Towards Integrated and Efficient Scientific Sensor Data Processing: A Database Approach

Ji Wu
School of Computing
National University of Singapore
wuji@comp.nus.edu.sg

Yongluan Zhou
Dept. of Mathematics & Computer Science
University of Southern Denmark
zhou@imada.sdu.dk

Karl Aberer
School of Computer and Communication Sciences
EPFL, Switzerland
karl.aberer@epfl.ch

Kian-Lee Tan
School of Computing
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

In this work, we focus on managing scientific environmental data, which are measurement readings collected from wireless sensors. In environmental science applications, raw sensor data often need to be validated, interpolated, aligned and aggregated before being used to construct meaningful result sets. Due to the lack of a system that integrates all the necessary processing steps, scientists often resort to multiple tools to manage and process the data, which can severely affect the efficiency of their work. In this paper, we propose a new data processing framework, *HyperGrid*, to address the problem. HyperGrid adopts a generic data model and a generic query processing and optimization framework. It offers an integrated environment to store, query, analyze and visualize scientific datasets. The experiments on real query set and data set show that the framework not only introduces little processing overhead, but also provides abundant opportunities to optimize the processing cost and thus significantly enhances the processing efficiency.

1. INTRODUCTION

Environmental monitoring data collected from wireless sensors typically need to be further processed before being utilized for scientific research. This is because raw sensor data are noisy and incomplete, and hence need to be cleaned. More importantly, there is a mismatch between what scientists desire from the data and what raw sensor data can offer.

Unlike traditional DBMS where users ask for information that can be directly looked up from the database tables, scientific queries are more analytical. The raw input data have to be interpreted with mathematical or geostatistical models provided by the users before they can be used to compute the user queries. We refer to such a step as "data preparation". Note that such data preparation is not a one time job. It is required to be adapted based on the requirements of the user queries. Also, scientists would often try to interpret data with different models to see how the query output would be like. The traditional Clean-Store-Query paradigm cannot be applied here. Instead, it is desirable that the data preparation phase be run at query time.

As there is a lack of a general framework to embrace all the necessary data processing, scientists often use diverse customized codes and various tools for different processing tasks. As such, the whole processing procedure is usually conducted in a number of separated steps. As we will see later, such an approach cannot exploit the opportunities of optimizations across multiple steps and hence prohibits efficient scientific query processing. Furthermore, the lack of a generic processing framework also prohibits the application of generic optimization techniques. Only ad-hoc optimizations written by customized codes are possible. Finally, scientific data processing often involves visualization products and progressive visualization is a desirable feature for many science applications. However, the multi-step processing approach limits the extent of progressive computation and visualization that can be exploited.

As one of our efforts to address the problem, this paper presents an integrated and easy-to-use data processing system for environmental scientists to help their routine tasks of sensor data manipulation. Our work is inspired by the success of the relational DBMS technology which provides an integrated and efficient business data management platform by a generic data model and a generic query processing and optimization framework. Analogously, we propose a new scientific data processing framework, called *HyperGrid*, which comprises a new data model, a query processing framework as well as several generic optimization techniques.

Our context for studying scientific data processing is the SensorScope project [2], a Wireless Sensor Network (WSN) which produces spatial and temporal measures for ecological and environmental monitoring. The system consists of multiple solar-powered sensing stations that measure key environmental data such as air temperature, humidity, solar radiation, wind speed and direction. These sensing stations periodically sample their sensors and transmit the readings through wireless channels to the central server. Scientists can then retrieve the data through the central server in real time. Because these data are still in the rudimentary form, a series of transformations have to be performed before they are ready for scientific research.

The *HyperGrid* system is designed and tailored to such scientific applications. It offers an integrated environment for managing scientific sensor data. The logical abstraction provided by *HyperGrid*

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

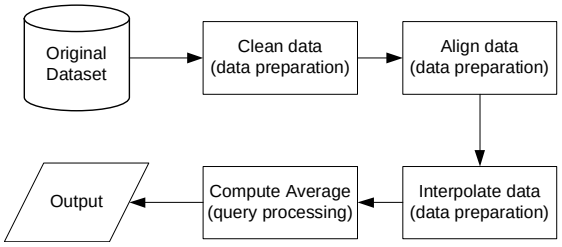

Figure 1: Work flow of Example 2.1

significantly saves users' efforts from handling low level data operations such as array manipulations and coordinates transformations. Furthermore, we show that such an integrated framework offers abundant opportunities for query optimization. Based on the users' requirements, we have implemented several optimization techniques in *HyperGrid*. And our experimental study has demonstrated the effectiveness of these techniques in boosting the performance for scientific query processing.

Before embarking on the details of the system, we first give motivating examples in the next section which features two representative queries issued by scientists and explain why processing these queries are poorly supported by the existing techniques.

2. MOTIVATING EXAMPLES

We show two concrete examples in this section. The first one illustrates a routine query which computes some statistical information about the dataset. The second example depicts a scenario where information is queried for the purpose of data exploration.

2.1 Scenario one

Consider the following query that a scientist typically issues:

EXAMPLE 2.1. *Return the average ambient temperature over the period from 2007-10-01 00:00 to 2007-10-04 00:00 for the region [45°52′1″N, 45°52′23″N] in latitude and [7°10′37″E, 7°10′59″E] in longitude on a $1'' \times 1''$ grid.*

What the scientist has available is a data file with raw temperature readings collected from sensors. Before the data can be used to answer the query, they need to go through several preprocessing steps. Firstly, the original dataset needs to be cleaned. Corrupted points are removed or replaced. Secondly, because sensor data are not available at all the locations specified in the query, the original data need to be interpolated over the geographical space. However, in order to do this, data values have to be first aligned on all dimensions except for those involved in the interpolation. In this example, data need to be aligned over the time dimension before being interpolated on the spatial dimensions (which means the set of data points involved in the same interpolation computation must either have the same temporal value or fall into the same interval). Methods to align the data include to randomly pick one representative data reading (i.e. sampling) for each aligned interval or to take an aggregation over the aligned interval. Once this is done, the spatial interpolation can be performed with the granularity $1'' \times 1''$ for each aligned time slice. Only at this stage is the curated dataset ready to answer the query, which is a simple average aggregation over time from 2007-10-01 00:00 to 2007-10-04 00:00 for the region between [45°52′1″N, 45°52′23″N] and [7°10′37″E, 7°10′59″E]. A flowchart describing the steps of producing the output is given in Figure 1.

It is interesting to note that most of the efforts are actually spent on preparing the data rather than answering the query. This is indeed quite common in scientific data processing. In fact a complete query specification should also include details on how the data are prepared because scientists can have various ways to preprocess the data and their choice of procedure directly influences the query results. There is, however, no standard formula on how the data should be prepared towards a given type of query. Parameters such as the size of the interval in data alignment are determined by knowledge experts and are subject to change. Unfortunately, relational DBMS cannot directly support many data preparation operations with configurable parameters. Although array database is able to achieve this, the separation between array elements and their spatio-temporal context makes the process tedious, especially when operations involve external models. Take interpolation as an example. Kriging as a popular geostatistical technique is often used for interpolation. It requires a variogram model to describe the degree of spatial dependence for a given region. Because the spatial coordinates of the data points in an array database cannot be directly obtained, such spatial dependence hence requires extra effort to derive by stitching the data points with the variogram model. Other operations involving external physical models also encounter the similar problem. In short, a pure array-based implementation is deemed inadequate for supporting such scientific queries.

2.2 Scenario two

The second example demonstrates a scenario that occurs frequently during data exploration. In this case, scientists want to study the factors that impact the solar radiation for some region. They hypothesize that the rainfall rate may be the major influencing factor, which can be negatively correlated to the solar radiation value. They want to verify their hypothesis through the following query:

EXAMPLE 2.2. *Compute the hourly average rainfall rate and hourly average solar radiation over the period τ at some point A. Display the results by plotting a diagram for each type of the measurement.*

To focus on the essence, we omit the details of the time and the location in this query. The query itself is not much different from the previous example except now the result needs to be displayed for visualization. In a data exploration scenario like this, scientists often do not need an accurate quantification as the answer, instead they prefer the results to be plotted so that they can see intuitively whether the outputs are as expected in general. A similar scenario would also emerge when scientists want to study a new physical model. In that case, a bunch of similar queries, differed only by the values of the model parameters under investigation, are executed. Scientists want to gain a quick understanding of how these parameters influence the behavior of the model. To achieve this, the processing engine must facilitate results to be generated in a progressive way to cater for the interactive visual exploration. We will return to this topic when optimization techniques are discussed in Section 6.

3. DATA MODEL

One important characteristic of environmental monitoring data is that they can be uniquely identified by the spatial and temporal attributes corresponding to where and when they are measured. Spatial and temporal attributes are not only used for identification purpose, more importantly, they are treated as the intrinsic properties of the associated measurements, through which scientists can reason about the meaning of the measured data as well as the physical phenomena implied by them.

Conventionally, array is chosen as the basic data model to store scientific measurement data. Structural regularity and concise representation make array-based model suitable for managing data involving complex computations. However, without external mapping spatio-temporal associations with the measurement data are lost with the array representation. This severely hinders efficient scientific sensor data manipulation. To this end, we extend logical array data model and propose a new *HyperGrid* data model. Scientists have been using grid (implemented mainly through array) to model and organize their data. Experience shows that grids work well for typical applications, provided input can be readily fit into the grid points. Data abstracted from grids can be ordered, filtered or aggregated according to the given criteria without much difficulty. Our proposed *HyperGrid* model is based on grid structure which can accommodate both rudimental sensor readings and high level data abstraction under the same framework.

Essentially, a HyperGrid can be seen as a collection of overlay grid structures built on top of a scientific dataset. Each overlay grid is called a *perspective*. Conceptually, a perspective is analogous to a view in traditional DBMS. However, we deliberately use a different term here to distinguish it from a database view[1]. All *perspectives* in a HyperGrid are derived from a single *base perspective*, which is a special *perspective* that links raw data readings with a grid construct. The *base perspective*, or simply *base*, has a predetermined data structure which sets the coordinates and dimensions of the HyperGrid in a multidimensional space. The grid granularity of a *base* is fixed in line with the resolutions of the measuring devices for the corresponding spatial or temporal dimensions. This implies any generated spatio-temporal coordinates can be precisely captured in *base*, ensuring lossless mappings from sensor readings to the corresponding grid points.

On top of the *base*, a set of *perspectives* can be defined by the user or inferred by the system to reflect the user's views over the data. These *perspectives* typically have a coarser grid granularity than the *base*. There is no limit on the number of *perspectives* in a given HyperGrid. Users can create as many *perspectives* as they want for their purposes.

Traditional grid structure only models objects in spatial domain. In *HyperGrid*, each *perspective* also includes time as an additional dimension in the grid space. From a data model point of view, temporal dimension is treated no differently from a spatial dimension. However, scientific queries over temporal space are more involved in query semantic especially for aggregations. We will discuss these issues later in details.

Like traditional grid, each *perspective* is composed of two parts: a grid topology and data values associated with it. The topology refers to the layout of a grid. Essentially it defines how data are grouped along each spatial and temporal dimension. The grid consists of cells that are regularly placed according to the topology definition. Each cell can be seen as an abstract object that represents certain spatial and temporal span. It is important to note each cell is identified by the spatial and temporal coordinate of its "lower-left" corner (imagine cell as a rectangular or an orthotope in a multidimensional space), rather than their relative position index as in the array-based model. This is a significant distinction between array and perspective. The coordinate not only uniquely identifies each cell within a perspective, more importantly, it associates cells among different perspectives through their spatio-temporal context. As we shall see, this can play a very important role in scientific computations.

A *perspective* is only instantiated when the grid topology is bound with data. However, in what follows, we abuse the notation P_i to denote both a *perspective* and its grid topology when there is no ambiguity. Hence, the data value associated with a cell $e \in P_i$ can be represented by $P_i[e]$. There are various types of data values depending on different dimension aggregations they entail. As an example, data values for hourly average temperatures at a given point in a geographical space is a 1-D aggregation because it only takes aggregation on one dimension (i.e. the time dimension). Intuitively, data values associated with an n-dimensional *perspective* can be from 0-D aggregation up to n-D aggregation. The only exception is *base*, whose associated data must be 0-D as it only stores data samples directly from the measuring devices. Our current implementation allows each *perspective* to associate one type of data value only. This makes transformations among *perspectives* neat and easy to manipulate.

4. OPERATIONS

Following the approach of DBMS, we also try to propose a generic data processing framework so that scientists could easily compose their routine data processing tasks and, as will be seen later, some generic optimization techniques could be applied to boost the processing performance. However, unlike the operators in traditional DBMS, operators' customizability is crucial to the scientific applications. This is actually one major reason why DBMS is not prevalent in scientific applications. Therefore, we endeavor to design generic but customizable operators, with which scientists could fill in their customized functions to form the specific operator that they need. By doing this, we can keep the benefits of having a generic processing framework while providing the necessary customizability.

HyperGrid adopts a transformation based framework; scientific data processing is modeled as transformations among different perspectives. Hence, the essence of HyperGrid is a sound and flexible perspective construction so that common data operations can be natively supported. In this section, we first describe the details of building a perspective, followed by an example to illustrate how common operations are supported by such construct.

4.1 Perspective construction

The construction of a new perspective (called target perspective) P_t typically requires one[2] reference perspective (called source perspective) P_s from which the new perspective is derived. To be clear, we will use subscripts *t* and *s* to distinguish *target* and *source* perspectives respectively. Cells in *target* and *source* perspectives are correspondingly referred to as *target cells* and *source cells* in the rest of the paper. Occasionally, the target and source perspectives are also called child and parent perspectives respectively when the context deems appropriate.

At the very outset, the *base* is used as the reference to create the first perspective. In addition to P_s, P_t may optionally require three pieces of information: a topological definition T_t, a data function D_t and an input selection function I_t. (i.e. $P_t = \langle P_s, T_t, D_t, I_t \rangle$)

As mentioned before, a topological definition T_t gives the internal layout of a grid. It determines the size and dimension of the cells within a perspective. Depending on the type of associated data values, grid layout can have different semantic meanings. For example, when the associated data is 0-D aggregation, the layout simply sets the grid granularity. On the other hand, when the associated data is k-D aggregation ($k > 0$), the layout also serves as part of the query semantic that instructs how data are grouped and aggregated. Notably, cell in a perspective inherits the character-

[1]This is mainly because the role perspective plays in HyperGrid is much more important than the role view plays in a DBMS.

[2]The only exception is perspective that implements *Merge* operation, in which case multiple source perspectives are required.

istics of traditional grid cell that captures the structural regularity. However, the former embodies a broader definition than the latter. A cell in a perspective is generalized as a logical computation unit, which may not be visualized as a single block or orthotope in a multidimensional space. For example, a cell can refer to a set of unconnected blocks or orthotopes that collectively form a logical unit. Also neighboring cells need not be disjoint or adjacent as in traditional grid. They are allowed to overlap or contain space between them (as in Example 8.1). We will see how this generalized notion of cell benefits query construction, especially for aggregation queries later in Section 4.3.

Data function D_t is the other component for perspective construction. It implements a scientific operation by dictating how data are transformed from P_s to P_t. The input of D_t are values associated with a set of source cells. The output of D_t is the computed result for some target cell e_t. Various forms of data functions for popular scientific operations are also discussed in detail in Section 4.3.

The construction of a new perspective involves both topological transformation and data transformation. These are two closely related processes. Topology conversion from T_s to T_t is implicitly performed through the output to input mapping of the data function D_t. Notice the output of D_t corresponds to the value of a cell confining to the topology T_t. However, the input of D_t is from cells confining to the topology T_s. Although it is not always the case, for some operations an explicit user-defined input selection function I_t is needed to instruct how cells under T_s should be selected to compute a target cell under the topology T_t. The function I_t takes one target cell $e_t \in P_t$ as input and returns as output the set of source cells $\{e_s | e_s \in P_s\}$ that will contribute to computing the value for the cell e_t.

As a final note, when a query is formulated as a series of perspective transformations, the last perspective in the series, called *surface perspective* (or simply *surface*), defines the final query results. In addition to the parameters above, a *surface* has one more optional parameter called *clipping window*, which defines a scope in the spatial and temporal domains where only data points within the defined window are returned.

4.2 Relationship between perspectives

Input selection function ensures data computation is carried out on the correct data set. However, such function is often not necessary for constructing a new perspective as long as the defined data transformation is *Location Consistent* as defined below:

DEFINITION 4.1. *Let $V(e)$ denote the scope of a cell e defined in the spatio-temporal domain. And let I_t be the input selection function for some data function D_t. The corresponding data transformation is said to be* Location Consistent (LC) *if the following* Location Equivalent *condition holds:*

$$V(e_t) = \bigcup_{e_s \in I_t(e_t)} V(e_s), \tag{1}$$

All other transformations that violate the Location Equivalent *condition are categorized as* Location Across (LA) *transformations.*

The input selection function can be omitted for *LC* transformation because target cell and its contributing source cells can be automatically paired through their Space-Time Identity. Hence, user-defined input selection is only required for perspective computed from *LA* transformations. Fortunately, as we shall see later, most of the operations belong to *LC* transformations. Hence, by exploring the important "location equivalent" relationship among perspectives, operations can be defined in a more concise way. Moreover, the query executor can also take advantage of *LC* property to optimize the query execution as discussed in Section 6.2.1.

Target Perspective Parameter	Default value or rule
Source Perspective P_s	the *base*
Topology Definition T_t	T_s
Data Function D_t	n.a. (compulsory parameter)
Input Selection Function I_t	*LC* rule

Table 1: Default settings for perspective parameters

4.3 Operators

HyperGrid provides users great freedom to create their own data operations through customized perspectives. We have described in the previous section that the definition of a perspective P_t is a quadruple $\langle P_s, T_t, D_t, I_t \rangle$. As examples, we show in this section how popular operations (convert, merge, interpolate and aggregate) in scientific sensor data processing can be readily supported by this construct. Although four parameters need to be supplied for the standard definition, in practice some of the parameters (such as I_t as described in the previous section) can be omitted by taking their default actions. Table 1 lists the default settings when the corresponding parameter is not specified. Table 2 summarizes the characteristics of the perspectives implementing these popular operations.

4.3.1 Convert

The *convert* operation converts data points in P_s to other values in P_t. The operation can be used in different ways for different purposes. One simple usage is to scale up or scale down values in the grid dataset by introducing a scaling factor in the data transformation rule. As another example, in data preparation phase, *Convert* can serve as a filter to clean corrupted sensor readings. This is achieved by converting erroneous data in P_s to "NULL" or some default values for the corresponding grid cell in P_t. A perspective that implements *convert* duplicates the topology of the source perspective (i.e. the default setting) since *convert* does not involve any structural change of the grid. Hence, other than P_s, the data function D_t is the only parameter to be specified, which can be formulated as follows:

DEFINITION 4.2. *Given $T_s = T_t$, let C denote the conversion function. The transformation rule D_t is:*

$$D_t(e) = C(P_s[e]),\ \forall e \in P_t \tag{2}$$

In the above definition, both P_s and P_t refer to the topologies instead of the entire perspectives. Because source and target perspectives share the same topology ($T_s = T_t$), for each grid cell e in the target perspective grid, we can find a corresponding data value associated with that cell in the source perspective.

4.3.2 Merge

Merge is the only operator which takes multiple perspectives as input. It is often used for producing a model that integrates multiple types of measurements, each represented by one source perspective. The operator enforces all source perspectives to have the identical topology and produces one target perspective with the same topology. Similar to *Convert*, the data transformation rule D_t is the only parameter to be customized, which can be defined as follows:

DEFINITION 4.3. *Given N is the number of source perspectives ($N > 1$), and $T_t = T_{s_i}, \forall i \in N$. Let $d(e)$ denote the set of*

Operator	# of Source Perspectives	Topology Definition	Data Function	Input Selection Function (data transformation type)
Convert	One	Default	User defined function	Default (*LC*)
Merge	Multiple	Default	User defined function	Default (*LC*)
Interpolate	One	New definition	User defined function	User defined function (*LA*)
Aggregate	One	New definition	User defined function	Default (*LC*)

Table 2: Characteristics of perspectives for different operators

data values from $\{P_{s_i}[e] | \ i \in N\}$. *The transformation function for Merge is:*

$$D_t(e) = \Gamma(d(e)), \forall e \in P_t \tag{3}$$

where Γ *is a user defined function that merges the corresponding cells from each of the source perspectives.*

4.3.3 Interpolate

In managing scientific data, especially environmental data, *interpolation* is such a popular yet expensive operation that deserves particular attention. As input are measurement readings, which are samples taken from continuously running physical processes (such as solar radiation, wind speed), without temporal interpolation it is very difficult to answer queries that ask for data at some point in time when no measurements were taken. Analogously, meteorologic phenomena monitored by WSN usually comes with the "coverage-holes" problem owing to the sparsity of the network or nodes failure. In the SensorScope project, to set up sufficient number of sensing stations in order to provide exhaustive coverage over a monitored region is infeasible due to prohibitive deployment costs. Hence, scientists also resort to spatial interpolation to generate a comprehensive data map for research and analysis.

Interpolation is a typical example of *LA* transformation. A perspective that implements *interpolation* defines its own grid layout T_t and data transformation rule D_t. T_t generates a set of new grid cells whose associated data values are to be interpolated. Computation for the interpolated points are defined by D_t, which comprises two steps. In the first step, a customized input selection function I_t is used to select candidate grid cells from P_s that will contribute to the computation for the grid cell in P_t. This is followed by applying a computation function to data values associated with the candidate cells to produce the interpolated result for the target cell in P_t.

DEFINITION 4.4. *Let* I_t *denote the input selection function for interpolation and* C *denote the corresponding computation function. The transformation rule for Interpolation is:*

$$D_t(e_t) = C(e_t, \Phi, \{P_s[e_s] | \ e_s \in I_t(e_t)\}), \forall e_t \in P_t \tag{4}$$

where Φ *is a statistical model based on which the interpolated value is calculated.*

4.3.4 Aggregate

Scientific data processing involves extensive aggregation operations for two reasons. Firstly, aggregation is used to compress sheer volume of data generated by the measuring devices to a manageable level. Secondly, scientific observations or assertions are typically supported by statistically significant data computed by certain aggregation functions rather than individual data readings.

Here we focus on aggregations with "group-by" clause on temporal or spatial attributes only since a predominant number of queries belongs to this type. The HyperGrid model natively supports spatio-temporal data aggregation because n-D data in a perspective essentially represents the n-D volume of the corresponding spatio-temporal span defined by its associated grid cell. This implies that for an aggregation perspective, the target topology T_t constitutes an important part of the aggregation semantic. Notably, each grid cell is an abstracted spatio-temporal notion, which may not be necessarily visualized as a single block or orthotope as in the traditional grid. This generalizes the concept of grid cell and gives user great flexibility to construct the "group-by" criteria. For example, user may want to know the breakdown by each hour the average temperature for a given region for the past 30 days, e.g. the average temperature of the past 30 days between 00 : 00 and 00 : 59, between 01 : 00 and 01 : 59 etc. Such queries are difficult to model by traditional grid construct since each grid cell in the result set refers to 30 segregated spatio-temporal blocks which are evenly spaced by 24 hours in the time domain. HyperGrid model allows multiple physically segregated blocks to form a single logical cell because in HyperGrid each cell is characterized by its spatial and temporal features, not just by a single cell boundary specification.

Like interpolation, the data transformation for an aggregation perspective is also a two-step approach. However, for the input selection step, user defined function is no longer required since aggregation belongs to *LC* data transformation. For the computation step, it simply applies an aggregation algebra (SUM, AVG etc.) to the candidate cells that transforms the k-D data in P_s to $(k+m)$-D data in the corresponding cell in T_t, where m is the number of dimensions whose associated values are aggregated.

DEFINITION 4.5. *Let* $I_{LE}(e_t)$ *denote the function that returns the set of source cells that collectively define the* location equivalent *scope as that of* e_t *in the spatio-temporal domain. And let C be the function that implements the aggregation algebra. Then the data function* D_t *can be formulated as:*

$$D_t(e_t) = C(\{P_s[e_s] | \ e_s \in I_{LE}(e_t)\}), \forall e_t \in P_t \tag{5}$$

4.3.5 Other operations

The perspective construction is a generalized notion to capture the transformation-based operations over grid data. In fact, the model is flexible enough to express more sophisticated operations other than the standard operations. In essence, any grid-based operations that can be characterized as topological change, data change, or both are supported by the construct.

4.4 Illustrative example

We refer the readers back to our motivating example 2.1. Given the above definitions, we can organize the required operations into a query tree, which consists of a series of data transformations from the *base* all the way to the *surface* (i.e. the output) as shown on the left of Figure 2. Each box in the graph represents a perspective that implements one scientific operation. Arrows pointing from source perspective to target perspective represent the data flow. Inside each box, there are three parameters separated by comma. They represent, from left to right, topology definition, data function and input selection function. If a default is taken, the parameter is replaced by a "*" in the corresponding position. A graphical illustration of the query execution is shown on the right of Figure 2.

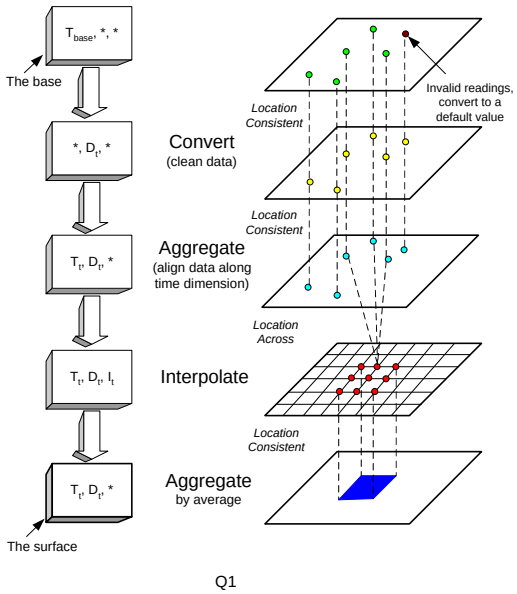

Figure 2: Illustration of the query execution in Example 2.1

5. QUERY EXECUTION STRATEGIES

There are two basic execution strategies: "top-down" and "bottom-up". The "top-down" strategy initiates the computation at the base perspective. The executor follows the data flow and materializes each of the perspectives one by one along the query tree. As an example, for the query plan in Figure 2, the executor first materializes the child perspective of the *base* (i.e. the *Convert* perspective), then uses the obtained results as the source to compute the next level perspective and so on until the *surface* is reached. However, a big drawback of this strategy is that the "top-down" computation leads to blocking execution: No output will be produced until the final surface perspective starts to get materialized.

On the contrary, in "bottom-up" strategy, the computation begins at the *surface* and carries out upward in a pipelined manner: For each target cell in the *surface*, the contributing source cells in its parent perspective need to be computed first. Then for each of these source cells, it in turn has to call the cells in its parent perspective to get itself computed. This continues until the *base* is reached with the actual values getting returned. The whole process essentially resembles the iterator model [16] in traditional DBMS. Different from "top-down" approach, "bottom-up" strategy does not require any intermediate perspective to be materialized and it produces results in a progressive way (like online query processing). This is desirable because it allows scientists to terminate the processing prematurely once they are unsatisfied with the partially produced answer.

However, "bottom-up" strategy is not a very efficient approach (which will be explained in Section 6.2). Therefore, what *HyperGrid* actually adopts is a hybrid strategy which combines the "bottom-up" with the "top-down". We call it *hybrid-k*, which means for a query plan with N perspectives, the top k perspectives in the query tree are computed first in a "top-down" manner while the lower $(N-k)$ perspectives are then computed using "bottom-up" method. In fact, the pure "top-down" and "bottom-up" approach can be seen as the special cases of the *hybrid-k* strategy, where "top-down" corresponds to *hybrid-N* and "bottom-up" corresponds to *hybrid-*0 respectively. In section 6.2, we will explain in detail why *hybrid-k* strategy is superior and how to determine the optimal k value for a given query plan.

6. OPTIMIZATION TECHNIQUES

As scientific queries usually take as input enormous amount of data and process them with expensive user-defined functions, how efficiently these queries can be executed becomes a critical issue. In this section we explore opportunities to optimize scientific query executions under the generic *HyperGrid* model.

6.1 Preprocessing and query rewrite

By preprocessing the original query plan, a series of perspective transformations can be rewritten in a succinct fashion. The goal of query rewrite is to produce a more economic plan that leads to reduced runtime costs without compromising the output quality. One effective approach is to coalesce adjacent perspectives in a query plan. The benefit of perspective coalescence is evident. Firstly, with fewer perspectives, the number of function invocations are reduced. Because the number of function invocations for each perspective computation is proportional to the number of cells in that perspective, minimizing total number of perspectives leads to considerable savings in terms of function call overheads. Secondly, the amount of buffered intermediate results are also reduced with fewer number of perspectives. Scientific computations may generate intermediate data that are too huge to be buffered in the memory. Hence the reduction of intermediate results may directly amount to the reduction in disk I/O.

Of course, it is not always possible to coalese any pair of adjacent perspectives in a query plan. At least one of the perspectives has to be *coalesce-amenable* in order to ensure query results are not compromised therefrom. A perspective is said to be *coalesce-amenable* if it uses default topology definition and *LC* transformation. For example, any perspective that implements the *convert* is *coalesce-amenable*. A *coalesce-amenable* perspective is free to choose to combine either with its parent perspective or with its child perspective. The coalescence process involves two steps: 1) map the topology of the parent perspective to the topology of its child; 2) merge data transformation functions through function composition. For example, given a *coalesce-amenable* perspective $P_k = \langle P_j, "default", D_k, "default" \rangle$ and its parent perspective $P_j = \langle P_i, T_j, D_j, I_j \rangle$, they can be combined to form a new perspective $P_m = \langle P_i, T_j, D_m, I_j \rangle$, where $D_m = D_j \circ D_k$ ($\circ$ denotes function composition). Similarly, if $P_l = \langle P_k, T_l, D_l, T_l \rangle$ is the child perspective of P_k, it is also possible to coalesce P_k with P_l to produce $P_n = \langle P_j, T_l, D_n, I_l \rangle$, where $D_n = D_k \circ D_l$. If the resultant perspective is still *coalesce-amenable*, it can continue to coalesce with its adjacent perspective. A query rewriter scans through a query plan and performs perspective coalescence until there is no more *coalesce-amenable* perspective existing in the plan or the plan is left with only one perspective.

For the query in Example 2.1, the query rewriter will coalesce the *convert* and *align* operators after the preprocessing.

6.2 Optimizing query execution

Section 5 has introduced two basic query execution strategies. The "bottom-up" approach is generally preferred over the "top-down" approach because the former allows query results to be produced in a progressive way. However, in practice, we find the "bottom-up" strategy may not be very efficient for two reasons. Firstly, it may lead to significant redundant computations. Secondly, a "bottom-up" execution may involve some "dull" computations which are useless to the query result. We explain these two issues and propose optimization techniques to tackle them in the following two subsections.

6.2.1 *Iterator with buffering*

The first problem with the "bottom-up" strategy is redundant computations. When a target cell in the child perspective requests the value for some cell in the parent perspective, the system would not know whether the same value has been computed before because nothing is saved or materialized in a "bottom-up" execution. Every data request will be computed from scratch following the iterator model. However, we show in this section that deliberate buffering strategy and intelligent choice of order in producing the cells of the surface perspective can effectively minimize such redundant computations.

Before delving into the details of the optimization techniques, let us first look at a strategy alternative to the basic iterator model. We attach a buffer for each intermediate perspective in the query tree. During the iteration, whenever a *NextCell*() function (analogous to the *Next*() function in an iterator) is returned, the results are stored in the attached buffer of the corresponding perspective. If, at a later time, the value of the same cell is requested again, the system can obtain the result directly from the buffer without recursively invoking the next level *NextCell*() function for the second time. Obviously, the buffering strategy avoids expensive redundant computations and hence reduces the query latency provided there are sufficient memory space to hold the intermediate results. Therefore, the crux of this approach is an efficient buffer strategy with low memory overheads and high hit ratio.

When a query only involves LC transformations, such buffer strategy is relatively easy to design, thanks to the topological regularity and cell's spatio-temporal identity that effectively correlate the perspectives in a query tree[3]. The spatio-temporal identity allows the system to identify cells in ancestor perspectives that contribute to the target cells to be computed. The topological regularity makes it possible to produce ordered output with respect to any spatial or temporal dimensions.

To ease exposition, we define what we call *Candidates Window (CW)* here. A *CW* defines a dynamic subspace of a perspective where cells subsumed by this space may potentially contribute to the future output. Notably, if all transformations are location consistent (LC), then perspectives along the query path will all share the same *CW*. In the beginning when the computation has not started, the *CW* is essentially the *clipping window* (refer to section 4.1) defined in the *surface* and buffers attached to each intermediate perspective are empty. The buffers begin to be filled with intermediate results when the computation starts. For *CW*, it starts to shrink as more output cells have been generated. Eventually, the size of *CW* reduces to zero when computation completes. Because buffers attached to each perspective only need to cache results for cells contained in the current *CW*, the buffer manager regularly expires cells in the buffer whose location (identified by its spatio-temporal coordinate) has fallen out of the latest *CW*. This strikes a dynamic balance such that the buffer size remains stable. Obviously, the space efficiency of the above buffering scheme depends on how fast *CW* shrinks with respect to the growing intermediate results during runtime, which in turn is determined by the order of the output sequence. For example, if output cells are generated in time ascending order, then *CW* will shrink steadily along time dimension from the lower end of the *clipping window* to the higher end during the query execution. Noticeably, buffer management using *CW* ensures optimized hit ratio because data discarded by the buffer is guaranteed not to be requested again by the subsequent computations. Also fine-grained buffer control is possible to improve space efficiency by taking multiple space-time dimensions as the sorting keys. Furthermore, the choice of dimension as the primary sorting key may directly impact the total buffer size needed for the query execution. Due to space constraints, we omit these details here.

When a query plan includes perspectives with LA transformations, however, optimal hit ratio can hardly be guaranteed for buffers corresponding to perspectives ascendant to the LA transformation perspective. That is because an LA transformation runs user-defined input selection function, which can choose any cells from its source perspective. This renders *CW*-based buffer strategy useless because the location equivalent property no longer holds. Nevertheless, we observe that most user-defined input selection criteria are not completely arbitrary. In fact, almost all of them exhibit certain locality property. This inspires us to use a *lookahead* heuristic to replace the *CW*-based buffer strategy for LA transformations. The idea is to run the input selection function in advance of the actual data computation for the target cells. By looking ahead the set of source cells that will be used to compute the next few target cells, the buffer manager can make intelligent decisions by only caching the results for the top k most referenced source cells (provided their results have already been computed previously). Owing to locality property, target cells in the vicinity are likely to share a big portion of source cells. This makes the *lookahead* approach practically effective in many cases.

6.2.2 *The* hybrid-k *strategy*

We also find the "bottom-up" strategy could sometimes involve "dull" computations which are useless to the query result. The reason for this has to do with the underlying structure for data storage. When a defined perspective contains a lot of holes (i.e. when the valid data point density of a perspective is low), for space efficiency the system will choose sparse array to represent that perspective, instead of an ordinary array. Note that an important characteristic of a typical HyperGrid query is that the density of valid data points of perspectives along a query path is often in non-decreasing order from the *base* to the *surface*. (Particularly, if a query involves interpolation, all descendent perspectives will have data point density of 100% since there will be no holes in the perspective after interpolating the space.) Hence, a typical scenario is that the system switches from sparse array implementation to ordinary array implementation for some perspective along the query path and continues to use ordinary array up to the *surface*.

Now consider the "bottom-up" strategy which iterates the computation from the *surface* to some perspective with very low data point density. It is very likely that the requested cell is a hole, which does not associate a valid data. However, under the "bottom-up" strategy the system would not know this, and the iteration therefore continues until the *base* is reached. As a result, some NULL values are returned and resources are wasted on computing something with NULL as input. In comparison, the "top-down" approach does not have this problem. This is because in "top-down" execution, perspectives are materialized **as a whole** one by one from the *base* downward. Computing a new perspective from a materialized sparse array only involves computations on those valid data points. One concern here, however, is that by using "top-down", it violates our initial requirement of generating the results in a progressive way since "top-down" execution is a blocking process. Therefore, the *hybrid-k* strategy comes into the picture. Because perspectives near the top of a query tree can have very low data

[3]It is worth noting that this buffer strategy is only meaningful when cells in the output perspective are overlapping. Otherwise, no buffer is needed because intermediate results will not be shared among output cells that are disjoint. A query executor can easily determine whether cells in the output perspective overlap and decide the necessity of enabling the buffer strategy.

point density (typically less than 0.05), materializing them may not sacrifice much in terms of query responsiveness. The objective here is to strike a balance between "top-down" and "bottom-up" strategy (i.e. to find an optimal k value) so that the query's average response time is minimized (the metric we use to measure user's satisfaction). We first formally define the average response time for a given query as follows:

DEFINITION 6.1. *Let $R(e)$ denote the latency from the time when computation for a query's* surface *perspective starts to the time when one of its output cell e is produced, the average response time for that query is defined as:*

$$\frac{\sum R(e)}{n_{sf}}, \forall e \in P_{sf} \quad (6)$$

where P_{sf} denotes the surface *perspective and n_{sf} is the total number of output cells in P_{sf}*

In the "bottom-up" strategy, we have the following recurrence relationship:

$$\begin{cases} R(e_1) = r(e_1) \\ R(e_m) = R(e_{m-1}) + r(e_m) \end{cases} \quad \forall e_m \in P_{sf}, m > 1 \quad (7)$$

where $r(e_m)$ is the latency from the request to compute the cell e_m is generated at the *surface* until the result is returned. Intuitively, $r(e_m) = \sum_{i=1}^{N} r_i(e_m)$ where $r_i(e_m)$ is the time taken to compute e_m at perspective i and N is the total number of perspectives. Assume the buffering strategy described in the previous section is enabled. Also for simplicity, assume $r_i(e_m)$ to be equal for all $e_m \in P_{sf}$ (say, it is μ_i). Then the average response time using the "bottom-up" strategy can be estimated as:

$$\frac{(n_{sf}+1)\sum_{i=1}^{N}\mu_i}{2}, \text{ N is the total number of perspectives} \quad (8)$$

On the other hand, if we choose to compute the first top k perspectives from the *base* in a "top-down" manner and the rest $N-k$ perspectives still using the "bottom-up" approach, we get the following average response time:

$$\sum_{j=1}^{k} \rho_j n_j \mu_j + \frac{(n_{sf}+1)\sum_{i=k+1}^{N}\mu_i}{2} \quad (9)$$

Given $\rho_u \leq \rho_v, 1 \leq u < v \leq N$

In the above equation, ρ_j is the data point density of perspective j. It is a value between 0 and 1. n_j is the total number of cells in perspective j. As can be seen, whether a perspective should be computed "top-down" or "bottom-up" really depends on the value ρ. That is the density of the array for the corresponding perspective. Using equation 9, the optimizer can determine the optimal k value so that the average response time is minimized.

6.3 Optimization for visualization

The previous sections introduce several optimization techniques to reduce the query execution costs or average response time. However, these are not the only goals for an optimizer. Scientific research often involves tasks such as to discover unusual trend or pattern from a dataset or to collect evidence for supporting new hypothesis etc. For these purposes, scientists need to filter useful information from abundant test cases by running a large number of exploratory queries or "what-if" queries. A pure cost-based optimizer is deemed inappropriate for such scenario especially under interactive mode where user is sitting in front of the monitor waiting for the results to be visualized. This is because in cost-based optimization, output cells have to be produced in an order by space or time dimension. This leads to results generated in a raster manner. A big disadvantage of this is user will not be able to get a rough idea of how the results look like until the majority of the cells have been computed. Probably the following comment well describes what is actually desired for a visualization output: "Overview first, zoom and filter, then details on demand" [18]. In our case, an overview means to provide the insight instead of the the accurate answer for each output cell. This implies the computation should prioritize "interesting" regions in the surface perspective that would help reveal global trend or unusual patterns etc. The definition of "interesting" here is context-dependent. But often, it refers to portions in the result set where data values have greater variations. An ideal executor should focus on these portions and progressively refine the answers if the user continues to be interested.

Algorithm 1 A directed random walk algorithm for visualization

Notations:
P_i: vector representation of the location of cell i
v_i: value of cell i
s: default stride for one step of walk

1: Randomly select a cell P_1 with s distance away from the starting cell P_0.
2: $i := 1$
3: **loop**
4: **if** $IsComputed(P_i) == false$ **then**
5: $v_i := ComputeCell(P_i)$
6: **else**
7: $v_i := OutputBuffer(P_i)$ /* directly retrieve the results from output buffer without recomputing */
8: **if** $f(v_i, v_{i-1}) > threshold$ **then**
9: $U := Normalize(P_{i-1} - P_i)$ /* unit vector with direction from cell i to cell $(i-1)$ */
10: **else**
11: $U := Normalize(P_i - P_{i-1})$
12: $P_{i+1} := P_i + Rand(0,1) \times s \times RandGauss(U, g(v_i - v_{i-1}))$ /* random walk */
13: **if** $num\_of\_computed\_cells \geq c \cdot total\_num\_of\_cells$ **then**
14: break
15: $i := i + 1$

While a plethora of optimization techniques have been proposed for scientific visualization (see [19] for an excellent overview on the state-of-the-art techniques), we choose to propose a simple but effective algorithm for our application. The purpose is to show *HyperGrid* can facilitate efficient scientific visualization. More sophisticated visualization techniques may be included in the future when need arises. The algorithm we proposed, called *directed random walk*, is summarized as follows: The executor first randomly selects k cells that are uniformly distributed in the surface perspective. Then starting from each of the k cells, a *directed random walk* is performed to pick the next cell to compute. The details are sketched in Algorithm 1. Whenever a step is taken, a new cell (P_i) is selected and its value (v_i) gets computed (lines 4-7). The value (v_i), together with the value of the cell where the random step is taken from (i.e. v_{i-1}), are fed into a function to evaluate the interestingness of the region. If the returned value is greater than the threshold (line 8), it means more cells between the two

(P_{i-1} and P_i) need to be visited. The direction of the next step of the random walk is therefore set to have a mean U facing towards the previous cell (line 9). Otherwise, the mean direction will be a reversed one (line 11). The actual direction for the next step is determined by function $RandGauss()$ which returns a random unit vector following Gaussian distribution with mean U and variance a function of the difference between v_i and v_{i-1} (line 12). The query executor runs k random walks in an interleaving fashion and terminates when user interrupts or a certain percentage (typically less than 50%) of the total output cells have been produced (lines 13-14). At this stage, the users should already get a very good overview of the query results. If the execution has not been terminated, it means a complete and accurate result is needed. In that case, the executor reverts to the strategy described in Section 6.2 to compute the remaining output cells.

7. EXPERIMENTAL SETUP

In this section we describe our prototype system as well as the query and dataset used to experimentally evaluate the HyperGrid model.

7.1 Implementation

As an application specifically designed for processing scientific environmental data, we choose to build the system using Mathematica [13], rather than a general-purpose programming tool such as C or Java. There are three main reasons for this choice: Firstly, Mathematica, as an excellent tool for mathematical computations, has the built-in capability to optimize numerical computations, which makes it particularly suitable for processing computationally intensive scientific operations. With Mathematica, we can save efforts from finding the best algorithms for solving particular mathematical problems and focus on the query processing aspect of the system. Secondly, an important feature of Mathematica, which general-purpose programming tools do not provide directly, is the powerful support for symbolic computation. It allows physical models to be manipulated precisely throughout the computation. Finally, we choose Mathematica because it is the tool many scientists often use and hence are already familiar with. So it would be easier for them to maintain and extend the system when necessary in the future.

The system consists of three main components: a query engine, an optimizer and a user interface. The query engine is the core of the system. It executes a query plan according to the given execution strategy. For each operation in the plan, the engine compiles the topology and data specifications given in the source and target perspectives, and performs the data transformations accordingly when the operation is invoked. The optimizer does a few things: rewrites the query, computes the optimal execution strategy, interacts with the engine to implement the buffering strategy and schedules the output sequence as discussed in Section 6. Lastly, a user interface is provided to allow user to specify perspectives, including customized user-defined data and input selection functions. So far, all user-defined functions need to be written in mathematica code. However, with MathLink [14], a generalized application interface provided by Mathematica, we do not see big obstacles to incorporate the current system with user-defined functions in C, Java or other languages.

To evaluate the performance of the HyperGrid system, we also implemented a pure array-based approach using Mathematica for comparison. The array-based approach represents the traditional way in which scientific data is processed to answer a query. There is no integrated query engine which automates the step-by-step data transformations to reach the final answer. Instead, the program offers functions to perform each individual data transformation operation, such as convert, merge and interpolate. Hence, user needs to manually organize the query plan tree. In addition, because an array-based approach only performs computations between arrays without the spatial and temporal context, an external function is required to translate the spatio-temporal points to the corresponding array representation for each data transformation as well as any physical model involved.

7.2 Dataset

The data we use for our experiments were collected from a SensorScope network which was deployed at the Grand-St-Bernard pass in Western Alps at 2400 m in September and October 2007 to monitor the ecological condition of the region. There are totally nine types of meteorologic measurements, namely ambient temperature, surface temperature, humidity, solar radiation, soil moisture, watermark, rain meter, wind speed and direction. Each type of measurements consists of over $588,000$ data points. And each data point can be identified by the time and the location where it was measured.

7.3 Query set

We use two categories of queries for our experiments. The first category consists of the query in Example 2.1 and its variants (details are described in Table 3). They represent routine queries which scientists use frequently to compute statistical information about the data. Typical steps of routine queries include data cleaning (through *convert*), alignment (through *aggregate*), interpolation (through *interpolate*) as well as other query-specific operations. The second category simulates the scenario where user wants to explore the data through visualization. We use the query in Example 2.2 for our experiment.

8. PERFORMANCE EVALUATION

The performance evaluation has two objectives. Firstly, we would like to assess the usability of HyperGrid as a tool to manage scientific environmental data, and compare it against the traditional array-based method. Secondly, we want to evaluate the effectiveness of our proposed optimization strategies in improving the users' experience when they use the system.

All the experiments were conducted on a 2.33 GHz Intel dual core machine with 4 GBs of memory running windows XP.

8.1 Routine query execution

Figure 3 reports the overall runtime performance of the native HyperGrid (using "bottom-up" strategy without any optimization) implementation and the array-based implementation for four different routine queries. Q2, Q3 and Q4 are variants of Q1 that vary the workload by either changing the measurement type or the spatio-temporal scope of the *surface*. In all the cases, the queries go through four data transformations (*convert*, *align*, *interpolate* and *aggregate*). From the figure, it is evident that *interpolate* always takes up the majority of the run time. This is because interpolation often comes with expensive user-defined data and input selection functions. In our experiments, we adopt the kriging model for interpolation. It uses an adapted k-nearest neighbor (k-NN) algorithm as the input selection function and a variogram model to estimate the degree of dependence between data points. Typically, some 15 to 22 neighbors are selected and used in the variogram for each target cell to be interpolated. Comparatively, other operations use either the built-in data functions or user-defined functions with much lower complexity, hence they consume significantly less CPU.

Query ID	Query detail
Q1	Illustrated in Example 2.1
Q2	Same as Q1 except the measurement type changes from "ambient temperature" to "watermark"
Q3	Same as Q1 except the clipping window along the time dimension is increased by 100%
Q4	Same as Q1 except the clipping window along latitude and longitude dimensions are increased by $(\sqrt{2}-1)$ respectively
Q5	Illustrated in Example 8.1

Table 3: Query set description

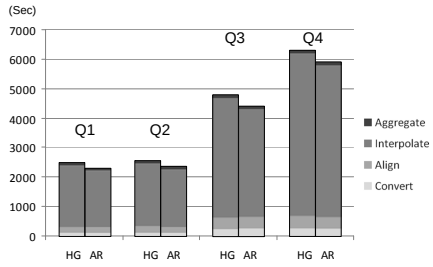

Figure 3: HyperGrid (HG) Vs. Array (AR)

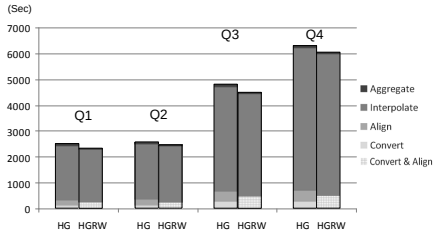

Figure 4: Effect of query rewrite

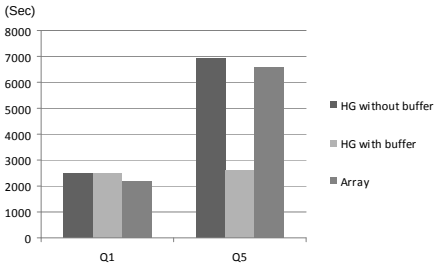

Figure 5: Effect of buffer strategy

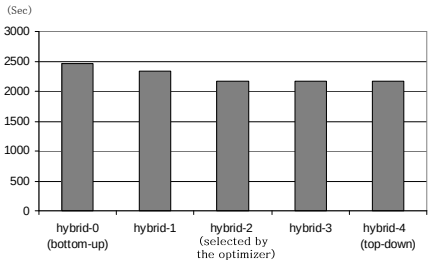

Figure 6: Optimizing execution strategy (total runtime)

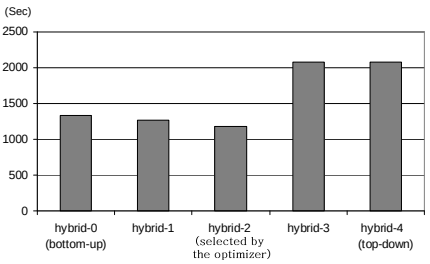

Figure 7: Optimizing execution strategy (average response time)

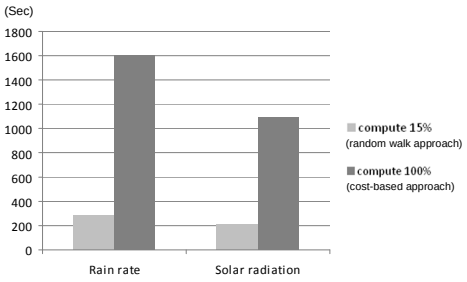

Figure 8: Run time cost for visualization

8.2 HyperGrid Vs. array-based implementation

The plot corresponding to array-based implementation (referred as "AR") in Figure 3 assumes the ideal scenario that human efforts for writing external functions (for each data transformations in the query plan) to translate spatial and temporal coordinates to the array elements are assumed to be zero. However, in practice this is often a tedious and time-consuming task, which cannot be quantified and reflected in the figure. Even with this unrealistic assumption (by ignoring all hidden costs incurred in array-based processing), HyperGrid (referred as "HG") only takes slightly longer time (less than 8%) than array-based implementation for all the test cases. This indicates that the *HyperGrid* model indeed incurs little overheads to the system. More importantly, by unifying all operations in a standard way and automating the entire query process, HyperGrid can explore optimization opportunities and further boost the runtime performance as illustrated in the next few sections.

8.3 Query rewrite

Figure 4 compares the runtime performance between *HyperGrid* query plans with and without query rewrite. We can see for all the test queries, the *convert* and the *align* perspectives are coalesced after the rewrite. And the rewritten plan (denoted by "HGRW") clearly runs faster than the original one for all the cases. However, the improvement is not very impressive. This is because the execution cost to compute the *convert* and the *align* perspectives does not constitute a significant portion of the total cost. If we break down the total runtime cost, for example Q1, we can see that the time taken to compute *convert* and the *align* in total is dropped from 341 sec to 245 sec. The saving is actually quite substantial.

8.4 Buffering strategy

Next, we evaluate the performance of the buffering strategy proposed in Section 6.2.1. The left part of Figure 5 depicts the runtime comparison between HyperGrid with buffering and two other approaches (native HyperGrid and array-based approach) for query Q1. It shows the buffered HyperGrid strategy does not improve the total runtime cost. This is expected because in Q1, all cells in the *surface* do not overlap in the spatio-temporal domain. It means the buffering strategy proposed in Section 6.2.1 would not be beneficial here because no previously computed results are reused. In order to evaluate the proposed buffer strategy for the case where output cells are overlapping in the spatio-temporal domain. We consider a new query variant as follows:

EXAMPLE 8.1. *Return the ambient temperature averaged over* 15*-minute interval for the period from 2007-10-01 00:00 to 2007-10-04 00:00 and for the region [45°52′1″N, 45°52′23″N] in latitude and [7°10′37″E, 7°10′59″E] in longitude on a* $1'' \times 1''$ *grid. The result should be updated every 5 minutes.*

The fundamental difference between Q1 and the above query (Q5) is that Q5 averages the data points over 15-minute interval

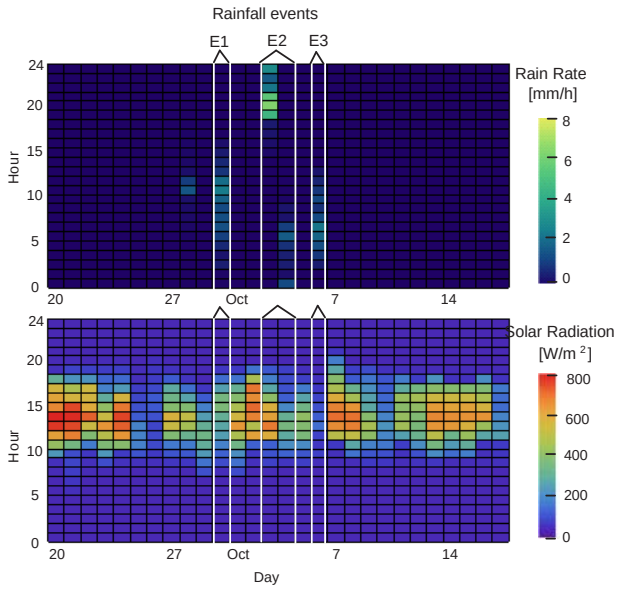

Figure 9: Plot with 15% result computed (using directed random walk algorithm)

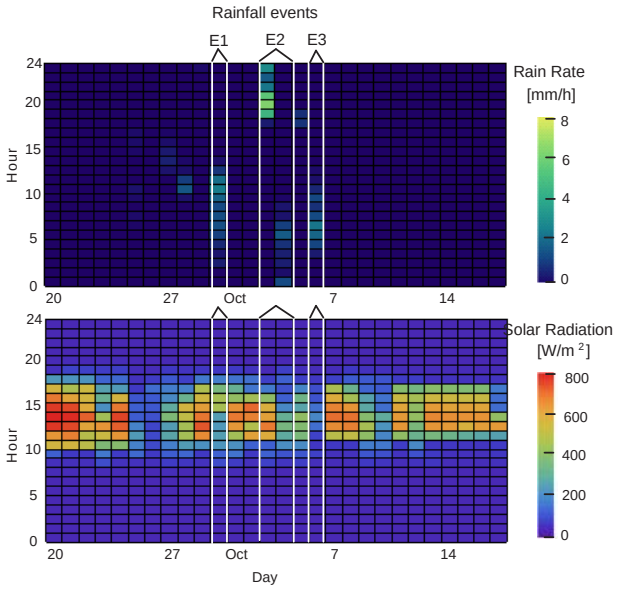

Figure 10: Plot with 100% result computed

instead of the entire three days. And a new result is generated for every 5-minute advancement along the time dimension. The output of Q5 essentially corresponds to results from a sliding window over time dimension with window size of "15-minute" and sliding step of "5-minute". This means each pair of the adjacent cells in the *surface* overlaps by "10-minute" on the temporal space. Experimental results of running Q5 is shown on the right of Figure 5. As expected, buffered HyperGrid strategy achieves significant runtime reduction this time owing to the effective buffer strategy that avoids doing the redundant computations. On the other hand, the native HyperGrid approach and the array-based method require much more time to process the query due to their inability to recognize and reuse previously computed intermediate results.

8.5 Optimizing execution strategy

Figures 6 and 7 depict the performance of all possible execution strategies for Q1 in terms of total runtime cost and average response time respectively. No other optimization technique, such as query rewrite, is enabled for this test case. So there will be 4 perspectives in the query plan and hence 5 possible execution strategies (from hybrid-0 to hybrid-4). Hybrid-0 is essentially the "bottom-up" strategy and hybrid-4 is the "top-down" strategy. In terms of the total runtime cost (Figure 6), hybrid-0 gives the worst performance due to the two reasons explained in section 6.2. The total run time gets reduced as more perspectives are computed in a "top-down" manner. It is no surprise that hybrid-4 gives the shortest total runtime. However, in terms of the average response time (which we think is the critical metric), hybrid-4 performs the worst. This is because no output cells are produced until the last perspective (i.e. the *surface*) starts to be computed. We can see from the figure that based on the cost model given in Section 6.2.2, the optimizer successfully finds the optimal execution strategy (in this case, hybrid-2) for the given query.

8.6 Visualization optimization

Lastly, we evaluate the optimization strategy for visualization proposed in Section 6.3. We use the scenario in Example 2.2 for this experiment. Basically the user needs to visualize the output of the hourly average rainfall rate together with the hourly average solar radiation values to discover whether the two are correlated. Figure 9 shows the results produced by the directed random walk approach which only computes 15% of the total output cells. The remaining 85% of the cells in the figure are obtained by applying a simple smoothing function. For comparison, the actual accurate output (with 100% output cells computed from the dataset) is shown in Figure 10. In both figures, the graph on the top indicates the hourly average rainfall rate for a period of 28 days (from Sep 20 to Oct 17 as indicated in the horizontal axis). Hour of a day is indicated in the vertical axis. The graph below is the corresponding solar radiation values during this period. As we can see, three rainfall events (E1, E2, E3) are identified from the rainfall rate graph. Each event is highlighted by a pair of white vertical bars that run across both graphs. In the solar radiation graph (either the one in Figure 9 or Figure 10), we can see that the values during the period of the rainfall events (especially E1 and E3) are lower (darker color) than other days for the same hour. This hints that rainfall and solar radiation are very likely to be correlated for this region. By comparing Figure 9 and Figure 10, we can see that most of the trends or patterns exhibited in Figure 10 can also be found in Figure 9. This clearly indicates that the directed random walk algorithm does a very good job in simulating the actual results by processing only a small fraction (15% in this example) of the output cells. Figure 8 reports the runtime costs for generating the two figures. For both the rainfall and solar radiation datasets, the time required by the directed random walk algorithm to simulate the results is less than $1/5$ of the time needed by the cost-based optimization algorithm to compute the full results. This shows the directed random walk algorithm can greatly improve the data exploration efficiency in many instances.

9. RELATED WORK

A traditional way to model regular gridded data is to use array, which is often regarded as the basic data type to store multidimensional data [17]. The database community has proposed quite a

few data models and languages to support array-based data management. AQL [10] is a calculus-based language for supporting low level array operations. Similarly, AML [12] also proposes a few operations for array manipulations. These operations, including those proposed in [3, 4] focus on aspects such as index patterns or sub-sampling of the array elements. Although these operations are important in image processing applications, they are not so useful for managing scientific environmental data. There are other forms of grid-based model proposed [9, 15]. However, those models are designed for other purposes. The feature of the work in [9] is an algebra of manipulating irregular grids, while [15] focuses on indexing technique for grided data. Moreover, although most of the scientific data, particularly monitoring data collected from sensors, carries important spatio-temporal identity, to the best of our knowledge, none of the existing array-based techniques takes that into account during operations. In MauveDB [8], the author proposed a model-based view approach to manage measurement data. While our notion of perspective shares the same flavor as the view in MauveDB, the two are in fact quite different. MauveDB focuses more on view maintenance issue to provide a consistent view to the user. Our approach, on the other hand, focuses primarily on query processing and using perspectives to implement and optimize scientific operations.

Scientific queries are often analytical. Hence, they typically involve data grouping or aggregation. Literatures on multi-dimensional OLAP have offered abundant techniques to speed up the performance for these queries such as [1, 7, 11]. In general, these techniques are difficult to be applied in our context. The reason is OLAP optimization techniques typically rely on the fact that the attributes for grouping by are known or at least deterministic before the query is issued. However, in our case, scientists can freely organize the data in the continuous spatio-temporal domain to form a query. Moreover, external physical model may be introduced to interpret or reorganize the data. These make the optimizer difficult to perform any pre-computation to improve the query response time. Data used in aggregation can also be modeled as volume [5, 6, 20], where spatial and temporal dimensions are treated indeed as continuum. The related techniques may be useful as a supplementary approach to support arbitrary grid granularity for HyperGrid in our future work.

10. CONCLUDING REMARKS AND FUTURE PLAN

We have presented the framework of *HyperGrid* in this paper. The primary objective of this work is to demonstrate that scientific data management can benefit from database technology that enables the integration of scientific workflows, which was largely segregated traditionally. With a uniform data model and database-style processing paradigm, scientific computations can be carried out in a more systematic way. An integrated architecture also reveals opportunities for query optimization. Based on scientists requirements, we have implemented several techniques to optimize query execution in the first version of our prototype system. Next, we plan to let the scientists test it, and enhance our design and techniques based on their feedbacks. In the future, we also intend to embrace more features into *HyperGrid*. One promising direction is data lineage tracing. Again, because of the integrated framework, we believe *HyperGrid* would be in a better position to support lineage tracing compared to other solutions.

11. ACKNOWLEDGMENTS

We would like to express our gratitude to Marc Parlange, Olivier Couach, Hendrik Huwald, Vincent Luyet, Daniel Nadeau and other members from the Laboratory of Environmental Fluid Mechanics and Hydrology at EPFL for their invaluable advice and suggestions given during the design stage of this work.

12. REFERENCES

[1] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *ICDE*, pages 232–243, 1997.

[2] G. Barrenetxea, F. Ingelrest, G. Schaefer, M. Vetterli, O. Couach, and M. Parlange. Sensorscope: Out-of-the-box environmental monitoring. In *IPSN*, pages 332–343, 2008.

[3] P. Baumann. Management of multidimensional discrete data. *VLDB J.*, 3(4):401–444, 1994.

[4] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. Geo/environmental and medical data management in the rasdaman system. In *VLDB*, pages 548–552, 1997.

[5] M. Chen, R. H. Clayton, A. V. Holden, and J. V. Tucker. Visualising cardiac anatomy using constructive volume geometry. In *FIMH*, pages 30–38, 2003.

[6] M. Chen, A. S. Winter, D. Rodgman, and S. Treuvett. Enriching volume modelling with scalar fields. In *Data Visualization: The State of the Art*, pages 345–362, 2003.

[7] Z. Chen and V. R. Narasayya. Efficient computation of multiple group by queries. In *SIGMOD Conference*, pages 263–274, 2005.

[8] A. Deshpande and S. Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD Conference*, pages 73–84, 2006.

[9] B. Howe and D. Maier. Algebraic manipulation of scientific datasets. In *VLDB*, pages 924–935, 2004.

[10] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *SIGMOD Conference*, pages 228–239, 1996.

[11] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. Olap on sequence data. In *SIGMOD Conference*, pages 649–660, 2008.

[12] A. P. Marathe and K. Salem. Query processing techniques for arrays. *VLDB J.*, 11(1):68–91, 2002.

[13] The wolfram mathematica homepage. http://www.wolfram.com/, 2008.

[14] The mathematica mathlink. http://www.wolfram.com/solutions/mathlink/mathlink.html, 2008.

[15] S. Papadomanolakis, A. Ailamaki, J. C. López, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD Conference*, pages 551–562, 2006.

[16] J. E. Richardson and M. J. Carey. Programming constructs for database system implementation in exodus. In *SIGMOD Conference*, pages 208–219, 1987.

[17] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, 2006.

[18] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL*, pages 336–343, 1996.

[19] B. Shneiderman. Extreme visualization: squeezing a billion records into a million pixels. In *SIGMOD Conference*, pages 3–12, 2008.

[20] T. Wang, S. Santini, and A. Gupta. An interpolated volume model for databases. In *ER*, pages 335–348, 2003.