SS-DB: A Standard Science DBMS Benchmark

Philippe Cudre-Mauroux[‡*] Hideaki Kimura[†] Kian-Tat Lim[+] Jennie Rogers[†]
Samuel Madden[‡] Michael Stonebraker[‡] Stanley B. Zdonik[†]

ABSTRACT

In this paper, we propose a new benchmark for scientific data management systems called SS-DB. This benchmark, loosely modeled on an astronomy workload, is intended to simulate applications that manipulate array-oriented data through relatively sophisticated user-defined functions. SS-DB is representative of the processing performed in a number of scientific domains in addition to astronomy, including earth science, oceanography, and medical image analysis. The benchmark includes three types of operations: (i) manipulation of raw imagery, including processing pixels to extract geo-spatial observations; (ii) manipulation of observations, including spatial aggregation and grouping into related sets; and (iii) manipulation of groups, including a number of relatively complex geometric operations.

We implemented the benchmark on two systems. The first is a sophisticated implementation in MySQL based on spatial indexes with a custom-built distribution layer; the second is an implementation on SciDB, a recently proposed [9] scientific database system, built from the ground up to manipulate array-oriented data. We find that SciDB performs substantially better than MySQL, due to a number of architectural features, including its use of columnar storage, aggressive compression (including avoiding the storage of array indices when possible), and an efficient, chunk-based storage manager that is able to parallelize many operations and perform most I/O sequentially.

1. INTRODUCTION

Current science users are generally very unhappy with relational DBMSs, for at least the following reasons:

1. Many science applications are hard to represent in the relational model. Rather, scientists tend to organize much of their data in ordered data models, such as arrays and vectors.
2. Most science users need specialized operations like frequency transforms, matrix decompositions, and multidimensional windowing that are not easy to express in SQL.
3. Required features such as provenance and version control are missing in current implementations.

The first two issues tend to generate unacceptable performance; all three make RDBMSs difficult to use for scientific data. These issues have been clearly articulated at recent XLDB workshops [5–7] and stop most large science applications from using RDBMSs. Instead, the common practice is to build capabilities on top of the "bare metal", using RDBMSs either not at all or only for the metadata of the application.

In recent years, there have been several research efforts to build database systems that offer a more natural data model than tables, such as arrays. Approaches include:

1. Using an array simulation on top of a relational DBMS. This is the approach suggested by MonetDB [16].
2. Using an array executor on top of blobs in a relational DBMS. This is the approach suggested in Rasdaman [4].
3. Using a from-scratch native implementation of arrays. This is the approach chosen by SciDB [9].

In order to properly understand the performance differences between these models, in this paper we propose a benchmark for scientific database systems called SS-DB. In contrast to previous benchmarking efforts (such as Jim Gray's "20 astronomy queries" [14]), we wanted SS-DB to be relatively general and not tailored to a particular scientific discipline. In addition, Gray only queried derived ("cooked") data sets. In contrast a complete benchmark should model all phases of scientific data management, namely:

1. Data ingest of raw imagery or sensor data
2. Queries to the raw data
3. Cooking the raw data into a derived data set
4. Queries to the cooked data

SS-DB includes all four phases of processing. It is loosely based on the requirements of the Large Synoptic Survey Telescope (LSST) [11, 15] but, to the best of our knowledge, also represents at least some requirements of earth science, oceanography, and medical imaging which deal with large arrays or meshes of data and perform similar operations as appear in our benchmark.

Astronomers acquire data using telescopes, detecting and measuring star-like and galaxy-like features in the images. Earth scientists acquire data using satellites or aircraft, detecting and measuring vegetation patterns or wildfire boundaries or gravitational anomalies. Oceanographers acquire ocean temperature or wave height or wind velocity data using satellites, detecting and measuring differences from models or long-term averages. Medical practitioners acquire data using X-rays or MRI, detecting and measuring tumors or other anomalies. All of these disciplines perform analyses on raw and derived data, including tracking the derived detections over time.

In addition to proposing a new benchmark, we highlight the potential performance gains of a purpose-built scientific data processing engine over a conventional relational system by providing SS-DB results for MySQL and SciDB (approach 3 above). We found that it was necessary to mix in aspects of approaches 1 and 2 to enable MySQL to execute the benchmark in a reasonable way. We hope additional systems will choose to run this benchmark and provide other architectural data points.

Our results show that SciDB is able to significantly outperform MySQL in all phases of processing, running the benchmark queries two orders of magnitude faster overall. The main reasons SciDB is able to outperform MySQL include:

- SciDB's array-oriented data model enables direct offsetting and omission of index attributes.
- SciDB uses a column-oriented design where each attribute is stored separately.
- SciDB uses aggressive compression.
- SciDB partitions data into overlapping tiles, which allows some operations to be processed in parallel in SciDB while they must be processed sequentially in MySQL.

In summary, the major contributions of this paper are:

- We propose a new scientific data processing benchmark, designed with collaboration from domain experts, that is representative of tasks real scientific users want to perform. It

*‡MIT CSAIL: {pcm,madden,stonebraker}@csail.mit.edu
†Brown University: {hkimura,jennie,sbz}@cs.brown.edu
+SLAC National Accelerator Lab: ktl@slac.stanford.edu

includes not just processing on cooked data, but operations over raw data and cooking operations themselves.
- We have developed an initial implementation of the benchmark on MySQL and SciDB and a performance comparison of the two approaches.
- We show the benefits that a ground-up implementation of an array-based data model can offer on SS-DB.

The rest of this paper is organized as follows. In Section 2, we present the benchmark specifications. Then, in Section 3 we discuss the SciDB implementation, followed in Section 4 by the MySQL implementation. Section 5 presents the results that we observed, with a discussion of the differences observed. Lastly, we review related work in Section 6 and conclude this paper in Section 7.

2. BENCHMARK COMPONENTS

The benchmark encompasses ingesting raw data (presumably from a sensor system, such as a space telescope or medical imager), cooking it into two different derived forms useful for further analysis, and then querying both the raw and derived data to find information of interest. We describe the ingest and cooking phases in the next three subsections, followed by the queries which must be executed. We then give instructions for how these phases are to be combined when running the benchmark. Lastly, we give benchmark parameters, which control the benchmark runs. A data generator that can produce the data for all configurations of this benchmark, canonical implementations of all of the required functions, and a more detailed description are available on our web site [1].

2.1 Raw Data Ingest

Raw data arrives as 2-D arrays. Such data could be from astronomy (e.g., telescope images), remote sensing (e.g., aircraft observations), oceanography (e.g., satellite imagery), medical imaging (e.g., X-rays), or simulations in a variety of fields.

Each 2-D array is in the same local coordinate system, which for simplicity starts at (0,0) and has increasing integer values for both dimensions, ending at 7499. Hence, each array has 56,250,000 cells. Each array cell has 11 values, V1,...,V11 which are 32-bit integers. Five of these values are based on actual observational data which, as is typical, has concentrations of "interesting data" in the total array space interspersed with data ("background") of less interest; the other six values are chosen non-uniformly to simulate scientific data. Notice that each array is 2.48 Gbytes in size.

The benchmark contains 400 such arrays taken at distinct times T1,...,T400. In practice, the times of these observations will not be regularly spaced, but for simplicity we assume regularity, i.e. array times are T, 2T, 3T, etc. The 400 arrays are arranged into 20 cycles, each containing 20 arrays. Scientific data is often processed or reprocessed at periodic intervals, simulated by these cycles. With 400 arrays, the total benchmark is 0.99 TBytes.

Moreover, there is a world coordinate system, which goes from 0 to 10^8 in each dimension. Each raw image has a starting point (I,J) in this coordinate system. Figure 1 shows this pictorially.

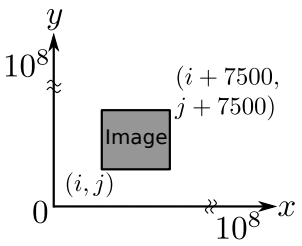

Figure 1: Coordinate space.

Images at different times have different values for (I,J). While a full data set would include samples from the entire world coordinate system, for the purposes of this limited-size benchmark, 80% of the images have I and J uniformly distributed between 49,984,189 and 50,015,811. The other 20% have I and J uniformly distributed between 0 and 99,992,499. The dense region in the center simulates a realistic rate of repetitive examination of a single region, while the sparse random coverage ensures that implementations must handle the entire global coordinate space.

Raw data can be compressed in any way the user chooses. However, any compression must be lossless.

We call the above specification the *normal* dataset. In order to support installations that are modest in size, there is a *small* version of the benchmark. This has a smaller number of images (160 in 8 cycles of 20) that are 3750 by 3750 in size in the same world coordinate system and have 80% distributed between 49,995,000 and 50,004,999. Hence, the smaller benchmark is 99 Gbytes.

There is also a *large* configuration. Here, there are 1000 images in 50 cycles of 20, each 15,000 by 15,000, again in the same world coordinate system with 80% distributed between 49,950,000 and 50,049,999, for a total size of 9.9 TBytes.

2.2 Cooking Data into Observations

A user-defined function F1 (see Appendix A-1), provided with the benchmark data generator, sequences through the raw 2-D array values and produces a collection of observations. Each observation consists of:

1. An (x,y) center, where x and y are 4-byte integers between 0 and 10^8. An observation can be identified by looking only at cell values within a square box of size D1 index values centered on (x,y) in the world coordinate space. The choice of D1 is discussed in Section 2.6.
2. A polygon boundary, B, specified as an ordered list of 4-byte (I, J) points in the world coordinate system, each between 0 and 10^8. Polygons have an average of 12 sides. If the number of edges in B for a given observation is greater than a maximum specified by value D2, that observation is ignored in order to filter out artifacts and excessive noise contamination. Each polygon has a bounding rectangle, whose length and width are bounded with a maximum size of D1.
3. 2 other observation-specific values (avgdist, pixelsum), which are 8-byte floating point numbers. These are meant to represent domain-specific observation information.

Each raw array generates an average of 2.3×10^4 observations, which amount to about 4 Mbytes. The data set will grow over the life of the benchmark to 1.6 Gbytes as new data arrives. Again lossless compression can be applied to this data set, if desired.

Observations could correspond to stars (astronomy), boundaries of vegetation regions (satellite imagery), temperature regions (oceanography), or medical anomalies (medical imaging).

2.3 Cooking Observations into Groups

One expects to see the same physical phenomena at different times in different arrays during the study. As such, we want to place observations into groups that appear to describe the same physical phenomenon. A physical observation may appear at different coordinates because it may be moving or the sensor may be in a different position.

Given a collection of groups assembled so far, a new observation, B1, is put into all existing groups for which a user defined function F2 (B1, B2) (see Appendix A-2), provided with the benchmark data, returns true for some B2 already in the group. Since there is invariably a maximum (x,y) distance that an observation can move in a time period T, the benchmark models this situation by specifying that two observations B1 and B2 at times N1 * T and N2 * T with centers (C1x, C1y) and (C2x, C2y) cannot be in the same group unless

$\sqrt{(C2x - C1x)^2 + (C2y - C1y)^2} < D3 \times (N2 - N1)$.
Hence D3 is the maximum allowable velocity in time period T for the object described by the group of observations. In other words, D3 is the maximum number of array cells in the world coordinate system that an object can move in time T. Again, the value of D3 will alter the complexity of the benchmark and is discussed further in Section 2.6.

Notice that an observation can belong to multiple groups. Also, if an observation is not put in any group, then it begins a new group as a singleton.

Although F2 is defined on pairs of observations, it may be more efficient to have F2 operate in "batch" mode on a collection of observations. Therefore, a user may wrap F2 in a function, F2', that operates on a batch of observations.

2.4 Queries on the Data

In addition to cooking and grouping, the benchmark includes a total of 9 queries in three categories:

- Queries and recooking on the raw data (**Q1–Q3**)
- Queries on the observation data (**Q4–Q6**)
- Queries on the observation groups (**Q7–Q9**)

All queries are described in terms of a slab size and a slab starting point, which together specify a rectangular region of the coordinate space to operate on. The starting point [X1, Y1] is given in the local coordinate space for the first three queries. In other queries the starting point [X2, Y2] is in the world coordinate space. Here, 80% of the time it is uniformly distributed in the "dense" central area, and 20% of the time it is uniformly distributed in the rest of the space. A function F3 on the project web site will return starting points with the correct distribution. A slab has a constant size of [U1, V1], and the choice of this pair of numbers is discussed in Section 2.6. We will also use a set of parameters D1,. . . ,D6, and the pairs [U2, V2] and [U3, V3] as a way to control the complexity of the benchmark. Common settings are described later.

Q1 aggregation: For the 20 images in each cycle (as defined in Section 2.1) and for a slab of size [U1, V1] in the local coordinate space, starting at [X1, Y1], compute the average value of Vi for a random value of i. This might simulate, for example, finding the average background noise in the raw imagery.

Q2 recooking: For a slab of size [U1, V1] starting at [X1, Y1], recook the raw imagery for the first image in the cycle with a different clustering function F1', which has the same distance and scaling properties as F1. F1' is provided on the project web site.

Q3 regridding: For a slab of size [U1, V1] starting at [X1, Y1], regrid the raw data for the images in the cycle, such that the cells collapse 10:3. All Vi values in the raw data must be regridded in this process. Gridding of the raw data values is performed by an interpolation function, I1, also provided on the project web site.

Q4 aggregation: For the observations in the cycle with centers in a slab of size [U2, V2] starting at [X2, Y2] in the world coordinate space, compute the average value of attribute Oi, for a randomly chosen i.

Q5 polygons: For the observations in the cycle and for a slab of size [U2, V2] starting at [X2, Y2] in the world coordinate space, compute the observations whose polygons overlap the slab.

Q6 density: For the observations in the cycle and for a slab of size [U2, V2] starting at [X2, Y2] in the world coordinate space, group the observations spatially into D4 by D4 tiles, where each tile may be located at any integral coordinates within the slab. Find the tiles containing more than D5 observations.

Q7 centroid: Find each group whose center falls in the slab of size [U2, V2] starting at [X2, Y2] in the world coordinate space at any time t. The center is defined to be the average value of the centers recorded for all the observations in the group.

Q8 trajectory: Define the trajectory to be the sequence of centers of the observations in an observation group. For each trajectory that intersects a slab of size [U3, V3] starting at [X2, Y2] in the world coordinate space, produce the raw data for a D6 by D6 tile centered on each center for all images that intersect the slab.

Q9 trajectory-2: Consider an alternate definition of trajectory to be the sequence of polygons that correspond to the boundary of the observation group. For a slab [U3, V3] starting at [X2, Y2], find the groups whose trajectory overlaps the slab at some time t and produce the raw data for a D6 by D6 tile centered on each center for all images that intersect the slab.

2.5 Running the Benchmark

The benchmark should perform the following process for 8 (small), 20 (normal) or 50 (large) cycles, depending on the data size:

1. Ingest 20 new raw data sets produced by running the data generator.
2. Cook each of the data sets by computing observation data for the 20 new data sets and then perform grouping on the observations.

Total elapsed time should be recorded for all 8, 20 or 50 cycles. This is the *loading/cooking* performance result of a benchmark run.

When all cycles are loaded and cooked, the benchmark also runs the benchmark query tasks indicated in the previous section with randomly chosen query parameters a total of 15 times. The total time of the 15 runs is the *querying* performance result of the benchmark run.

2.6 Benchmark Parameters

The difficulty of the benchmark can be adjusted by making the slabs either larger or smaller. In addition, the difficulty can also be adjusted by increasing or decreasing the other benchmark parameters. Therefore, we specify the following parameter configurations:

	D1	D2	D3	D4	D5	D6	U1,V1	U2,V2	U3,V3
Easy	25	25	0.05	50	10	50	3750	5000	500
Medium	50	50	0.1	100	20	100	7500	10000	1000
Hard	100	100	0.2	200	40	200	15000	20000	2000

As a result, the benchmark can be run for the small, normal, or large data sets and with easy, medium, or hard parameters. In Section 5 we give results for the benchmark for the small data set with the medium parameter choices for a single computing node. We also give results for the normal/medium benchmark spread over 10 nodes. First, however, we discuss the implementation of the benchmark in SciDB and MySQL.

3. IMPLEMENTING SS-DB ON SCIDB

In this section we describe how we implemented SS-DB on SciDB. We start by giving a high-level introduction to SciDB, and then describe the arrays and structures used to implement the benchmark.

3.1 SciDB Basics

SciDB supports a multi-dimensional, nested array model. More specifically, arrays can have any number of named dimensions with contiguous integer values between 1 and N. Each combination of dimension values defines a cell. Every cell has the same data type(s) for its value(s), which are one or more scalar values, and/or one or more arrays.

Each array attribute is stored independently of the others (so-called "vertical" storage). Spatially contiguous values of the same attribute are co-located on disk. Each array is thus divided into fixed-sized multidimensional array chunks, each containing the values of one attribute for a portion of the array in the multi-dimensional space (see Figure 2). Chunks can be either dense or sparse. In dense

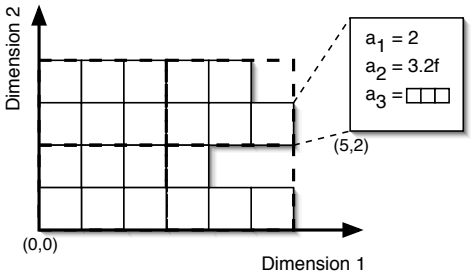

Figure 2: SciDB's nested array model and chunking model. Here, each array cell contains three attributes, an integer, a floating point number, and a 1-D nested array.

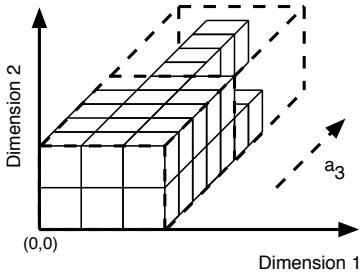

Figure 3: A series of variable-length, one-dimensional arrays nested in a bi-dimensional space.

chunks, scalar values are compactly stored in array order, without any index or dimensional information, and can be efficiently accessed by direct offsetting. Sparse chunks contain variable-length vectors of values, each preceded with dimensional information. The position of each array chunk is stored using a dedicated index. Nested arrays must all share the same dimensions and can also be co-located on disk: a series of M-dimensional arrays nested in an N-dimensional parent array is represented in SciDB as an (M+N)-dimensional array and can be chunked arbitrarily in order to regroup spatially contiguous nested values (see Figure 3). Chunks are all stored in a compressed form on disk using a variety of compression techniques (including Null-Suppression, Lempel-Ziv and Run-Length Encoding schemes) and are decompressed on the fly when being accessed. Chunk sizes are selected in order to minimize I/O costs, as in [13].

3.2 Operators and User-Defined Functions

As well as having a rich set of built-in, array-oriented operators, SciDB makes heavy use of user-defined functions (UDFs), which must be coded in C++, the implementation language of SciDB. SciDB's operators can be structural and modify the structure of the input arrays, content-dependent and based on the values of the input arrays, or both; SciDB includes a collection of basic operators built using this UDF interface, as described in [9], and that will be treated specially by the optimizer. As in Postgres, UDFs can internally call other UDFs. SciDB's queries are expressed as XML fragments containing a series of UDF instantiations. At the logical level, UDFs are defined as taking one or several arrays as input and producing one or several arrays as output. Internally, all UDFs operate at a chunk granularity: they receive one or several array chunks as input, create internal chunk iterators to consume the chunk values, and produce a series of chunks as output (see Figure 4). SciDB implements its own chunk buffer pool in order to minimize disk I/O and unnecessary compression/decompression: processed chunks are tagged with a flag indicating whether or not they should stay in main memory for further processing and are replaced whenever necessary using a clock eviction algorithm.

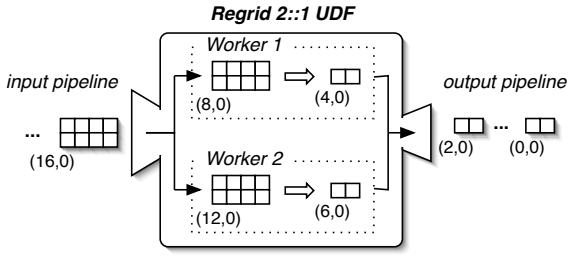

Figure 4: A simple User-Defined Function, taking one array as input and producing another array as output and compacting cells 2:1 in all dimensions inside the chunks.

Given that all arrays are physically divided into chunks and that UDFs operate at a chunk granularity, SciDB is able to automatically parallelize many UDFs by creating several instances of the UDFs (e.g., one per processor) and distributing the input chunks accordingly to keep all UDF instances busy. This can be done both locally – by creating several worker threads, each running a UDF instance – and in distributed settings by distributing the chunks over several nodes and running parallel UDFs on each node. Output arrays can then either remain scattered across several nodes or be reconstructed as one local array on one of the processing nodes. UDFs are annotated with whether or not they can be parallelized in this way since some operations (e.g., matrix inversions) cannot operate on strictly independent array partitions. For those operations, SciDB supports a set of very flexible replication mechanisms, where data can either be replicated at the chunk level or can be replicated on the fringes of the chunks in order to create overlapping array partitions (the next section gives a few examples of such overlapping partitions in the context of the benchmark).

3.3 SciDB Structures and Queries for SS-DB

Raw Data: Most SS-DB data is array-oriented and is thus a natural fit for SciDB's data model. Raw imagery data is stored as a series of dense, two-dimensional arrays. Each new image is stored in an array with a timestamp and (I,J) offset (see Figure 1). A small, sparse index is used to efficiently lookup arrays which overlay a given time range and X,Y offset. Since every array is chunked across the 2-D plane and stored very compactly on disk, retrieving contiguous array portions for Q1, Q2, Q3, Q8, and Q9 is thus very efficient and can be executed in parallel. Output arrays containing imagery data (for Q3, Q8 and Q9) are handled similarly.

Cooking cannot always be executed correctly using independent array partitions, since the raw pixels corresponding to a single observation may overlap several array chunks. Cooking images using individual chunks can thus result in an observation being missed or detected multiple times. In order to parallelize cooking while guaranteeing correct query execution, the SciDB implementation partitions the imagery arrays with an overlap D1 in both dimensions. Hence, each observation is guaranteed to be fully contained in at least one chunk (see Figure 5 for an illustration). This ensures that all chunks are self-contained and can be processed in parallel, independently of the other chunks. To avoid duplicates, we discard all observations whose center falls on an overlapping cell (our partitioning ensures that the same observation will also be detected in spatially neighboring chunks with a center falling on a non-overlapping cell). When $2D1 \geq D6$, this overlap also allows SciDB to retrieve the $D6^2$ pixels around any pixel in a chunk for Q8 and Q9 without having to retrieve the neighboring chunks (which would also be feasible but is costly in a distributed setting).

Derived Data: The cooking process produces observations, which are stored as a series of sparse 2-D arrays. As with imagery, each

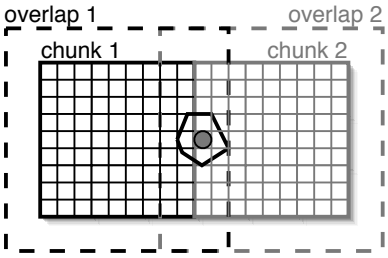

Figure 5: Two chunks with overlap; the observation in the middle of the chunks is correctly detected thanks to the overlap region and stored within the second chunk.

observation array is indexed on the time and offset of the imagery array it was derived from. Cooking produces several sparse arrays for each 2-D input array, one for each observation value (sum of pixels, average pixel distance, etc.). Observation values are stored in those arrays at the cell position corresponding to the center of the observation in the 2-D space. Hence, retrieving a series of neighboring observation values based on time/I/J intervals is very efficient. Cooking also produces a 1-D nested polygon array for each observation. Those nested arrays are also spatially co-located and stored as sparse 3-D arrays following the scheme described above in Section 3.2. We created a dedicated UDF for intersecting polygons by reimplementing the algorithm used for polygon intersection in MySQL (the *SliceScan* algorithm [2]).

Observation groups are stored as a series of sparse arrays. For each observation, we maintain a 1-D nested array to record the list of groups it belongs to. Those nested arrays are co-located on the 2-D plane as for the polygon arrays described above. The group centers are stored separately in a dedicated sparse 2-D array in order to answer Q7 efficiently. Finally, when grouping observations, we pre-compute 3-D sparse arrays materializing the trajectories of the observations in space and time in order to be able to answer Q8 and Q9 efficiently.

Distribution: In a distributed setting involving N nodes, all input array chunks are distributed in a round-robin fashion across the nodes. Each node stores thus on average 1/Nth of the chunks of each array. The queries are then parallelized by running local UDFs on the N nodes and by processing the relevant portions of the arrays locally. Query results are stored locally for cooking and queries Q2, Q3, Q5, Q6, Q7, Q8 and Q9, while they are sent to a coordinator and aggregated on a single node for the other queries. Grouping is the only global operation in the benchmark as it operates on very large portions of the observation arrays as a whole. As such, it is very difficult to parallelize, so the operation is performed on a local node in SciDB, by successively retrieving the relevant portions of the observation arrays from the other nodes. The grouping results are stored locally as well (note that the resulting arrays could also be distributed over all nodes as an alternative).

4. IMPLEMENTING SS-DB ON MYSQL

We implemented SS-DB on MySQL to see how an RDBMS would perform; in this section we describe our implementation.

4.1 Assumptions

All data and query results are stored in MySQL databases. Many projects store only the metadata in an RDBMS; in contrast, we insisted that all data reside inside MySQL. Since many scientific projects have skilled development resources, we used advanced MySQL features such as spatial indexes. In addition, we wrote a few thousand lines of C++ code with embedded SQL to implement the benchmark queries, the cooking operations, and to parallelize some queries across multiple nodes. In this endeavor, we preferred to use native MySQL functionality, even where this required a less straightforward or lower-performing design, and only used custom code when required and when it did not add an unreasonable development cost. We chose not to create code in situations where we would have had to implement substantial DBMS functionality ourselves, such as when parallelizing Q4–Q7. Lastly, we tuned our C++ code, the MySQL schema design, and the MySQL queries, using the services of a MySQL expert. In essence, we made use of the skilled resources that would be available to a large scientific research project in an effort to get the best performance out of MySQL without writing our own new database.

4.2 Tables and Indexes

There are three kinds of objects to store: raw images, observations, and groups.

We created one table for each raw image. To avoid storing one row per pixel location, we grouped pixels into square tiles of size 100 by 100. Hence, each raw image table has a tile_id, a tile_location, and a BLOB containing the raw data. We found this mixed approach of cells with small BLOBs to perform better than storing a large number of rows in the database and to be significantly simpler than a fully BLOB-based approach where database indexing and filtering could not be used.

In addition, we created an additional table that includes the name of each raw data table and the time of that table's image. This is used in cooking and grouping queries. These tables are stored in InnoDB so that we could use clustered B-tree indexes.

Observations are stored in two tables. An Observation table records each observation's unique id, observed time, and center. The ObservationPolygon table, which is used in constructing groups, lists all points in an observation and orders them. To speed up Q4–Q6, the Observation table also stores geometry objects for both the center and the polygon boundary of an observation. We used MyISAM as the storage engine for these tables to take advantage of its spatial indexing features.

Similarly, groups are stored in two tables. A Group table contains a tuple for each group as well as a GEOMETRY object to store and index the trajectory defined by the centers of all the observations in the group. Again, MyISAM storage is used to allow spatial indexes. In addition, the GroupObservation table records the mapping between groups and observations; it is stored in InnoDB.

Finally we constructed an additional table called GroupTrajectory to speed up Q9. It represents the location where an observation group exists during some time period. For example, if a group contains an observation A at time 20 and an observation B at time 30, we store a tuple (from_time=20, to_time=30, trajectory=polygons of A and B) meaning the group could exist at any time between 20 and 30 in any spatial location within the union of the polygons. We store such a tuple for each of the series of observations in each group and build a spatial index on the trajectory in MyISAM.

Finally, as an optimization for Q7–Q9, we do not physically store singleton groups in the database to make the cooking faster and the spatial index smaller. Instead, we issue another query on the observation table and mix the results with those from the group table. Because the center of a singleton group is always the same as the only observation in the group, we do not need to store it twice. Obviously, this saves space and time, but may limit flexibility when dealing with future queries not in the benchmark.

4.3 Data Loading and Cooking

We modified the benchmark-generated data to append the tile numbers for raw pixels, which we need for raw data storage. Moreover, we sorted image files by MySQL's primary key, so that its data loads are perfectly sequential and go as fast as possible.

We generated observations and their groups using the benchmark-provided code. Our C++ code retrieves and stores data in batches, using a `LOAD DATA` operation from a CSV file, both for input at the image level and for inserting observations for an image.

4.4 Partitioning and Distribution

When running on a 10 node configuration, we distributed the raw images in a round-robin fashion. This ensures skew-free partitioning and maximizes the opportunity to parallelize queries that retrieve pixel data from a number of images (e.g., Q1, Q3, Q8, Q9). In order to work with spatial queries (Q4–Q9), observation data and group data need MySQL's spatial indexes which do not support distribution. Therefore, we store these spatial indexes at a single master node, which limits their parallelizability.

4.5 Query Implementation

Q1 and Q3 (raw image queries) are easy to parallelize because all the operations occur on a per image basis. Q1 issues SQL queries to each node in parallel to compute sums and counts, and then merges these at a single node to produce the final average. Q3 simply runs the cooking and re-gridding C++ code and stores the result at each node. Since Q2 handles only one image, we do not parallelize its execution.

Q4–Q6 (observation queries) are not parallelized because all the observations and their spatial indexes are stored at the single master node. Instead, the query implementation simply uses the spatial index on the master node.

Q7–Q9 (observation group queries) are also not parallelized for this reason. However, Q8 and Q9 can parallelize the retrieval of the raw pixels at the various nodes. Specifically, we issue spatial queries in SQL at the master MySQL node to get a list of observations and their centers. We then retrieve pixels in parallel for the centers from images that intersect the slab.

As with loading, for all queries, we retrieve and store data in batches as much as possible. For example, a list of new observations is written into a CSV file first and then loaded into MySQL via a single `LOAD DATA` operation, saving the overhead of multiple query calls.

5. BENCHMARK RESULTS

In this section, we show the results of the SS-DB benchmark on our SciDB implementation and MySQL. We also analyze the performance of the two systems to characterize which features cause differences.

5.1 Setup

We ran the benchmark on our internal version of SciDB and MySQL 5.1.39 on a gigabit ethernet cluster of 10 Linux machines, each of which runs Ubuntu Karmic and has 2 GB RAM, a 1.5 TB SATA HDD and a dual core 3.2 GHz CPU.

The data sizes we use are *normal* and *small*, both with the *medium* benchmark parameters. The *small* dataset is tested on a single machine while the *normal* dataset is distributed across all 10 machines. We loaded the data onto each node, cooked the raw images to get observations and their groups, and ran the benchmark queries. Each benchmark query was executed with 5 different starting points (Xi and Yi offsets) and was run 3 times, for a total of 15 runs. All queries are cold start, meaning that we cleared the caches and restarted the database services between every run of each query. Times reported are the total runtime of all 15 runs.

5.2 Results

Table 1 shows the results of the benchmark on the two systems. The (scaleup) line shows the ratio of runtime with the *normal* and *small* datasets, in which a lower number means a better scaleup. Note that a linear scaleup would be 1.0, and a value less than 1.0 indicates super-linear scaling. Although the loading/cooking scaleup for SciDB is less than for MySQL, query runtimes scale much better in SciDB and also improve superlinearly. Only Q7 fails to show significant parallelization in SciDB (SciDB stores the array used in Q7 on a single node), whereas Q4–Q9 do not scale well in MySQL. The (MySQL/SciDB) block shows the ratio of runtime with MySQL and SciDB.

5.3 Analysis

The results above show that SciDB performs one or two orders of magnitude faster on most queries. The difference is especially high (three orders of magnitude) in Q4–Q6 with large datasets. The total runtime of the benchmark queries is about 40 times faster in the *small* dataset, 120 times faster in the *normal* dataset. Differences on the small dataset are due to architectural advantages of SciDB (compression, column-orientation, efficient representations of sparse and dense arrays), while the normal dataset is able to further take advantage of parallelization. We explain these differences in the rest of this section, starting with a discussion of parallelization.

5.4 Parallel Performance

We look at parallel performance in loading as well as in each of the three groups of queries.

Data Loading and Cooking: Both SciDB and MySQL successfully parallelize loading the raw images and detecting observations on each node. In MySQL, loading the 10x larger dataset (*normal*) took almost the same time as the smaller dataset except for a slight slowdown for observation cooking (1.8x) because observations are collected at a single master node.

SciDB's distributed data loading works as follows: it first distributes the input arrays over the nodes; then, each node partitions (i.e. chunks) its local arrays and distributes the resulting chunks over all the nodes in a round-robin fashion. Distributed loading is thus a highly parallel operation in SciDB, with many chunk exchanges between arbitrary nodes. Hence, it suffers a 2x performance overhead compared to the local loading due to the network exchanges and the asynchronous I/O operations needed to write chunks coming from other nodes.

Overall, SciDB is much faster than MySQL at loading because MySQL is quite slow at populating spatial indexes at load time (see Section 5.5).

Neither DBMS scales up well when detecting groups of observations. This is because finding groups requires global knowledge of observations (to merge into other groups) and is not amenable to parallelization.

Queries on Raw Data (Q1–Q3): In these queries, SciDB scaled up quite well and performed even faster with the larger dataset. This happens because, although the *normal* dataset has 10x larger data in total, the size of each image is only 4x larger than in the *small* dataset. Q1–Q3 process the same number of images in both settings, therefore the scaled up SciDB performs 2.5x faster (0.4x scaleup ratio) with 10x more processing power.

The same speedup occurs with MySQL, but it does not scale up for Q2 because Q2 processes only one image, and our benchmark implementation on MySQL partitions data by image while SciDB partitions images into chunks, resulting in a 630x performance difference between the two implementations on the larger dataset.

Queries on Observations (Q4–Q6): In these queries, the performance difference due to parallelization is far larger. SciDB scales up well (only 2x slowdown in the larger dataset) because it distributes observations to the 10 nodes and parallelizes the query execution. MySQL, however, does not scale up and performs two

DBMS	Dataset	Loading/Cooking [min]				Query Runtimes [min]									
		Load	Obsv	Group	Total	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Total
MySQL	*small*	760	110	2	872	123	21	393	0.4	0.36	0.6	0.6	49	50	638
	normal	770	200	90	1060	54	44	161	50	32	51	52	395	395	1234
	(scaleup)	(1.0)	(1.8)	(45)	(1.2)	(0.4)	(2.1)	(0.4)	(125)	(89)	(85)	(87)	(8.1)	(7.9)	(1.93)
SciDB	*small*	34	1.6	0.6	36	8.2	0.2	3.7	0.007	0.01	0.01	0.01	1.8	1.9	16
	normal	67	1.9	15	84	3.6	0.07	1.7	0.015	0.017	0.02	0.11	2.2	2.3	10
	(scaleup)	(2.0)	(1.2)	(25)	(2.3)	(0.4)	(0.4)	(0.4)	(2.1)	(1.7)	(2)	(11)	(1.2)	(1.2)	(0.63)
(MySQL	*small*	(22)	(69)	(3.3)	(24)	(15)	(105)	(106)	(57)	(36)	(60)	(60)	(27)	(26)	(40)
/SciDB)	*normal*	(12)	(105)	(6)	(13)	(15)	(630)	(95)	(3330)	(1880)	(2550)	(470)	(180)	(170)	(120)

Table 1: Benchmark results. (*num*) is a ratio of runtimes, either *normal* vs. *small* (scaleup) or MySQL vs. SciDB.

orders of magnitude slower on the larger dataset.

The slowdown of MySQL is even larger than the growth of the data (10x) because MySQL's spatial index performs poorly when the size of the spatial index becomes much larger than RAM. This slowdown could be avoided by distributing the spatial index to the 10 nodes, but as we noted earlier MySQL does not support a distributed spatial index. This results in the huge (three orders of magnitude) performance difference between SciDB and MySQL on the larger dataset.

Queries on Groups (Q7–Q9): Similarly, SciDB performs significantly faster in these queries because SciDB distributes all imagery and observation data at the chunk granularity over the 10 nodes and scales up nearly perfectly (only 20% slower in the larger dataset) while MySQL does not.

As described earlier, Q8 and Q9 retrieve raw pixel data for each observation that satisfies the query. MySQL parallelizes this retrieval, which results in smaller performance differences. Still, receiving the pixel data from each node and processing it on the master node introduces a CPU and I/O bottleneck in MySQL, resulting in a performance difference of about two orders of magnitude.

5.5 Other Performance Factors

So far, we have discussed the ability of SciDB and MySQL to parallelize queries over several nodes, but, as noted above, the performance difference does not come from parallelization only. SciDB performs orders of magnitude better than MySQL even on a single node for most SS-DB operations. We explain the reasons behind SciDB's superior performance below.

Columnar storage, dense-packing, and sequential I/O: SciDB adopts a very compact, array-oriented, and vertical storage model that is inherently more efficient than the N-ary storage model used by MySQL on the present benchmark. For instance, MySQL systematically reads 52 bytes worth of data for each array cell processed (2x4 bytes for the cell position plus 11x4 bytes for the 11 values contained in the cell), independently of the number of values that are really accessed in the cell. A column-oriented system reads only 12 bytes per cell and per value (2x4 bytes for the position of the cell plus 4 bytes for the value itself), while SciDB's dense array storage only reads the 4 bytes of the value itself. The situation is analogous for sparse observation arrays, except that SciDB has to store 9 bytes in addition to every cell value in that case (2x4 bytes for the cell position, and 1 byte representing the timestamp).

SciDB also limits the number of seeks required to read contiguous cell values. It seeks once per array chunk only – both when reading and writing values – and reads/writes all values in a chunk using one buffered I/O operation. In addition, the SciDB engine reads and writes chunks in the same order as they are laid down on disk whenever possible to further reduce seek times.

Array Compression: To further minimize disk transfers, SciDB stores all chunks in a compressed form on disk and decompresses them on-the-fly at access time. In the context of SS-DB, we decided to use Lempel-Ziv as the main compression algorithm, since it offers a good trade-off between compression ratios and computational complexity on SS-DB data. Lempel-Ziv typically yields a 1.9:1 compression ratio on dense SS-DB chunks. Sparse chunks get compressed with a compression ratio of 1.33:1 on average, except polygon arrays, which can be compressed with a higher compression ratio of 2:1 on average.

Local Parallelism: As noted above in Section 3.2, SciDB's UDFs take advantage of multi-core architectures: in order to speed up query execution, UDFs can create several worker threads locally and process chunks in parallel. The machines used to run the benchmark each contain a 2-core Xeon processor, registering a total of 4 cores at the OS level (2 cores plus 2 virtual cores through Intel's hyperthreading technology). The speedup gained through local parallelism is variable and depends on the computational complexity of the queries (see *Performance Drill-Down* below).

Partitioning with overlap: The performance benefits of partitioning with overlap are more difficult to assess. On one hand, overlap imposes some computational overhead to replicate some of the cells and process them in addition to the non-overlapping cells. It also imposes some storage overhead. On the other hand, overlap makes it possible to parallelize some operations that would be impossible to parallelize in distributed settings otherwise. The benefits of overlap are thus highly query-dependent and cannot be generalized. In the context of SS-DB, overlap was used to parallelize cooking and image data processing in distributed settings. Overlap incurs as much as 20% computational and storage overhead on the dense SS-DB chunks storing image data, but it allows us to parallelize cooking (e.g. for Q2) and the retrieval of neighboring raw pixels for Q8 and Q9 on a chunk basis.

Poor MySQL Index and Loading Performance: MySQL continued to exhibit poor performance at loading image data despite our efforts to speed it up. Throughput was only 3 MB/sec in reading a CSV file and writing it out to MySQL InnoDB files, far slower than the raw drive bandwidth (35 MB/sec for read+write). One reason is that MySQL InnoDB does not allow pre-allocating a large database file when each InnoDB table is created (`innodb_file_per_table`). Further, it does not allow setting the size of increase when the file needs to expand (`innodb_autoextend_increment` fixing it instead at just 4 MB for `innodb_file_per_table`), leading to slow loading for large files (> 3 GB).

Also, we observe that the build and query performance of MySQL's spatial index quickly deteriorates when the size of the spatial index file becomes much larger than RAM even though we execute `CREATE SPATIAL INDEX` after we load all tuples. We surmise that MySQL is reading and writing R-tree nodes for each tuple to build its spatial index, causing an enormous number of disk seeks, node splits, index file expansions, and substantial fragmentation unless the entire spatial index fits in main memory.

Although performance gaps caused by architectural differences will remain as described in previous sections, this poor performance of MySQL may be lessened to some degree in other RDBMSs.

	Q1		Q2		Q5	
	time	speedup	time	speedup	time	speedup
Baseline	120	-	6	-	0.02	-
Vertical Stor.	14	8.6	0.7	8.6	0.017	1.18
Compression	9	13.3	0.4	15	0.012	1.7
Parallelism	8.2	14.6	0.2	30	0.01	2

Table 2: Performance results (in minutes) for 3 queries with some of SciDB's features turned off.

Performance Drill-Down: To summarize the architecture differences, we analyzed the performance of SciDB on a subset of the queries by turning off some of its core features. Table 2 gives the execution times of three queries on the small dataset when columnar storage, compression, and local parallelism are all turned off (*Baseline*). The table then reports execution times when each of those features is turned on one after the other (*Vertical Stor.*, *Compression*, and finally *Parallelism*, the last of which corresponds to the usual SciDB with all features on). The table also gives the speedup with respect to the baseline for each configuration.

The first query we picked, Q1, is a highly I/O-bound, aggregate query on raw images. The baseline is here comparable to the performance of MySQL. Columnar storage and compression greatly improves the performance of this query (speedup of 13 altogether). Local parallelism only helps marginally since most of the time is spent in I/O transfers.

The second query we picked, Q2, is a cooking query that reads raw imagery and produces sparse observations. The baseline for this query is three times faster than its MySQL counterpart – thanks mainly to the bulk sparse array inserts in SciDB (only one seek to append a whole array, no spatial index construction). As cooking involves a series of CPU-intensive operations (object and polygon detections), local parallelism speeds up this query.

The third query we analyzed, Q5, is a spatial query on sparse observations. The baseline for this query is considerably faster than MySQL; here, the chunking mechanism of SciDB and the colocation of the sparse, nested arrays containing the polygons greatly minimizes the number of seeks required – which end up being the dominant factor for this query. The speedup gained from the other features is limited – though compression speeds up the query to a certain extent as polygon arrays tend to compress very well.

6. RELATED WORK

There have been several other efforts to build scientific database benchmarks. Sequoia 2000 [10] is a benchmark designed for GIS-style applications with some imagery but on a much smaller scale than SS-DB. The Sloan Digital Sky Survey (SDSS) [14] includes a number of benchmark queries and is based on an astronomy application related to one of the driving applications for our benchmark. These queries, however, focus only on operations that relational databases can effectively compute and as such do not include any cooking or operations on raw data. Our work on SS-DB was partly inspired by observations from scientists in projects like PanSTARRS [3], a follow-on to SDSS, who had significant additional requirements that SQLServer/SDSS does not model.

There have been several other efforts to support arrays in database systems, all of which would eventually be interesting to run on our benchmark. Greenplum [17], MonetDB [8,16], and other relational vendors have advocated storing arrays inside of the database system as tables, e.g., storing an array A[I, J] as the table T (I, J, A[I, J]). In this way, there is a row for each cell of the array. Another approach is to use BLOBs as a storage layer for array data, as was done in RasdaMan [4]. The same tactic is also used by SAP for storing financial data. In our MySQL implementation, we used a hybrid of these approaches, using small cells stored as BLOBs to reduce table cardinality but still relying on the database system to index and retrieve those cells from disk. The third approach, taken by SciDB, and several other systems [12] is to build a native array store from the ground up. As we show in this paper, this approach seems to offer substantial performance gains, at least on SS-DB, relative to the emulation-based approach.

7. CONCLUSIONS

This paper proposes SS-DB, a scientific benchmark in the form of nine queries designed to capture common computational needs from several scientific disciplines. This benchmark has been vetted by domain experts, and, as such, it is a useful tool for comparing architectural choices among data management systems.

We have begun this process by using it to compare the performance of MySQL (as a surrogate for relational systems) and SciDB (as a surrogate for purpose-built, from-the-ground-up scientific data management systems). We discovered that there are several important architectural features that any data manager for science should include. These include columnar storage, aggressive compression (e.g., eliminating the need to store array indices), a storage manager that supports tiled-chunks that can be stored contiguously and read sequentially, and the ability to support overlap across chunk borders. UDFs should also be easy to parallelize, and load time should be taken seriously by supporting things like pre-allocation of files. At present, SciDB encapsulates all of these architectural features which accounts for its impressive performance.

It is important to note that most of these features are not especially new. It is the entire package of features that is demanded by scientific applications that is important here. If one wants to seriously support these data-intensive applications, it is crucial to be able to do *all* of these things seamlessly. There may also be other features that could improve performance more, and we invite others to use this benchmark to demonstrate these improvements.

8. REFERENCES

[1] https://confluence.slac.stanford.edu/display/XLDB/SS-DB+Benchmark.
[2] http://forge.mysql.com/wiki/GIS_Functions.
[3] http://pan-starrs.ifa.hawaii.edu/public/home.html.
[4] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In SIGMOD, 1998.
[5] J. Becla and K.-T. Lim. Report from the 2nd Workshop on Extremely Large Databases. Data Science Journal, 7:196–208, 2008.
[6] J. Becla and K.-T. Lim. Report from the First Workshop on Extremely Large Databases. Data Science Journal, 7:1–13, 2008.
[7] J. Becla and K.-T. Lim. Report from the 3rd Workshop on Extremely Large Databases. Data Science Journal, 9:MR1–MR16, 2010.
[8] P. Boncz et al. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In SIGMOD, pages 479–490, 2006.
[9] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of SciDB: a science-oriented DBMS. VLDB, 2(2):1534–1537, 2009.
[10] J. Dozier, M. Stonebraker, and J. Frew. Sequoia 2000: A next-generation information system for the study of global change. In IEEE Symposium on Mass Storage Systems, pages 47–56, 1994.
[11] Z. Ivezic et al. LSST: From Science Drivers To Reference Design And Anticipated Data Products. In Bulletin of the American Astronomical Society, volume 41, page 366, 2009.
[12] A. P. Marathe et al. Query processing techniques for arrays. VLDB J., 11, 2002.
[13] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In ICDE, pages 328–336, 1994.
[14] A. S. Szalay et al. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. SIGMOD Rec., 29(2):451–462, 2000.
[15] J. A. Tyson. Large Synoptic Survey Telescope: Overview. In SPIE, 2002.
[16] A. van Ballegooij, R. Cornacchia, A. P. de Vries, and M. L. Kersten. Distribution rules for array database queries. In DEXA, 2005.
[17] F. M. Waas. Beyond conventional data warehousing – massively parallel data processing with Greenplum database. In BIRTE (Informal Proceedings), 2008.

APPENDIX

A-1. COOKING FUNCTION (F1)

```
Function F1(image):
"""
Detect pixels exceeding a threshold in an
image, grouping together adjacent over-
threshold pixels into distinct objects.

Use threshold = 1000 for F1,
threshold = 900 for F1'.
"""

objectId is an integer type
pixelObject is a type containing a list of \
  pixel coordinates and values

current is an objectId vector of size \
  image.width initialized to 0
previous is an objectId vector of size \
  image.width initialized to 0
fresh is a set of objectIds \
  initialized to empty
finalizable is a set of objectIds
  initialized to empty
obs is a set of pixelObjects indexed by \
  objectId initialized to empty
currentY is an integer initialized to 0
currentObjectId is an objectId \
  initialized to 1
width is an integer initialized to 0

for each pixel in image, starting at 0,0 \
  and incrementing by x and then y:
  if pixel.y != currentY:
    # We've moved to a new line.
    flushFinalizable()
    currentY = y
    previous = current
    set all of current to 0

  if pixel.v0 < threshold:
    # We don't meet the threshold criterion.
    return

  # Check for neighbors,
  # merging existing objects if necessary.
  leftId = leftNeighbor(pixel.x)
  if leftId != 0:
    # We have a left-hand neighbor.
    # Assume it is the best for now.
    objectId = leftId
    upId = upNeighbor(pixel.x)
    if upId != 0:
      # We also have an upper neighbor.
      # Merge both into the lower-numbered
      # object, if they differ.
      if leftId < upId:
        mergeObjects(leftId, upId)
      else if leftId > upId:
        objectId = upId
        mergeObjects(upId, leftId)
  else:
    # No left-hand neighbor.
    upId = upNeighbor(pixel.x)
    if upId != 0:
      objectId = upId
      mergeUpper(x)
    else:
      # A new object with no neighbors.
      objectId = currentObjectId
      increment currentObjectId

  # Record this pixel in its pixelObject.
  add new pixel(x,y,value) to obs[objectId]

  # Record the selected objectId in the
  # buffer and remember that this objectId
  # was updated and so is not finalizable.
  current[x] = objectId
  add objectId to fresh
  remove objectId from finalizable
##########################################
Function leftNeighbor(x):
"""
Return the left-hand neighbor objectId for
a given x coordinate, if one exists.
Return 0 otherwise.
"""

if x > 0:
  return current[x - 1]
return 0
##########################################
Function upNeighbor(x):
"""
Return an upper neighbor objectId for a
given x coordinate, from the "previous"
array, if one exists. Checks upper-left,
upper, and upper-right neighbors, returning
the smallest objectId if two exist.
Return 0 otherwise.
"""

if currentY == 0: # no previous line
  return 0
bestId = previous[x]
if x > 0:
  upperLeftId = previous[x - 1]
  if upperLeftId != 0 and \
    (bestId == 0 or upperLeftId < bestId):
    bestId = upperLeftId
if x < image.width - 1:
  upperRightId = previous[x + 1]
  if upperRightId != 0 and \
    (bestId == 0 or upperRightId < bestId):
    bestId = upperRightId
return bestId
##########################################
Function mergeUpper(x):
"""
Check for and merge two disconnected upper
objects into one with the lower objectId.
This only happens when:
  Previous: A 0 B
   Current: 0 q <-- x
```

```
where A and B are different and q is over
the threshold.  Otherwise, just return
without doing anything.
"""

if x == 0 or x == image.width - 1:
  return
leftId = previous[x - 1]
rightId = previous[x + 1]
if leftId == 0 or rightId == 0:
  return
if leftId < rightId:
  mergeObjects(leftId, rightId)
else if rightId < leftId:
  mergeObjects(rightId, leftId)
else:
  return
##########################################
Function mergeObjects(destinationId, \
  sourceId):
"""
Add the pixels of the source into the
destination and change the objectIds in the
current and previous arrays.
"""

if destinationId == sourceId:
  return

dest = obs[destinationId]
src = obs[sourceId]
for each pixel in src:
  add pixel to dest
  if pixel.y == currentY - 1:
    previous[pixel.x] = destinationId
  else if pixel.y == currentY:
    current[pixel.x] = destinationId
remove obs[sourceId] from obs
##########################################
Function computePolygon(object):
"""
Find the vertices of a bounding polygon for
a set of pixels.
"""

currentPosition = pixel coordinates with \
  the lowest y coordinate of those with \
  the lowest x coordinate
initialDirection = (-1, -1)
initialize vertices to empty

(nextPosition, direction) = \
  chooseNext(object, currentPosition, \
             initialDirection)
initialDirection = direction
add currentPosition to vertices

currentPosition = nextPosition
(nextPosition, direction) = \
  chooseNext(object, currentPosition, \
             direction)

while currentPosition != initialPosition \
  or direction != initialDirection:
  add currentPosition to vertices
  currentPosition = nextPosition
  (nextPosition, direction) = \
    chooseNext(object, currentPosition, \
               direction)

return vertices
##########################################
Function chooseNext(object, position, \
  direction):
"""
Try to trace around the object, starting
with a left turn and then steering to the
right as needed.
"""

if object has only one pixel in it:
  return (position, direction)
if dir is a diagonal direction:
  rotate direction left 90 degrees
else:
  rotate direction left 45 degrees
while coordinate position + direction is \
  not in object:
  rotate direction right 45 degrees
return (position + direction, direction)
##########################################
Function flushFinalizable():
"""
Turn pixelObjects in finalizable into
observations.  The objects that were
updated with new pixels in this line are the
ones that are potentially finalizable in the
next line.
"""

for each objectId in finalizable:
  initialize pixelSum, weightedSumX, \
    weightedSumY to 0.0
  initialize weightedDistanceSum to 0.0
  for each pixel in obs[objectId]:
    pixelSum += pixel.value
    weightedSumX += pixel.x * pixel.value
    weightedSumY += pixel.y * pixel.value
  centroidX = weightedSumX / pixelSum
  centroidY = weightedSumY / pixelSum
  for each pixel in obs[objectId]:
    distanceX = pixel.x - centroidX
    distanceY = pixel.y - centroidY
    weightedDistanceSum = pixel.value * \
      sqrt(distanceX * distanceX + \
           distanceY * distanceY)
  averageDistance = \
    weightedDistanceSum / pixelSum
  polygon = computePolygon(object)
  if polygon.edges <= D2:
    output new observation( \
      centroidX, centroidY, \
      polygon, pixelSum, averageDistance)
  remove obs[objectId] from obs
finalizable = fresh
clear fresh
```

A-2. GROUPING FUNCTION (F2)

```
Function F2(observation1, observation2):
"""
Decide if two observations belong in the
same group.
"""

distanceX = observation2.x - observation1.x
distanceY = observation2.y - observation1.y
distanceT = observation2.time - \
  observation1.time
return distanceX * distanceX + \
  distanceY * distanceY <= D3 * distanceT
```