEs), such a query would be difficult to optimize due to its complexity.

Another important requirement in data exploration is interactivity. Since the amount of data users have to explore is generally large, it is important to provide online results. This allows the user to interrupt the query and modify it to better reflect her interests. Moreover, many applications can be satisfied with a subset of results, without the need to compute all of them. Most SQL implementations do not allow such functionality, making users wait for results until the entire query is finished.

Our approach treats an exploration query as a data-driven search task. The search space consists of all possible windows, which can vary in size and overlap. We use a cost-benefit analysis to quantify a utility measure to rank the windows and decide on the order by which we will explore them. We use sampling to estimate values for conditions on the data, which allows us to compute a distance from these values to the values specified in the query. We then guide the search using a *utility* estimate, which is a combination of this distance, called *benefit*, and the *cost* of reading the corresponding data from disk. We use shape-based conditions to prune parts of the search space.

Since logical windows may not correspond to the physical placement of data on disk, reading multiple dispersed windows may result in a significant I/O overhead. The overhead comes from seeks and reading the same data pages multiple times. Thus there is a tension between online performance and query completion time. To control this tension, we use a prefetching technique: by prefetching additional data when reading a window and thus decreasing the number of dispersed reads, it is possible to reduce the overhead and finish the query faster. However, this approach might result in worse online performance, since delays for online results might increase. To manage this trade-off, we use an adaptive prefetching algorithm that dynamically adjusts the size of prefetching during execution. While new results are being discovered, we prefetch less to keep delays to a minimum. When no new results can be found, we start to prefetch more to speed up the execution and finish the query faster. Additionally, we allow users to control the prefetching strategy to favor one side of the trade-off or the other.

Our new framework, which we call *Semantic Windows* (SW), can be implemented as a layer on top of existing DBMSs. For experimentation purposes, we implemented a *distributed* version of the framework on top of PostgreSQL. The computation is done by multiple *workers* residing on different nodes, which interact between themselves and with the DBMS via TCP/IP. To explore windows efficiently, we require efficient execution of multidimensional range queries, which can be achieved via common index structures, like B-trees or R-trees. To estimate utilities we collect a sample of the data offline and store it in the DBMS.

We conducted an experimental evaluation using synthetic and real data. For the latter we used SDSS [1]. The experimental results show that in many cases our framework can offer online results quickly and continuously, outperforming traditional DBMSs significantly. For the comparison we used a complex SQL query, mentioned above, and query completion time of SW queries is often better. Due to the complexity of the query, the query processor was not able to plan and execute it efficiently. However, even when the framework performs worse in total time, the difference can be eliminated or significantly reduced by prefetching.

In summary, the proposed SW framework:

- Allows users to interactively explore data in terms of multidimensional windows and select interesting ones via shape- and content-based predicates.
- Provides online results quickly, facilitating human-in-the-loop processing needed for interactive data exploration.
- Uses a data-driven search algorithm integrated with stratified sampling, adaptive prefetching and data placement to offer interactive online performance without sacrificing query completion times.
- Offers a diversification approach that allows users to direct the search for qualifying SWs in unexplored regions of the search space, thus allowing the user to control the classical exploration vs. exploitation trade-off.
- Provides an efficient distributed execution framework that partitions the data space to allow parallelism while effectively dealing with partition boundaries.

The remainder of the paper is organized as follows. Section 2 gives a formal description of the framework. Section 3 describes extensions for SQL to support SW queries. Section 4 gives a description of the algorithm and our prefetching technique. Section 5 talks about the architecture and implementation details. Section 6 provides results of the experimental evaluation. Section 7 reviews related work. Section 8 concludes the paper.

2. SW MODEL AND QUERIES

Assume a data set S containing objects with attributes $a_1, \dots, a_m$ (e.g., brightness, price, etc.) and coordinates $x_1, \dots, x_n$. Thus, S constitutes an n-dimensional search area with dimensions $d_1, \dots, d_n$. We will often specify S in terms of the intervals it spans (e.g., $S = [L_1, U_1) \times [L_2, U_2)$ for a two-dimensional data set). Next, we define a *grid* on top of S. The grid G_S is defined as a vector of *steps*: $(s_1, s_2, \dots, s_n)$. It divides each interval $[L_i, U_i)$ into disjoint sub-intervals of size s_i, starting from L_i: $[L_i, L_i + s_i) \cup [L_i + s_i, L_i + 2s_i) \cup \dots \cup [L_i + k \cdot s_i, U_i)$. The last sub-interval might have size less than s_i, which has no impact on the discussion. Thus, S is divided into a number of disjoint *cells*. The search space of an SW query consists of all possible *windows*. A window is a union of adjacent cells that constitutes an n-dimensional rectangle. Since the grid determines the windows available for exploration, the user can specify a particular grid for every query.

Let us revisit examples presented in the Introduction. Example 1 could be represented as follows:

- $d_1 = ra$, $d_2 = dec$, $a_1 = brightness$ [1].
- $S = [100°, 300°) \times [5°, 40°)$, $G_S = (1°, 1°)$.

and Example 2 as:

- $d_1 = time$, $a_1 = price$.

- $G_S = (1\,year)$, $S = [1999/02/01, 2003/11/30)$

An *objective function* $f(w)$ is a scalar function, defined for window w. There are two types of objective functions:

- *Content-based.* They are computed over objects belonging to the window. Since the value must be scalar, this type is restricted to aggregates (average, sum, etc). We further restrict them to be *distributive and algebraic* [6]. This is done for efficiency purposes — the value of $f(w)$ must be computable from the corresponding values of the cells in w. We discuss this in more detail in Section 5.
- *Shape-based.* They describe the shape of a window and do not depend on data within the window. We restrict ourselves to the following functions: $card(w)$ and $len_{d_i}(w)$. The former defines the number of cells in w, which we call the *cardinality* of w. The latter is the length of w in dimension d_i in cells. Assuming w spans interval $[l_i, u_i)$ in d_i, $len_{d_i}(w) = \frac{u_i - l_i}{s_i}$. Other functions, for example computing the perimeter or area, are possible.

A *condition* c is a predicate involving an objective function. The result of computing condition c for window w is denoted as w_c. The framework is restricted to algebraic comparisons, for example $f(w) > 50$.

The conditions for Example 1 can be defined as: $len_{ra}(w) = 3$, $len_{dec}(w) = 2$ and $avg\_brightness(w) > 0.8$. For Example 2, the conditions can be expressed as: $len_{time}(w) \geq 1$, $len_{time}(w) \leq 3$ and $avg\_price(w) > 50$.

An SW query can now be defined as $Q_{SW} = \{S, G_S, C\}$, where C is a set of conditions. The result of the query is defined as: $RES_Q = \{w \in W_S | \forall c \in C : w_c = true\}$, where W_S is a set of all windows, defined by G_S.

3. EXISTING SQL EXTENSIONS FOR DATA EXPLORATION

As a first option, we look into expressing SW queries using SQL. While SQL has constructs for working with groups of tuples, such as `GROUP BY` and `OVER`, they are insufficient for expressing all possible windows. `GROUP BY` does not allow overlapping groups, which makes it impossible to express overlapping windows. `OVER` allows users to study a group of tuples (also called window in SQL) in the context of the current tuple via `PARTITION BY` clause. However, it allows only one such a group for every tuple, not a series of groups with different shapes. Thus, only a subset of possible windows can be expressed this way. The standard SQL is even more restrictive and does not allow functions to be used in `PARTITION BY`. This makes it difficult to express multidimensional windows at all.

One general way, which we implemented, is to express an SW query as follows:

1. Compute objective function values for every cell of the grid via `GROUP BY`.
2. Generate every possible window by combining cells, using recursive CTEs. Values for the objective functions are computed by combining the values of the cells.
3. Filter windows that do not satisfy the conditions.

This type of query leads to two major problems. First, due to its complexity most traditional query optimizers would likely have hard time executing the query efficiently. As an example, we provide experimental results for PostgreSQL in Section 6. More importantly, the query performs an aggregation in the beginning. This means the computation is blocked until all cells have been computed. Such a query would not be able to output online results. Techniques like *online aggregation* [8] are very limited here, since exact, not approximate, results are required. Also, applying online aggregation to such a complex query is challenging at best.

[1]The original SDSS data does not contain a brightness attribute. However, this attribute or a similar one can be computed from other attributes using an appropriate function.

```
SELECT LB(ra), UB(ra), LB(dec), UB(dec),
       AVG(brightness)
FROM sdss
GRID BY ra BETWEEN 100 AND 300 STEP 1,
        dec BETWEEN 5 AND 40 STEP 1
HAVING AVG(brightness) > 0.8 AND
       LEN(ra)  = 3 AND
       LEN(dec) = 2
```

Figure 2: An SW query written with the proposed SQL extensions

To address these problems we propose to extend SQL to directly express SW queries. Our extensions are as follows:

- The new `GRID BY` clause for defining the search space. This clause replaces `GROUP BY` (both cannot be used at the same time).
- New functions that can be used for describing windows. Namely, $LB(d_i), UB(d_i)$ to compute the lower and upper boundaries of a window in dimension d_i. Other functions are possible depending on the user's needs.
- New functions for defining shape-based conditions. Namely, $LEN(d_i)$, which is equivalent to $len_{d_i}(w)$. This function is syntactic sugar, since it is possible to compute it by using boundary functions introduced above.

Additionally, we reuse the existing SQL `HAVING` clause to define conditions for the query. Figure 2 shows how Example 1 can be expressed with the proposed extensions.

In the `GRID BY` clause, `BETWEEN` defines the boundaries of the search area for every dimension and `STEP` defines steps of the grid (`ra`, `dec` are attributes of `sdss` and serve as dimensions). The query is processed in the same way as a `GROUP BY` query in SQL, except that instead of groups it works with windows defined via the `GRID BY` clause. `HAVING` has the same meaning — filtering windows that do not satisfy conditions. Since `SELECT` outputs windows, only functions describing a window can be used there: the ones describing the shape and the ones that were used for defining conditions. Similar restrictions are imposed in the SQL standard for `GROUP BY` queries.

4. THE SW FRAMEWORK

4.1 Data-Driven Online Search

We first describe our basic algorithm for executing SW queries with online results. Logically, the search process can be described as follows:

1. Data set S is divided into cells c_i as specified by grid G_S.
2. All possible windows w are enumerated and explored one by one in an arbitrary order.
3. If window w satisfies all conditions (i.e., $\{\forall c \in C : w_c = true\}$), the window belongs to the result.

This suggests a naive algorithm to compute an SW query. The algorithm presented in this section is designed to provide online results in an efficient way. The main idea is to dynamically generate promising candidate windows as the search progresses and explore them in a particular order.

The search space of all possible windows is structured as a graph. First we define relationships between windows. Giving a window w, an *extension* of w is a window w', which is constructed by combining cells of w with a number of adjacent cells from its neighborhood. If w' is extended in a single dimension and direction from w, it is called a *neighbor*. An example can be seen in Figure 3. A two-dimensional search area is divided into four cells, labeled 1 through 4. Window 1|2|3|4 [2] is an extension of window 1, since it is produced by adding adjacent cells 2 through 4. At the same time, 1|2|3|4 is a neighbor of 1|2, since 1|2 is extended only in the vertical dimension and the only direction — "down". The resulting search graph consists of vertices representing windows and edges connecting neighbors.

With the search graph defined, we use a heuristic best-first search approach to traverse it. The heuristic is based on the utility of a window, which will be discussed in more detail in Section 4.2. In a nutshell, utility is a combination of the cost of reading a window from disk and the potential benefit of the window, which is a measure of how likely the window is to satisfy the query conditions. The resulting algorithm is presented as pseudo-code below:

Algorithm 1 Heuristic Online Search Algorithm

```
Input: search space $S$, grid $G_S$, conditions $C$
Output: resulting windows $RES_Q$

procedure HeuristicSearch($S, G_S, C$)
    $PQ \leftarrow \emptyset$, $RES_Q \leftarrow \emptyset$        ▷ PQ — priority queue
    $StartW \leftarrow StartWindows(S, G_S, C)$
    for all $w \in StartW$ do
        $EstimateUtility(w)$
        $insert(PQ, w)$        ▷ utility as priority
    while $\neg empty(PQ)$ do
        $w \leftarrow pop(PQ)$
        $UpdateUtility(w)$
        if $Utility(w) \geq Utility(top(PQ))$ then
            $Read(w)$        ▷ read from disk if needed
            $UpdateResult(RES_Q, C, w)$
            $N \leftarrow GetNeighbors(w, S, G_S)$
            for all $n \in N$ do
                $EstimateUtility(n)$
                $insert(PQ, n)$
        else
            $insert(PQ, w)$
```

The algorithm uses a priority queue to explore candidate windows according to their utilities. The utility is estimated before a window is read from disk via a precomputed sample. Since estimations might improve while new data is read from disk during the search, when the next window is popped from the queue, we update its utility via the $UpdateUtility()$ function. This can be seen as lazy utility update. If the window still has the highest utility, it is explored (i.e., read from disk and checked for satisfying the conditions). Otherwise, it is returned to the queue to be explored later. Additionally, we periodically update the whole queue to avoid stale estimations. In this case only the windows for which new data is available are actually updated.

The procedure begins by determining the initial windows via the $StartWindows()$ function. Since the search space might be large, it is important to aggressively prune windows. Suppose the user specifies a shape-based condition that defines the minimum length n for resulting windows in some dimension (e.g., $len_{d_i}(w) \geq n$). $StartWindows()$ does not generate windows that cannot satisfy this condition, effectively pruning initial parts of the graph. Otherwise, the search starts from cells. A similar check is made in the $GetNeighbors()$ function, which generates all neighbors of the current window. It checks for conditions that specify the maximum length in a dimension (e.g., $len_{d_j}(w) \leq m$) and does not generate neighbors that violate such conditions.

Since the number of windows can be very large, at some point the priority queue may not fit into memory. It is possible to spill the tail of the queue into disk and keep only its head in memory. When a new window has a low utility, it is appended to the tail (in any order). When the head becomes small enough, part of the tail is loaded into memory and priority-sorted, if needed. For efficiency, the tail can be separated into several buckets of different utility ranges where windows inside a bucket have an arbitrary ordering.

While additional pruning, based on other conditions, might further increase the efficiency of the algorithm, it is not safe to apply in general. Since providing approximate results is not an option we consider, it is not possible to discard a window or its extensions solely on the basis of estimations. Content-based conditions must be checked on the exact data. In general, extensions have to be explored as well, since they too can produce valid results. However, in some restricted cases, it is possible to prune them.

One example is the so-called *anti-monotone* constraints [9]. In case of a content-based condition $sum(w) < 10$ and assuming $sum()$ can produce only non-negative values, it is possible to prune all windows that contain the current window w' if $sum(w') \geq 10$, since $sum()$ is monotonic on the size of a window. The length and cardinality of a window are other examples of such functions. Since they are data-independent, the corresponding conditions are always safe to use for pruning, which our algorithm supports. Such anti-monotone pruning, however, would not necessarily decrease the amount of data that has to be read. Windows that just overlap with w' might still be valid candidates for the result.

4.2 Computing Window Utilities

The utility of a window is a combination of its *cost and benefit*. The cost determines how expensive it is to read a window from disk. Since the grid is defined at the logical level without considering the underlying data distribution, some windows may be more expensive to read than others, if the data distribution is skewed. Also, since windows overlap, the cost of a window may decrease during the execution if some of its cells have already been read as parts of other windows. We assume that the system caches objective function values for every cell it reads, so it is not necessary to read a cell multiple times. The cost is computed as follows. Let $|S| = n, |G_S| = m$, where $|S|$ is the total number of objects in the data set and $|G_S|$ is the total number of cells. Let the number of objects belonging to non-cached cells of window w be $|w|_{nc}$. Then the cost C_w of the window is computed as: $C_w = \frac{|w|_{nc}}{n/m} = \frac{|w|_{nc}m}{n}$.

[2] $c_1|\dots|c_k$ labels a window consisting of cells c_1 through c_k

In case data does not have considerable skew, the cost is approximately equal to the number of non-cached cells belonging to the window. However, in general the cost might differ significantly, depending on the skew. To compute the cost accurately, it is necessary to estimate the number of objects in a window. We use a precomputed sample for the initial estimations and update these estimations during the execution as we read data. When reading a window, we not only compute the objective function values, but also the number of objects for every cell. This results in computing an additional aggregate for the same cells, which does not incur any overhead.

The second part of the utility computation is benefit estimation. Here, we determine how "close" a window is to satisfying the user-defined conditions. First, we compute the benefit for every condition independently. The framework currently supports only comparisons as conditions, but the approach can be generalized for other types of predicates. Assume an objective function $f(w)$ and the corresponding condition in the form: $f(w)$ *op* *val*, where *op* is a comparison operator. We assume $f(w)$ can be estimated for window w via the precomputed sample. Let the estimated value be f_w. Then the benefit b_w^f for condition f for window w is computed as follows:

$$b_w^f = \begin{cases} \max\left\{0, 1 - \frac{|f_w - val|}{eps}\right\} & \text{if } f_w \; op \; val = false \\ 1 & \text{if } f_w \; op \; val = true \end{cases}$$

The value *eps* determines the precision of the estimation and is introduced to normalize the benefit value to be between 0 and 1. It often can be determined for a particular function based on the distance of the corresponding attribute values from *val*. For example, if $f = avg(a_i)$, then $\max\{|val - min(a_i)|, |val - max(a_i)|\}$ can serve as *eps*. Alternatively, a value of the magnitude of *val* can be taken initially and then updated as the search progresses.

The total benefit B_w of window w is computed as the minimum of the individual benefits, since a resulting window must satisfy all conditions: $B_w = \min_{f \in C} b_w^f$.

Since $b_w^f \in [0, 1]$, it follows that $B_w \in [0, 1]$. The utility of window w is a combination of the benefit and cost:

$$U_w = sB_w + (1 - s)\left(1 - \min\left\{\frac{C_w}{k}, 1\right\}\right)$$

The cost is divided by k to normalize it to $[0, 1]$. In case the user did not provide any restrictions on the maximum cardinality for windows, k is equal to m. Otherwise, it is equal to the maximum possible cardinality inferred from shape-based conditions. The parameter s ($s \in [0, 1]$) is the *weight* of the benefit. Lowering the value allows exploring cheaper, but promising, windows first, while increasing it prioritizes windows with higher benefits at the expense of the cost. Intuitively, it is better to first explore windows with high benefits and use the cost as a tie-breaker, picking the cheaper window when benefits are close to each other.

We illustrate the algorithm with the search graph from the previous section, presented in Figure 3. The exact parameters for the search area are irrelevant. Assume the whole data set contains $n = 200$ objects and the number of objects in cells 1, 2, 3 and 4 ($m = 4$) is 50, 20, 30 and 100 respectively. We assume that $eps = 10, k = 4$ and $s = \frac{1}{2}$. The only condition is $f(w) > 13$ and the estimated values f_w are specified on top of windows. The numbers to the left (bottom) of windows specify benefits, costs and utilities as b, c and u respectively. Initially, only the values for cells are computed, and the values for other windows are computed when they are explored. The search progresses as follows:

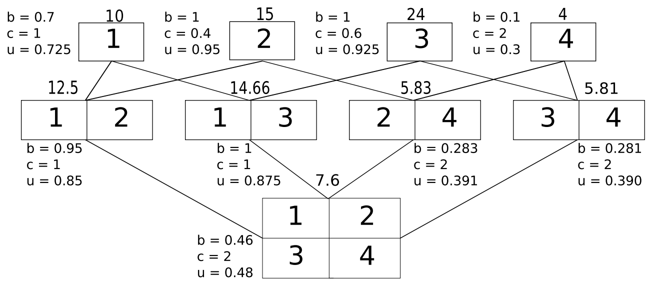

Figure 3: The search graph for an SW query, annotated with *b*enefits, *c*osts, *u*tilities and objective function estimations

1. Since there are no shape-based conditions, the search starts with cells. Initially, $PQ = [2, 3, 1, 4]$, and the algorithm picks cell 2, which is read from disk. It generates two neighbors: $1|2$ and $2|4$. $f(w)$ is estimated from the sample, and the windows are put into the queue. Note that $C_{1|2} = C_1 = 1, C_{2|4} = C_4 = 2$, since cell 2 has already been read.
2. $PQ = [3, 1|2, 1, 2|4, 4]$. The algorithm picks cell 3 and reads it from disk. It generates its neighbors: $1|3$ and $3|4$.
3. $PQ = [1|3, 1|2, 1, 2|4, 3|4, 4]$. The next window to explore is $1|3$. It is processed in the same way and generates the only neighbor possible — $1|2|3|4$.
4. $PQ = [1|2, 1, 1|2|3|4, 2|4, 3|4, 4]$. The search moves to window $1|2$. Since cells 1 and 2 have already been read, the window is not read from disk and explored in memory. Its only extension, $1|2|3|4$, was generated already and is skipped.
5. $PQ = [1, 1|2|3|4, 2|4, 3|4, 4]$. The search then goes through 1, $1|2|3|4$, $2|4$, $3|4$ and, finally, 4 in the order of their utilities. Due to caching, all windows except $1|2|3|4$ are checked in memory, and when $1|2|3|4$ is explored, only cell 4 is read from disk.

At every step, after a window is read from disk, the condition is checked. If the window satisfies the condition, it is output to the user. Otherwise, it is filtered.

4.3 Progress-driven Prefetching

Reconsider the heuristic search algorithm presented in Section 4.1. Every time a window is explored, all of its cells that do not reside in the cache have to be read from disk. If the grid contains a large number of cells, the search process might perform a large number of reads, which might incur considerable overhead. If the data placement on disk does not correspond to the locality implied by windows, reading even a single window might result in touching a large number of pages dispersed throughout the data file. The problem goes beyond disk seeks. If only a small portion of objects from each page belongs to the window, these pages have to be re-read later when other windows touch them. This might create thrashing. An implementation of the algorithm that does not modify the database engine cannot deal with this problem. The problem might be partially remedied by clustering the data, yet there are times when users cannot change the placement of data easily. The ordering might be dictated by other types of queries users run on the same data. Another possible way is to materialize the grid by precomputing all cells. However, this requires that the query parameters (i.e., the search area, the grid and functions) be known in advance. Since exploration can often be an unpredictable ad hoc process, we assume the parameters may vary between queries. In this case, materialization performed for every query will make online answering impossible.

To address this problem, we use an adaptive prefetching strategy. Our framework explores windows in the same way, but it prefetches additional cells with every read. The window is extended to a new window, according to the definition of the extension from Section 4.1. When the size of prefetching increases, intermediate delays might become more pronounced due to the additional data read with every window. At the same time, due to the decreased number of reads, this reduces the overhead and the query completion times. Decreasing the size of prefetching has the opposite effect. While this approach can be seen as offering a trade-off between the online performance and query completion time, we show that, in some cases, prefetching can be beneficial for both, as we demonstrate through experiments.

During the search it is beneficial to dynamically change the size of prefetching. A read can have two outcomes. *Positive* reads result in reading cells that belong to the resulting windows. It might be the window just read or windows overlapping with it. While new results keep coming, the framework prefetches a constant *default* amount, controlled by a parameter. By setting this parameter users can favor a particular side of the trade-off. On the other hand, a *false positive* read does not contribute to the result, reading cells that do not belong to the resulting windows. A false positive can happen for two reasons:

- The remaining data does not contain any more results. To confirm this, the search process still has to finish reading the data. All remaining reads are going to be false positives. In this case the best strategy would be to read it in as few requests as possible.
- Due to sampling errors, utilities might be estimated incorrectly. Since new results are still possible, it is better to continue reading data via select, short requests.

Since it is basically impossible to distinguish between the two cases without seeing all the data, we made the prefetching strategy adapt to the current situation. In this new technique, which we refer to as *progress-driven prefetching*, the size of prefetching increases with every new consecutive false positive read, which addresses the first case. When a positive read is encountered, the size is reset to the default value to switch back at providing online results. The size of prefetching, p, is computed as:

$$p = (1+\alpha)^{\alpha + fp\_reads} - 1$$

$\alpha \geq 0$ is the parameter that controls the default prefetching size. We call it the *aggressiveness* of prefetching. In case $\alpha = 0$, $p = 0$ and no additional data is read. Increasing α results in increasing the default prefetching size, which favors the query completion time. fp_reads is the number of consecutive false positives. When it increases, p increases exponentially. If a new result is discovered, fp_reads is set to 0, and p is automatically reset. This corresponds to the adaptable strategy described above. The exponential increase was chosen so that the size would grow quickly in case no results can be found. This allows us to finish the query faster. We assume that the number of consecutive false positives due to sampling errors is not going to be large, so online performance will not suffer.

Algorithm 2 Prefetch Algorithm

```
Input: window $w$, prefetch size $p$
Output: window to read $w'$

procedure PREFETCH($w, p$)
    $w' \leftarrow w$
    for $i = 1 \rightarrow n$ do          $\triangleright$ n — number of dimensions
        for $dir \in \{left, right\}$ do
            $max \leftarrow C_{w'} + p \prod_{k \neq i} len_{d_k}(w')$
            repeat
                $ext \leftarrow GetNeighbor(w', d_i, dir)$
                if $C_{ext} \leq max$ then
                    $w' \leftarrow ext$
            until $C_{ext} > max$
```

The pseudo-code of Algorithm 2 describes how a window is extended according to the value of p. The size defines a "cost budget" for the extension. This budget is applied as a number of possible extension cells independently for every direction in every dimension. The budget-based approach allows us to address a possible data skew. Since for a fixed dimension a window can be extended in two directions, we denote them as $left$ and $right$.

4.4 Diversifying Results: Exploration vs. Exploitation

Our basic SW strategy is designed for "exploitation" in that it is optimized to produce qualifying SWs as quickly as possible without taking into account what parts of the underlying data space these results may come from. In many data exploration scenarios, however, a user may want to get a quick sense of the overall data space, requiring the underlying algorithm to efficiently "explore" all regions of the data, even though this may slow down query execution.

With the basic algorithm, when a number of resulting windows is output, other promising windows that overlap with them will have reduced costs and higher utilities due to the cached cells. Such windows will be explored first, which might favor results located close to each other. In some cases it might be desirable to jump to another part of the search space that contains promising, but possibly more expensive, windows. This way the user might get a better understanding of the search area and results it contains. We considered two approaches to achieve that.

The first approach is to include a notion of *diversity* of results within the utility computation. We define a *cluster* of results as an MBR (Minimum Bounding Rectangle) containing all resulting windows that overlap each other. Discovering all final clusters faster might give a better understanding of the whole result. When a new window is explored we compute the minimum Euclidean distance $dist$ from the window to the clusters already found. The distance is normalized to $[0, 1]$ and included as a part of the window's benefit: $B'_w = \frac{B_w + dist}{2}$, resulting in the modified utility U'_w. If the window being explored belongs to a cluster, we find the next highest-utility window w' with a non-zero $dist$ and compare utilities U'_w and $U'_{w'}$. If w' has a higher utility, it is explored first. We call this a "jump". With this ap-

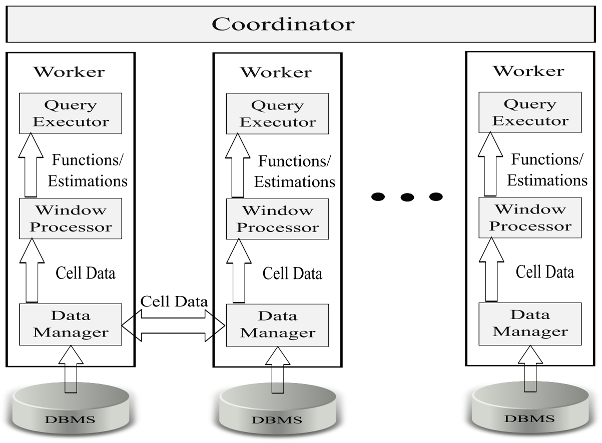

Figure 4: Distributed SW architecture

proach, promising windows might be stifled by consecutive jumps. To avoid this problem, the jumping is turned off at the current step if the last jump resulted in a false positive.

Another approach is to divide the whole search area into sub-areas, according to a user's specification. For example, a time-series data might be divided into years. Each sub-area has its own queue of windows and the search alternates between them at every step. If a window spans multiple sub-areas, it belongs to the sub-area containing its left-most coordinate point, which we call the window's *anchor*. This approach is similar to the online aggregation over a `GROUP BY` query [8], where different groups are explored at the (approximately) same pace. Since some sub-areas might not contain results, the approach may cause large delays. At the same time, it makes the exploration more uniform.

5. ARCHITECTURE AND IMPLEMENTATION

We implemented the framework as a distributed layer on top of PostgreSQL, which is used as a data back-end. The algorithm is contained within a client that interacts with the DBMS. To perform distributed computation, the clients, which we call *workers*, can work in parallel under the supervision of a *coordinator*. The coordinator is responsible for starting workers, collecting all results and presenting them to the user. The search area is partitioned into disjoint sub-areas among the workers. A window belongs to the worker responsible for the sub-area containing the window's *anchor*, its leftmost point. Since some windows span multiple partitions, the worker is responsible for requesting the corresponding cell data from other workers. An overview of the distributed architecture is shown in Figure 4.

The worker consists of the Query Executor, which implements the search algorithm, including various optimizations such as prefetching. The Window Processor is responsible for computing utilities and objective function values (exact and estimated), based on the information about cells provided by the Data Manager. The Data Manager implements all the logic related to reading cells from disk, maintaining additional cell meta-data, and requesting cell data from other workers. This includes:

Caching. The cache contains objective function values for all cells read from disk and requested from other workers. When a new window is explored, only the cells that are not in the cache are read. We assume that the objective values for all cells can fit into memory. This is a fair assumption, since data objects themselves do not have to be cached.

Sample Maintenance. The Data Manager maintains a sample that is used for estimating objective function values and the number of objects for every cell that has not been read from disk. We assume that a precomputed sample is available in the beginning of query execution. The parts of the sample belonging to other partitions are requested from the corresponding workers.

DBMS Interaction and I/O. When a request for a window is made, the Data Manager performs the read via a query to the underlying DBMS. It requests objective function values for non-cached cells belonging to the window in a single query. These values are combined with the cached ones at the Window Processor producing the objective value for the window. Windows consisting entirely of cached cells are processed without the DBMS. We assume that the value of an objective function for a window can be combined from the values of cells the window consists of. All common aggregates (e.g., $min()$, $sum()$, $avg()$, etc.) support this property.

Remote Requests. When a window spans multiple partitions, the Data Manager requests objective values for the cells belonging to other partitions from other workers. If a remote worker does not have the data in cache, it delays the request until the data becomes available. Eventually it is going to read all its local data and, thus, will be able to answer all requests. After every disk read, the worker checks if it can fulfill more requests. If so, it returns the data to the requester. At the same time, the requester continues to explore other windows. When the remote data comes, the corresponding windows are computed and reinserted into the queue. The only way a worker may block is when it has finished exploring its entire sub-area and is waiting for remote data. Thus, the total query time is essentially dominated by the total disk time of the slowest worker.

6. EXPERIMENTAL EVALUATION

For the experiments we used the prototype implementation described in Section 5, written in C++. Single-node experiments were performed on a Linux machine (kernel 3.8) with an Intel Q6600 CPU, 4GB of memory and WD 750GB HDD. All experiments with the distributed version presented in Section 6.7 were performed using EBS-optimized m1.large Amazon EC2 instances (Amazon Linux, kernel 3.4).

Data Sets For the experiments, we used three synthetic data sets and SDSS [1]. Each synthetic data set was generated according to a predefined grid. The number of tuples within each cell was generated using a normal distribution with a fixed expectation. Each synthetic data set contains eight clusters of tuples, where a cluster is defined as a union of non-empty adjacent cells. We generated three queries, one for each data set, which select four clusters. These target clusters differ in their distance from each other, which we call *spread*. Essentially, each data set contains the same clusters (and tuples), although their coordinates differ. Thus, each query has the same conditions, which allowed us to measure the effect of the spread in a clean way. The parameters of the query are: $S = [0, 1000000) \times [0, 1000000), s_1 = s_2 = 10000, card() \in (5, 10), avg() \in (20, 30)$.

For the SDSS data set, the situation is different. It is hard to ensure the same cleanliness of the experiment when dealing with different spreads of the result. We thus chose three queries with (approximately) the same selectivity and different spreads. However, in this case the conditions for each query differ and the search process explores different candidate windows in every case. The parameters of the queries are: $S = [113, 229) \times [8, 34), s_1 = s_2 = 0.5, card() \in (10, 20)/(5, 10)/(15, 20), avg(\sqrt{rowv^2 + colv^2}) \in (95, 96)/(100, 101)/(181, 182)$, where ra, dec (from the relational SDSS) are dimensions, / divides high, medium and low spread respectively and $rowv, colv$ are velocity attributes.

Each of the data sets takes approximately 35GB of space, as reported by the DBMS. PostgreSQL's shared buffer size was set at 2GB. Computing an objective function for a window is transformed into a SQL prepared statement call. The statement is basically a range query, defining the window, with a `GROUP BY` clause to compute individual cells. For the efficient execution of range queries, we created a GiST index for each data set [3] The size of each index is approximately 1GB. Each query results in a bitmap index scan, reading the data pages determined during the scan and an aggregation to compute the objective function. PostgreSQL performed all aggregates in memory, without spilling to disk.

Data Placement Alternatives As described in Section 4.3, the placement of data on disk has a profound impact on the performance of the algorithm. In the experimental evaluation we considered three options:

- Ordering tuples by one of the coordinates (e.g., order by the x-axis). In this case windows generally contain tuples heavily dispersed around the data file. We denote this option as "Synth/SDSS-*axis*" in text (e.g., "SDSS-ra").
- Clustering the tuples according to the GiST index. This reduces the dispersion of tuples, but since R-trees do not guarantee efficient ordering, there still might be considerable overhead for each range query. This option is denoted with suffix "-ind".
- Clustering the tuples by their coordinates in the data file. One common way to do this is to use a space-filling curve. Since we used a Hilbert curve, we denoted this option with suffix "-H". Another way is to cluster together tuples from the same part of the search area. For example, tuples from each of the eight synthetic clusters are placed together on disk, but no locality is enforced between the clusters. This is a convenient option in case more data is added and the search area is extended, since it does not destroy the Hilbert ordering. We define this option with suffix "-clust".

Stratified Sampling We used a stratified sampling approach to estimate utilities. Assuming the total number of cells is m and the total sample budget is n tuples, each cell is sampled with $t = \frac{n}{m}$ tuples. If a cell contains fewer tuples than t, all its tuples are included in the sample and the remaining cell budget is distributed among other cells. The idea is similar to the idea of *fundamental regions* [4] and congressional sampling [2]. A cell is a logical choice for a fundamental region. Each cell is independently sampled with SRS (Simple Random Sampling). Since cells effectively have different sampling ratios, we store the ratio with each sampled tuple to correctly estimate the required values, which is the common way to do this. All experiments reported were performed using a 1% stratified sample.

[3]In PostgreSQL, GiST indexes are used instead of R-trees, which would be a logical choice for SW queries.

Table 1: Query completion times for different aggressiveness values (in seconds)

Dataset	No pref	$\alpha = 0.5$	$\alpha = 1.0$	$\alpha = 2.0$
Synth-x	28,206.84	13,521.55	8,602.45	6,957.33
Synth-clust	1,123.12	859.08	886.01	817.59
SDSS-dec	26,725.05	4,542.17	3,145.15	2,109.76
SDSS-clust	1,510.59	1,145.37	1,130	1,158.29

6.1 Query Completion Times

In this experiment we studied the effect of the data placement and prefetching on the query completion time. Table 1 provides the results for the high spread query. Other queries exhibited the same trend. α denotes the aggressiveness of prefetching, where "No pref" means no prefetching.

To establish a baseline for the comparison, we ran a corresponding SQL query (as described in Section 3) in PostgreSQL and measured the total and I/O (disk) time. Due to the nature of the SQL query, PostgreSQL did a single read of the data file, and then aggregated and processed all windows in memory. For the synthetic data set, the query resulted in $\underline{1{,}457.84s}$ total and $\underline{677.94s}$ I/O time. For SDSS the query resulted in $\underline{3{,}589.93s}$ total and $\underline{849.70s}$ I/O time. The difference between the synthetic and SDSS times was due to small differences in the size of the data sets and the parameters of the queries (SDSS selected more windows, which resulted in more CPU overhead for PostgreSQL).

As Table 1 shows, in case when data is physically clustered on disk (-clust), the framework is able to outperform PostgreSQL even without using prefetching. This is due to a very small CPU overhead for the framework. When using prefetching, the difference becomes even more pronounced. In the "SDSS-clust" case, using $\alpha = 2.0$ resulted in 30% less completion time. It is important to mention that the framework starts outputting results from the beginning, while the SQL approach outputs all results only at the end.

In case the data is physically dispersed on disk (-x and -dec ordering), prefetching allowed us to reduce the completion time significantly, i.e., by an order of magnitude. In the case of SDSS, the framework eventually started outperforming PostgreSQL, while for the synthetic data a considerable overhead remained. In general, the performance improvement depends on the properties of the data set, such as the degree of skew or the number of clusters. Despite the remaining overhead for the synthetic data, we believe the framework remains very useful even in such cases, since it starts outputting results quickly.

6.2 Online Performance

This experiment studies the effect of prefetching on online performance (i.e., delays with which results are output). As the previous experiment showed, increasing the prefetching size reduces the query completion time. At the same time, it should increase delays for results since the amount of data read with each window increases. To present the experiment more clearly, instead of showing individual result times, we show times to deliver a portion of the entire result (a percentage of the total number of answers). The time to find all the answers (i.e., 100% in the figures) and the completion time of the query generally differ. The former might be smaller, since even when all answers are found, the search process has to read the remaining data to confirm this. This means users can be sure the result is final only when the query finishes

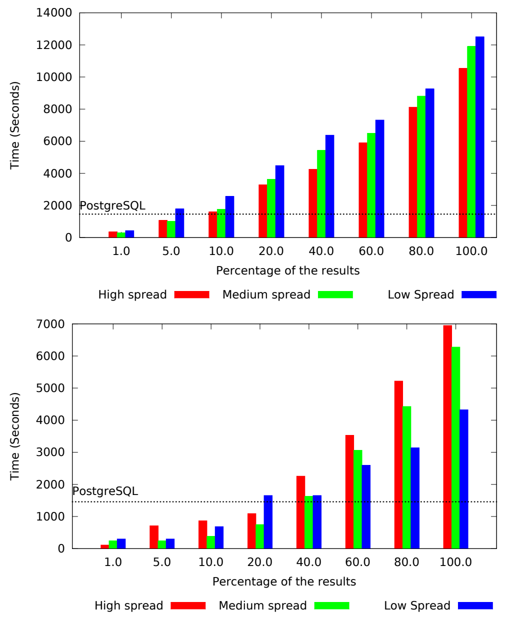

Figure 5: Online performance of the synthetic queries (Synth-x, top: $\alpha = 0.5$, bottom: $\alpha = 2.0$)

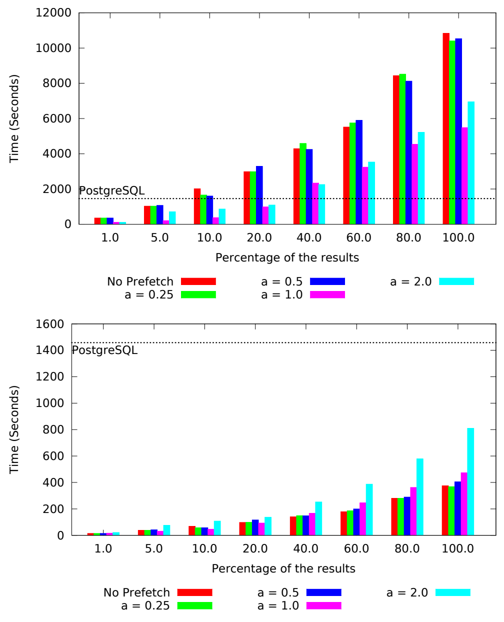

Figure 6: Online performance of the high-spread synthetic query (top: Synth-x, bottom: Synth-clust)

completely. We provide the PostgreSQL baseline, explained in the previous section, as a dotted line in the figures.

Figure 5 shows the online performance of all three queries with different spreads for the synthetic data set (sorted by the axis x). All queries behaved approximately the same, since at every step the algorithm considers windows from the whole search space. The differences were due to the physical placement of data. For the case of $\alpha = 2.0$ the final result was found faster for the low spread query. Since in this case clusters were situated close to each other, prefetching large amounts of data around one cluster allowed the algorithm to "touch" another, nearby, cluster as well. Other orderings of the data set resulted in the same behavior.

Figure 6 shows the online performance of the high-spread query for the "Synth-x" and "Synth-clust". For "Synth-x" larger aggressiveness values resulted in much better online performance during the whole execution, although $\alpha = 2.0$ created longer delays at some points. The situation changes with the beneficial clustered ordering. While values up to $\alpha = 1.0$ behaved approximately the same, $\alpha = 2.0$ created much longer delays. This exposes a trade-off between the query completion time savings coming from prefetching data and the delays for online results. Figure 7, which shows the results for the high spread SDSS query, demonstrates the same trend. If the user is not aware of the ordering, $\alpha = 1.0$ might be considered a "safe" value on average, which both provides considerable savings and does not cause large initial delays. For advanced usage, the aggressiveness should be made available to users to control. If the user is satisfied with the current online results, she can increase the value to finish the query faster. If the ordering is not beneficial (e.g., axis-based), the value should not be set to less than $\alpha = 1.0$.

Other queries exhibited the same trend and are not show. We do not show the results similar to Figure 5 for SDSS. Since different spread queries for SDSS differ in the number of results and query parameters, the algorithm behaved different each time (e.g., considered different candidate windows and used different sizes for prefetching), which made the direct comparison of different queries meaningless.

We should mention that first results are output in 10 seconds in most cases and in 80 seconds in worst cases. This fulfills one of the most important goals: start delivering online results fast.

6.3 Physical Ordering of Data

Here we studied the effect of the physical data placement, described in Section 4.3, in more detail. Table 2 presents statistics of data file reads for three different ordering options described at the beginning of Section 6. We ran a single query (one per data set) and used `systemtap` probes in PostgreSQL to collect the statistics. In the table, "Total" refers to the total time of reading all blocks, without considering the time to process the tuples.

Table 2: Disk statistics for the synthetic dataset

Data set	Total (s)	Mean/Dev read (ms)	Reads (blks)	Re-reads (blks)
Synth-x	24,987	2.4/2.5	10,476,601	6,477,523
Synth-ind	3,053	0.7/1.7	4,217,096	218,018
Synth-clust	738	0.2/0.8	4,001,263	2,185
Synth-H	747	0.2/0.8	4,000,592	1,514

When the physical ordering did not work well with range queries (i.e., -x), the DBMS effectively had to read the same data file more than twice (see the "Re-reads" column), which supports the thrashing claim made in Section 4.3. Moreover, since each range query resulted in multiple dispersed reads, the time of a single read grew considerably and became more unpredictable (see the "Mean/Dev" column), because of seeks. SDSS showed the same results.

6.4 Prefetching Strategies

The size of prefetching depends on the number of consecutive false positive reads. We call such a prefetching strat-

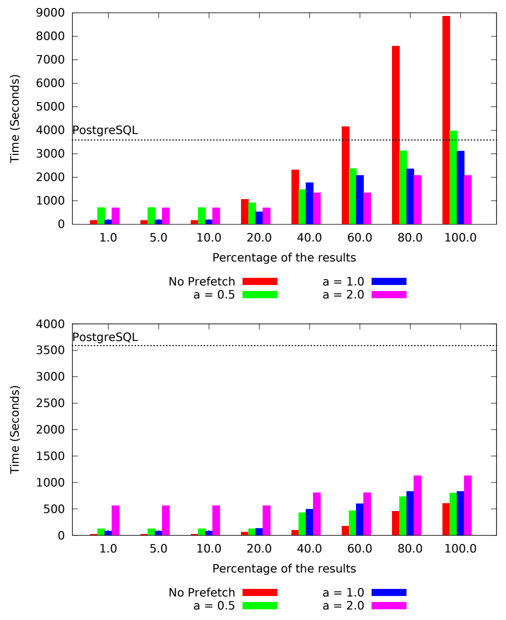

Figure 7: Online performance of the high-spread SDSS query (top: SDSS-dec, bottom: SDSS-clust)

egy “dynamic”. However, even in the absence of false positives, there is a default amount of prefetching at every step, namely $(1+\alpha)^\alpha - 1$. We call the strategy where the aggressiveness does not depend on the number of false positives, “static”. The experiment presented in this section studies the difference in online performance between the two.

Figure 8 presents the online performance of both strategies for two different SDSS queries (dec-axis ordering). Other queries showed the same trend. “Total time” bars show the completion time (“100%” bars show the time to find all results). For the same aggressiveness level, the dynamic strategy was better in both online and total performance. When $\alpha = 2.0$, the difference seems less pronounced, although for the “Total time” it reached about 900 seconds for the low-spread and 700 seconds for the medium-spread query. It is evident that taking false positives into account makes prefetching more efficient.

6.5 Diversity of Results

The experiment presented in this section studies different diversity strategies from Section 4.4. The resulting times to discover a number of clusters for the medium-spread SDSS query (clustered ordering, no prefetching) are presented in Table 3. By discovering a cluster we mean finding at least one window belonging to the cluster. Since clusters are unknown in advance, we provide the times after analyzing the final result. “Original” refers to the basic algorithm of Section 4.1. “Utility jumps” refers to the approach with modifying the utility, described in Section 4.4. “Dist jumps” examines the best k candidates (from the priority queue) at each step and chooses the furthest from the current clusters. “X static” refers to another strategy from Section 4.4, where the search area is evenly divided into X sub-areas.

Initially, the original and the two jump strategies performed similarly. However, the time to discover all clusters was reduced by 40% comparing with the original approach. While “Dist jumps” performed slightly better, it requires a

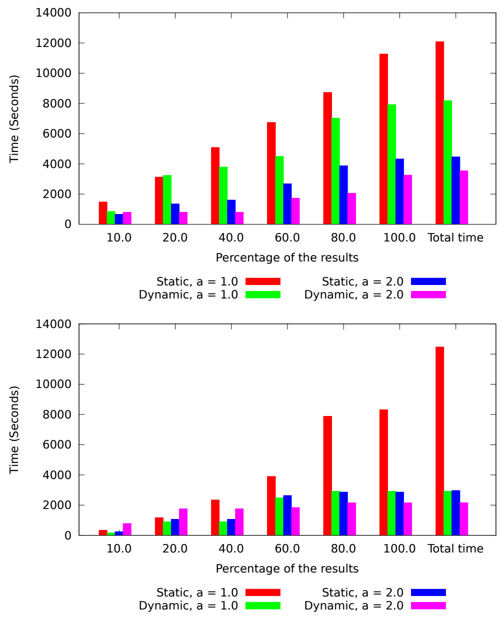

Figure 8: Online performance of the static and dynamic prefetching (SDSS-dec, top: low-spread SDSS query, bottom: medium-spread SDSS query)

Table 3: Times to discover clusters for the medium-spread SDSS query (SDSS-clust, no pref, in seconds)

Strategy	First cluster	5 clusters	All clusters
Original	12.55	56.06	223.53
Dist jumps	11.41	56.85	158.03
Utility jumps	11.43	54.36	171
4 static	19.78	56.40	674.19
9 static	43.13	122.90	1132.10
16 static	33.58	154.85	825.58

parameter — the number of candidates to consider for a jump. On the other hand, “Utility jumps” does not require any parameters. The results show that the static strategies might perform significantly worse. While for the jump strategies the online performance remained approximately the same, static strategies resulted in much larger delays for online results, since they forced uniform exploration across its groups. Such uniform exploration, while serving a different purpose, might still be beneficial for discovering clusters in some cases. When we ran the same experiment for the low-spread SDSS query, static strategies reduced the time to find all clusters from 1,219 seconds to 130/339/694 seconds for the 4/9/16 static strategy correspondingly. The jump strategies did not help at all. This was due to poor window estimations, which are essential to make effective jumps.

6.6 Impact of Estimation Errors

Since the search process is guided by sampling-based estimations, the estimation quality plays an important role on performance. We measured the online performance with an “ideal” 100% sample and then started introducing Gaussian noise into it. Figure 9 presents online performance results for medium-spread queries for both data sets (clustered ordering, no prefetching). The noise percentage equals to the mean of a Gaussian distribution (“No noise” corresponds to

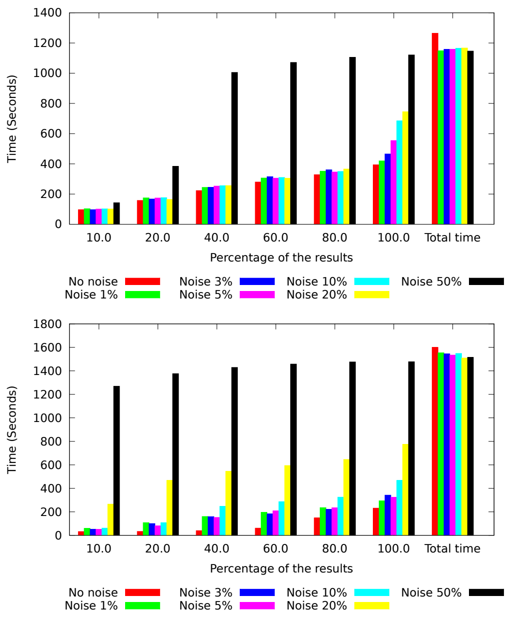

Figure 9: Online performance of different noise levels (SDSS-clust, no pref, top: medium-spread synthetic query, bottom: medium-spread SDSS query)

the ideal sample). For every window, the estimated value of the objective function v became $v \times (1 \pm \frac{n}{100})$, where n was generated by the distribution. The standard deviation was fixed at 5.0. Other queries showed the same trend.

Small noise levels did not introduce any significant difference initially. While the number of false positives was large, the algorithm was able to "stumble" upon results, when reading nearby windows, since the number of undiscovered windows was large at the beginning. Later during the execution, when the number of undiscovered windows diminished, even small noise levels ($\geq 10\%$) started reducing the online performance. For example, for SDSS at the 40% percentage mark the 10% noise performed more than 200 seconds worse when comparing with the ideal sample, which is significant for this scale. It is important to mention that the target interval for the objective function was much tighter for the SDSS query. This explains why the SDSS query performed much worse in the case of 20% noise from the beginning, while for the synthetic query the same was true for a higher, 50%, noise level. Since the interval for the synthetic query was larger, estimations for some resulting windows remained within the interval when the noise level was small.

6.7 Distributed Processing

We now present experimental results for the same queries over distributed data. When a search area is partitioned among the workers, there is flexibility in partitioning the data itself, which we studied along three cases. In the first one, the data partitioning corresponds to area partitioning, which requires some workers to perform remote data requests. We call this *no-overlap*. Another case, *full-overlap*, involves creating overlapping partitions so that workers have all data available locally and no remote requests are made. This case is possible only if shape-based conditions (e.g., the cardinality of windows) are known in advance or if the data is fully replicated. Otherwise, the overlap will be partial and workers will be making remote requests, albeit a smaller number of them. We call this third case *part-overlap*.

Table 4: Results for the distributed synthetic high-spread query (Synth-clust, $\alpha = 1.0$)

Nodes, Overlap	First result	All results	Total time
1 node, no	6	820	1820
2 nodes, no	6	470	1050
4 nodes, no	5	360	580
8 nodes, no	7	200	350
1 node, full	6	820	1820
2 nodes, full	6	490	1250
4 nodes, full	5	350	790
8 nodes, full	7	255	650
8 nodes, part	7	300	540

Table 4 presents the results for the synthetic high-spread query. All times are in seconds and rounded to the nearest integer. Since the I/O performance of Amazon EBS may vary, the number of runs was at least 10. For the no-overlap case, the reason behind the sub-linear scalability is primarily the skew, since the distribution of results is uneven, especially for the case of 4 and 8 nodes, which may likely be common in practice. Moreover, the pattern of reads heavily depends on a worker's area, which creates another difference in result times. As for the total time, we tried to make the partitioning as even as possible, so that each worker has approximately the same amount of data. However, since partitions must be aligned with cells, this is not completely possible. Total disk times differed for up to 100 seconds between workers. Supporting the claim of Section 5, the total time was dominated by the slowest worker's total disk time. The additional overhead was kept under 10 seconds.

The full-overlap case performed worse than the no-overlap case in general. For the total times, this can be explained by the additional disk reads overhead. The overlapping parts were read more than once independently by workers that needed them, so some workers had more data to read. In the no-overlap case, such data was read on a single node and served to other nodes from cache. Result times did not become better, since the number of resulting windows spanning the overlap was small. Moreover, these windows had to be read from disk, which introduced delays. One could argue that reading all data locally might reduce delays when the number of resulting windows spanning the overlap is large. However, there are times when such promising data would be read by other workers sooner and served faster during remote requests. Overall, the full-overlap case does not consistently offer improvements over the no-overlap case.

For the part-overlap case, we present only the result for 8 nodes, since it is consistent with the same trend. The total time is between the times for other cases, which was expected. The time to get all results was worse, due to a different read pattern: the required remote data came in the middle of a large local read and the corresponding windows were explored later. The prefetching pattern was different as well.

Since the total completion time is determined by the total disk time, we performed another experiment where we varied the amount of data read by each worker. With increasing the size skew the total time grew to 390 and 455 seconds for the no-overlap case of 8 nodes. This shows the need to carefully balance workers, which is common for any distributed algorithm. This can be done before running the queries or by automatically assigning sub-areas to workers while estimating their data sizes from the sample.

7. RELATED WORK

In OLAP cubes [6], users start exploring the data by pre-aggregating it (e.g., summary sales for each year) and then can *drill-down* into interesting parts (e.g., a particular year) or *roll-up* by aggregating further. However, such methods do not allow complex exploration via "rich" windows, as in our framework, since they are essentially `GROUP BY`s in nature. Thus, the applicability of the OLAP cubing techniques to our model is limited.

Online aggregation [8, 7] strives to bring approximate results to the user while minimizing the overhead. The approximate query result is updated as the computation evolves. While SW too aims at quickly providing online results, all results are guaranteed to be exact. Moreover, SW works with windows instead of individual tuples; this difference, as we argued, limits the applicability of existing online aggregation techniques.

Online results have been studied for more complex queries, e.g., for skyline-over-join [10]. In some cases it is possible to carefully examine input and output spaces of the join operator to determine input tuple ranges that might produce dominating results needed for the skyline. Such ranges are processed first. In this case online results are guaranteed to be exact. In our case, while the input space is determined by the dimensional attributes, the output depends on a function of the *measurement* attributes. Thus, the output space cannot be easily examined to determine candidate windows.

Fagin's algorithm [5] can be used to select top-k objects as ranked by an aggregate function of their interesting properties. In our case we select candidates by using the cost and benefit as the properties and the sum as the function. Since our framework explores all possible windows (subject to pruning), applying Fagin's algorithm for finding candidate windows would incur more overhead than a priority queue, especially since we have to compute the score function for every window in any case.

While there are several recent systems that address various big data challenges, many of these systems (e.g., SciBORQ [11], BlinkDB [3]) use approximate query answering via sampling. In contrast, we use sampling to steer the search for qualifying windows, providing exact results. Also, these systems do not support window-based data exploration, thus addressing fundamentally different problems.

8. CONCLUSIONS AND FUTURE WORK

Easy-to-use, interactive approaches for human-in-the-loop exploratory analysis of data at scale still remain as an open research area. We presented one such approach, called Semantic Windows (SW), which allows users to conveniently perform structured search via shape and content constraints over a multi-dimensional data space.

Our framework uses a data-driven search algorithm coupled with a variety of complementary techniques, including stratified sampling, adaptive prefetching and sophisticated data placement, to search the underlying data space quickly while providing online results. We described a prototype implementation of the framework as a distributed layer on top of PostgreSQL and conducted an experimental evaluation with real and artificial data to study the impact of various design and algorithmic decisions. The results showed that SW significantly improves online performance without sacrificing query completion time, relative to representative state of the art solutions.

There are several fertile directions for future research. Currently SW supports algebraic comparisons for conditions. We also hear the need to support functions involving multiple windows (e.g., distance, similarity), which would enable operations such as clustering. We also would like to support *optimization queries* that involve min/max functions, e.g., "search for windows with the maximum brightness". In this case, it is generally more difficult to present useful online feedback to the user, since the optimality has to be validated across all windows, making approximate answering with provable errors bounds a promising approach.

We are working towards a general interactive data exploration tool that offers rich, ad hoc search and mining capabilities expressed through a seamless combination of constraints and queries. To this end, we have started to integrate constraint-programming and DBMS techniques under a single platform.

9. ACKNOWLEDGEMENTS

This research was supported in part by the Intel Science and Technology Center for Big Data and NSF IIS-1111423.

10. REFERENCES

[1] The sloan digital sky survey (sdss). http://www.sdss.org/.
[2] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, pages 487–498, 2000.
[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys '13*, pages 29–42, 2013.
[4] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *ACM TODS*, 32(2), June 2007.
[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.
[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.
[7] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28(2):287–298, June 1999.
[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.
[9] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD*, pages 13–24, 1998.
[10] V. Raghavan and E. A. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In *ICDE*, pages 733–744, 2010.
[11] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, pages 296–301, 2011.