ELSEVIER

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE @ DIRECT®

Information Sciences 159 (2004) 177–202

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

MeshSQL: the query language for simulation mesh data

Byung S. Lee [a,*], Ron Musick [b,1]

[a] *Department of Computer Science, University of Vermont, Votey Building, Burlington, VT 05405-0156, USA*
[b] *iKuni, Inc., 3400 Hillview Avenue, Palo Alto, CA 94304, USA*

Received in revised form 30 September 2002; accepted 28 February 2003

Abstract

Mesh data has been a common form of data produced and searched in scientific simulations, and has been growing rapidly in the size thanks to the increasing computing power. Today, there are visualization tools that assist scientists to explore and examine the data, but their query capabilities are limited to a small set of fixed visualization operations, which is far too short to meet the needs of most users. Thus, it is imperative to provide ad hoc query tools for them.

In this paper, we propose an ad hoc query language MeshSQL, which has been extended from ANSI SQL99 to support the features unique to simulation mesh data, such as temporality, spatial regions, statistics, and similarity. After classifying MeshSQL queries based on three criteria related to efficient implementations of the queries, we present the syntax and semantics of MeshSQL, and support them with examples. We also discuss implementing MeshSQL queries in SQL99 in an object-relational database system that allows incorporating user-defined types and functions. To our knowledge, MeshSQL is the first and the only query language for simulation mesh data.
© 2003 Elsevier Inc. All rights reserved.

Keywords: Simulation mesh data; Query language; SQL99

* Corresponding author. Fax: +1-802-656-0696.
E-mail addresses: bslee@cs.uvm.edu (B.S. Lee), musick@ikuni.com (R. Musick).
[1] Fax: +1-650-320-9827. Work done while at Lawrence Livermore National Laboratory.

0020-0255/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2003.02.002

1. Introduction

Scientific simulation is a cost-effective way of conducting a test without actually building a product, and has been used in various fields including the military defense, financial trading, manufacturing, medical imaging, oil exploration, and weather forecasting. Most scientific simulation data are produced in a mesh data format, such as NetCDF [1], HDF [2], and SILO [3]. Scientists are producing simulation data in a large scale, but have very limited tools available for exploring and querying the produced data. There are visualization tools [4,5] available today, which provide a few fixed forms of primitive query operations such as finding points, iso-surfaces, and slices. However, scientists need more powerful query tools that enable them to interactively search the data in an ad hoc manner. There has been some work done in other contexts to develop declarative, ad hoc query languages [8,10,11]. However, there is none developed for *simulation mesh data*.

In this paper we present the design and specification of a query language called MeshSQL, which is geared for querying the geometry and fields of simulation mesh data. Specifically, we characterize mesh queries as an integration of temporal queries [16], spatial region queries [8,9], statistical queries [12,18], and similarity queries [25,26]. We also classify MeshSQL queries based on three orthogonal criteria linked to the design and implementation of them.

MeshSQL is extended from SQL99 [6] for pragmatic reasons. A MeshSQL statement is written against an abstract mesh data file and is translated into an SQL99 statement or script against SQL99 tables. Because the expressive power of SQL99 is short of supporting full-fledged mesh queries, we make some, the least possible, extensions to SQL99. Currently the extensions include supporting a mesh region as the first class object, time_step as the simulation cycle, and a partitioned grouping attribute (similar to that in ODMG OQL [7]) for a grouped summary of query results. Among these extensions, the first two are implemented using the existing SQL99, and only the partitioned grouping requires a change of SQL99 itself.

In the background of MeshSQL design is the AQSim [2] project [20,21], currently in progress at Lawrence Livermore National Laboratory. AQSim's query engine is an object-relational database management system (ORDBMS), which is suitable to process mesh queries due to its capability of incorporating user-defined types, indexes, and functions. Related to this, we present our implementation ideas for MeshSQL in this paper.

The rest of this paper is organized as follows. We first present the concept and model of simulation mesh data in Section 2, the characteristics of MeshSQL in Section 3, and the classification of MeshSQL queries in Section 4. Then, we

[2] An acronym of "Approximate queries on simulation data".

present the language specification in Section 5, and follow it with examples in Section 6 and implementation considerations in Section 7. Finally, Section 8 concludes the paper. Appendix A.3.1 outlines the grammar of MeshSQL.

2. Simulation mesh data

We present the concept, mathematical model, and basic algebraic operations of simulation mesh data in this section.

2.1. Concept

Fig. 1 shows an example of mesh data generated by simulating a can crushed against a wall. It shows snapshots taken at three time steps (a.k.a. cycles) in a sequence of 44 steps. The data is defined across time steps in a temporal dimension and, at each time step, in a three-dimensional Cartesian space of spatial variables x, y, and z. There are ten "field variables"—displx, disply, displz, velx, vely, velz, acclx, accly, acclz, and eqps, among which eqps is defined at each zone, thus called "zone-centered," and the rest are defined at each node, thus called "node-centered". The first three denote the displacements in x, y, and z directions, the second three denote the velocities, and the third three denote the accelerations. The last variable eqps denotes the equivalent plastic strain, which is the measurement of strain or stress on the can surface.

Mesh data generated with simulations exhibit characteristics summarized as follows. (See [13] for a more comprehensive discussion.)

- *Regularity*—Mesh data may be regular or irregular depending on the geometric pattern of the points at which the values of fields are computed. We can distinguish between spatial regularity and temporal regularity. Spatially regular mesh data have data values computed at "a regular grid or some other geometric structure" [13]. Temporally regular mesh data have data values computed at regular time intervals. Irregular mesh data require that the coordinates of grid points—either spatial or temporal or both—must be stored together with the computed values. In contrast, regular mesh data allow the coordinates to be calculated and, therefore, not stored.
- *Time-variation*—Mesh data may be time-invariant or time-variant depending on whether the coordinates of the points change over time. Time-variant mesh is very common in the simulations of dynamic processes like deforming artifacts or changing natural phenomena. Examples are simulating a car crash, a rod bending, or a weather change. Unlike time-invariant data, time-variant data require the neighborhood relationships (i.e., topology) of mesh grid points be stored together with the coordinates (i.e., geometry) of points to allow tracing the points whose coordinates change over time.

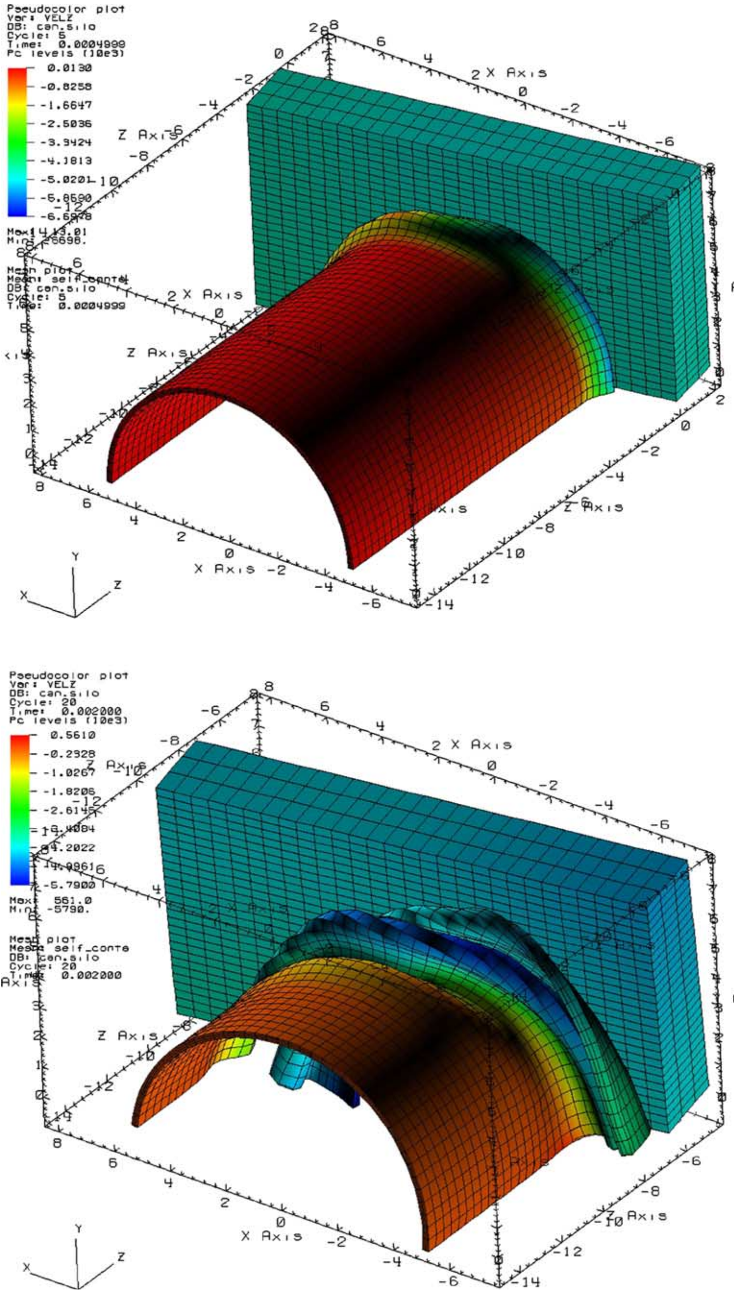

Fig. 1. Crushing can mesh data: field = VELZ at time steps 5, 20 and 35.

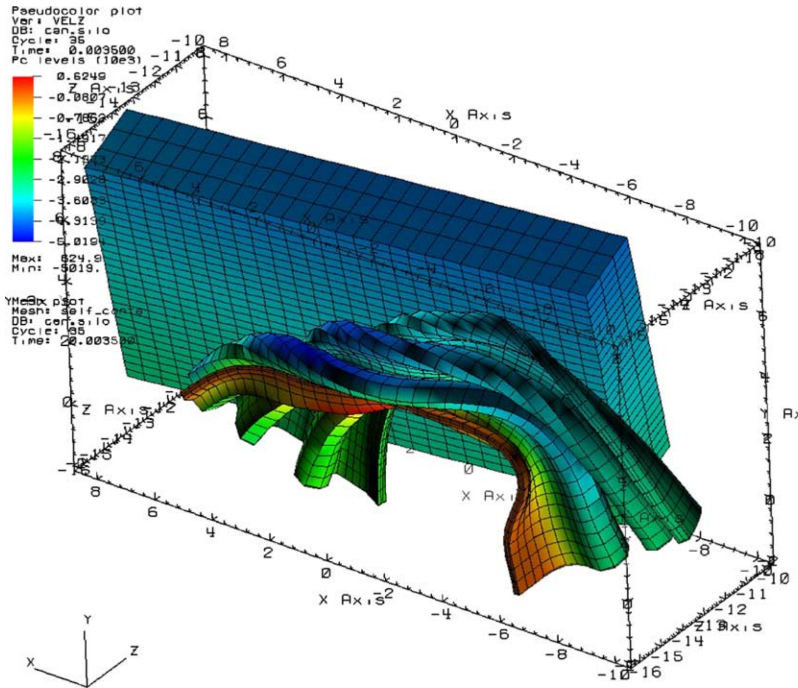

Fig. 1 (*continued*)

- *Density*—Mesh data may be dense to a varying degree depending on how many grid points are empty, that is, have no associated data. For example, a simulation of air turbulence would generate data values (e.g., velocity, orientation) at every point, therefore dense, whereas a simulation of particle collision would generate data values at only a small number of points, therefore sparse [13].

MeshSQL is applicable to any kind of mesh data described above.

2.2. Mathematical model

As indicated in [14], a rigorous mathematical model of simulation mesh data is complicated, and involves topology, geometry, and set of fields. In this section we present a simplified model that concerns only geometry and fields and still adequately supports the designed query language.

Simulation mesh data can be modeled as a discrete representation of continuous data, sampled at grid points while spanning a sequence of multiple time steps. Our mathematical model of the data is based on irregular, time-variant

data, of which regular or time-invariant data is a special case. Our model also assumes that the topological order of the points does not change over time. Then, simulation mesh data M can be defined as a set of records as follows.

$$M \equiv \{\langle t, x_1, x_2, \ldots, x_n, v_1, v_2, \ldots, v_m \rangle\} \tag{1}$$

where t denotes the time step of simulation, $x_1, x_2, \ldots, x_n$ denote n-dimensional spatial coordinate variables (where the coordinates may be calculated or stored depending on the regularity of the mesh), and $v_1, v_2, \ldots, v_m$ denote m field variables defined at each *node* located at $x_1, x_2, \ldots, x_n$ at time step t. Alternatively, a field variable may be defined at each *zone* of a mesh. In a regular mesh, a zone is an n-dimensional cubic bounded by the surrounding 2^n mesh nodes. A finite collection of contiguous nodes (or zones) defines a mesh *region*. In the rest of the paper, we consider only mesh nodes unless doing so diminishes generality. In most cases, we can either convert zone-centered fields to node-centered fields or use zone numbers in place of node indices. In case we must consider mesh zones instead of mesh nodes, we state them explicitly.

Note that, an alternate model of simulation data is a time series, which is suitable for simulation data partitioned by the time step.

$$M = \{\langle t, D_i \rangle\} \quad \text{where } D_i = \{\langle x_1, x_2, \ldots, x_n, v_1, v_2, \ldots, v_m \rangle\} \tag{2}$$

However, as argued by Davies et al. in [15], we consider time step just as another coordinate variable and, therefore, use the model in Eq. (1) while taking the time series semantics into consideration for implementing queries (in Section 7.3). This approach provides a flexibility in the query expressions supported by the model.

2.3. Basic algebraic operations

Geometrically mesh data M are configured as an $n + m + 1$ dimensional array each of whose elements is a mesh data record defined in Eq. (1). Let us consider a *node index* that determines the topological order of all mesh nodes in the data and a functional mapping (f) from a set of node indices ($\{\langle t, k_1, k_2, \ldots, k_n \rangle\}$) to a set of mesh data records (M). Then, for each mesh record $m \in M$:

$$m = f(t, k_1, k_2, \ldots, k_n) \tag{3}$$

We can rewrite Eq. (3) as follows. Let the vector $\vec{X}$ of spatiotemporal variables be:

$$\vec{X} = \begin{pmatrix} t \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \in R^{n+1} \tag{4}$$

where R refers to the domain of a real number. Additionally, let the vector $\vec{V}$ of field variables be:

$$\vec{V} = \begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix} \in R^m \tag{5}$$

and let the vector $\vec{K}$ of node indices be:

$$\vec{K} = \begin{pmatrix} k_0 \\ k_1 \\ \vdots \\ k_n \end{pmatrix} \in Z^{n+1} \tag{6}$$

where $k_0 \equiv t$ and Z refers to the domain of an integer. Then, from Eq. (3) we derive the following two functional mappings:

$$\vec{X} = \phi(\vec{K}) \tag{7}$$

$$\vec{V} = \psi(\vec{K}) \tag{8}$$

where ϕ and ψ are arbitrary functions of $\vec{K}$, respectively. In other words, at each time step, a node index functionally determines the values of the spatiotemporal variables and the field variables defined at the node. Given Eqs. (7) and (8), we define the inverse functions of ϕ and ψ as:

$$\vec{K} = \phi^{-1}(\vec{X}) \tag{9}$$

$$\vec{K} = \psi^{-1}(\vec{V}) \tag{10}$$

and introduce two basic algebraic operations on mesh data:

- Given a query condition P_X on the spatiotemporal coordinates ($\vec{X}$), find the values of the field variables ($\vec{V}$).
- Given a query condition P_V on the field variables ($\vec{V}$), find the values of the spatiotemporal variables ($\vec{X}$).

The former operation is performed by applying the function ψ to the result of $\phi^{-1}(\vec{X})$, that is, by first identifying the nodes whose spatiotemporal variables match the condition P_X (using Eq. (9)) and then retrieving the field variables of

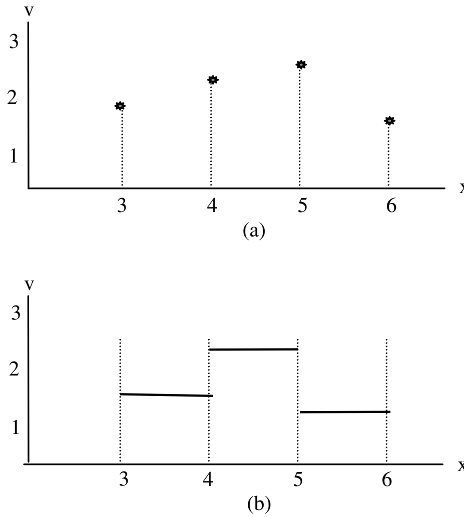

Fig. 2. Fields of a one-dimensional mesh at nodes and zones. (a) Node-centered field (applicable query conditions: $3.3 < x < 5.6$, $1.2 < v < 2.9$). (b) Zone-centered field (applicable query conditions: $x = 3.3$, $3.3 < x < 5.6$, $1.2 < v < 2.9$).

the nodes (using Eq. (8)). In contrast, the latter operation is performed by applying ϕ to the result of $\psi^{-1}(\vec{V})$, that is, by first identifying the nodes whose field variables match the condition P_V (using Eq. (10)) and then retrieving the spatial coordinates of the nodes (using Eq. (7)). Note that, as illustrated in Fig. 2, both P_X and P_V can be only range queries (i.e., $\vec{X}_0 < \vec{X} < \vec{X}_1$, $\vec{V}_0 < \vec{V} < \vec{V}_1$) if $\vec{V}$ is node-centered, whereas P_X can be a random query (i.e., $\vec{X} = \vec{X}_0$) as well if $\vec{V}$ is zone-centered.

3. Characteristics of MeshSQL

In order to design a query language, it is important to first understand what kinds of queries are likely to be addressed to mesh data. Our investigation characterizes MeshSQL queries as a combination of temporal queries, spatial region queries, statistical queries, and similarity queries.

For data generated in a sequence of time steps, MeshSQL queries are *temporal queries* [16]. Scientists are interested in querying mesh data during a time interval as well as at individual time points. As a temporal query, a MeshSQL query deals with simulation mesh data as time series data, and supports operations like temporal aggregations and temporal groupings. The temporal domain of simulation mesh data is the time step, which is associated with real time scale that ranges from microseconds (e.g., in the simulation of a car crash) to days or weeks (e.g., in the simulation of meteorological changes).

Scientists may ask queries based on either time step or real-scale time. Therefore, the time series should support a dual calendar. (A calendar allows for using application-dependent semantics of time.) For simplicity, we consider only the time step as the single calendar in this paper.

Some MeshSQL queries are *spatial region queries* [8,9]. Scientists are often interested in finding mesh regions that show a certain phenomenon or behavior of their interest. A region is basically a set of connected mesh nodes or zones, which can be displayed on a visualization tool. For this purpose, we can treat a mesh region as an instance of the abstract data type region with such an attribute as the set of the indices of nodes in a region. Instances of the type region can be stored persistently in the database and retrieved by subsequent queries. The keyword "into database" introduced in Section 5.2 is an instruction for it.

Most MeshSQL queries are *statistical queries* [12,18]. Scientists want a *summary* (e.g., aggregates per group) of field variables rather than their individual values for a given region. The summary may involve spatial and temporal variables as well as field variables. An example is "the average temperature for each group of pressure partitioned by the pressure intervals 0–1200, 1200–1800, and 1800–2400". This summary is a spatial summary if the average is taken over all selected mesh regions at each time instant, a temporal summary if taken over a time interval at each selected mesh region, and a spatiotemporal summary if taken over all selected mesh regions and time interval together. Summary data (a.k.a. "summary table") has been the subject of research in the statistical database field over a decade [12,22,24] and recently refocused in the context of OLAP databases [19,23]. We embody the concept of summary data by slightly modifying the conventional group-by clause of SQL language, as described in Section 5.2.4.

Some MeshSQL queries are *similarity queries* (a.k.a. proximity queries) [25,26]. Scientists sometimes want to identify regions that are "similar to" a particular region based on a certain distance metric. The particular region may be the result of a previous query. As a similarity query, a MeshSQL query retrieves the mesh regions that are within the specified distance from a given mesh region. The distance is calculated with a user-defined function, which often utilizes data mining techniques. The most popularly used similarity queries are *range queries* and *nearest neighbor queries* [25]. A range query finds all mesh regions that are within a certain distance from a given region, whereas a k-nearest neighbor query finds the first k nearest regions.

4. Classification of MeshSQL queries

Based on the query characteristics described in the previous section and the implementation considerations discussed in Section 7.3, we have identified five

alternative criteria to classifying queries on simulation mesh data. Let us list in this section each classification criterion and discuss individual types in each criterion.

Query target forms and query conditions

- Type *SR* for a query that retrieves the summaries of variables given the specification of a mesh region. A mesh region is specified with a predicate on spatial and temporal variables only without involving field variables. Specifically, a temporal variable is used to identify the time steps, and spatial variables are used to identify the spatial bounding regions.
- Type *RP* for a query that retrieves mesh regions given a predicate condition on variables. The result of a query in this case is the set of selected regions. The query predicate may involve field variables as well as spatial and temporal variables.
- Type *SP* for a query that retrieves the summaries of variables given a predicate condition on variables. Note that SP is equivalent to RP followed by SR. That is, $Q_{\mathrm{SP}}(\text{mesh data}) \equiv Q_{\mathrm{SR}}(Q_{\mathrm{RP}}(\text{meshdata}))$ where Q_{SP}, Q_{SR}, and Q_{RP} denote queries of the types SP, SR, and RP, respectively.

The number of time steps in a query condition

- Type *1T* for a query that searches mesh data at a particular one time step. In this case, there cannot be any temporal dimension in the summary specified by the query. Therefore, we do not consider a 1T query a temporal query.
- Type *NT* for a query that searches mesh data across multiple time steps, which is specified as a set of time intervals. An NT query is a temporal query and returns temporally summarized data [17] as the query result.

The number of variables in a query condition

Let M denote the number of different variables specified in a query condition. Then:

- Type *NV* for a query with a *null* query condition (i.e., $M = 0$). An SQL select statement with no "where" clause is of this type.
- Type *UV* for a query with a *univariate* query condition (i.e., $M = 1$).
- Type *MV* for a query with a *multivariate* query condition (i.e., $M > 1$).

5. Specification of MeshSQL

In this section we present the specifications of MeshSQL statements for creating and querying mesh data.

5.1. Creating mesh data

Mesh data are identified with its name and registered with the information about the geometrical and field variables of the data as well as the owner, format, and location of the data files. For example:

```
create meshdata CrushingCan as (
  user = lee,
  format = silo,
  path = /usr/meshsql/data/crushingcan.dat,
  coordinates = (x double, y double, z double),
  fields = (displx double, disply double, displz double, velx double, vely
            double, velz double, acclz double, accly double, acclz double,
            eqps double),
  element_methods = ( ),
  region_methods = (some_distance(rno integer) returns double)
);
create region_method some_distance(rno integer) returns double
begin
 –Calculate and return the distance between this region and the
 –region whose region_no = rno.
 ...
end
```

This statement creates abstract mesh data named "CrushingCan." The data files are owned by the user "lee", have the format "silo", and are stored as files named "crushingcan.dat" in the directory "/usr/meshsql/data." The meshes are defined in Cartesian coordinates and have the ten fields associated with each mesh element (e.g., node). There can be two kinds of methods: element_methods applied to mesh elements and region_methods applied to mesh regions. The statement shows one region_method "some_distance". (This method is used in Section 6.3.)

5.2. Querying mesh data

Appendix A contains a summary of the grammar of MeshSQL select statements. Examples in this section are based on the crushing can data shown in Fig. 1.

select [number] [**similar**] **region** [**into database**] | select_list
from mesh_data
[**where** condition]

[**group by** grouping_list
[**having** group_selection_condition]]
[**order by** ordering_key_list]

5.2.1. Select clause

The result from a query is specified as either "*region*" or a select list. The keyword "region" denotes a set of mesh nodes or zones connected contiguously in a closed region of mesh and returned as the result of a query. The lifetime of a region is only during a session by default, but we can save regions in a database (table) by using the keywords "*into database*". The keyword "similar" indicates that the query performs a similarity search on the persistent regions.

A select list can contain a temporal variable (denoted by the keyword "*time_step*"), spatial variables, field variables, non-aggregate functions, and aggregate functions. As in SQL, an aggregate function can be on the list only with its associated grouping attribute. A function, whether aggregate or not, may be either user-defined or system-defined. We list a limited number of system-defined functions in Appendix A.3.1, leaving room for adding more functions.

5.2.2. From clause

MeshSQL is not intended for spatial or temporal *joins* between two mesh data sets because these operations are very atypical of using simulation mesh data. Thus, only one mesh data set is allowed in the "from" clause.

5.2.3. Where clause

The condition in the "where" clause is a Boolean predicate expression on the spatiotemporal and field variables, and its syntax is identical to that of ANSI SQL. Therefore, we do not elaborate on its syntax but only show some examples:

- eqps > 1.0 [Simple condition]
- eqps > any (select eqps from CrushingCan where ...) [Group comparison condition]
- eqps between 0.0 and 1.0; eqps not between 0.0 and 1.0 [Range condition]
- exists (select eqps from CrushingCan where ...); not exists (...) [Exists condition]
- eqps is null; eqps is not null [Null condition]
- displx between −2 and 0 and diply between −5 and 0 and displz between 0 and 10 and (eqps > 2.5 or eqps < 0.5) [Compound condition]

5.2.4. Group-by clause

The "group-by" clause is used to retrieve a summary of data. Specifically, its grouping list defines the partitions to which aggregate functions in the select list

are applied. The grouping attribute may be "region", "time_step", or any of the spatial and field variables. If "region" is specified, each persistent region makes one partition. If "time_step" is specified, each time interval makes one partition. If a variable is specified, each partition is defined by the initial value and the increment of the variable. In case the variable appears in both the "where" clause and the "group-by" clause, the aggregate functions are applied to an intersection of the intervals specified in the "where" clause and the partitions specified in the "group-by" clause. For example, for "where var between 10 and 20 or var between 30 and 40 group by var 12, 4", the groups considered for aggregation are in the intervals of var = [12–15], [16–19], [32–35], and [36–39].

The following examples show grouping by region, time_step, and other variables:

(*G1*) Select region, avg(eqps), stddev(eqps) from crushingCan where ... group by region; ⇒ calculates the summary (i.e., avg(eqps) and stddev(eqps)) in each persistent region (across all time steps).

(*G2*) Select region, time_step, avg(eqps), stddev(eqps) from crushingCan where time_step between 9 and 21 ... group by region, time_step 10, 1; ⇒ calculate the summary in each persistent region at each time step 10–20.

(*G3*) Select region, time_step, displx, avg(eqps), stddev(eqps) from crushingCan where time_step between 9 and 21 ... group by region, time_step 10, 1, displx −5, 1; ⇒ calculate the summary in each partition of diplx $[-5,-4)$, $[-4,-3)$, etc. in each persistent region at each time step 10–20.

(*G4*) Select time_step, avg(eqps), stddev(eqps) from crushingCan where time_step between 9 and 21 ... group by time_step 10, 4; ⇒ calculate the summary in *all selected regions* in each interval of time steps [10–13], [14–17], and [18–20].

5.2.5. Having clause

Like in ANSI SQL, the "having" clause is always preceded by a group-by clause and is used to select the groups to be retrieved as a query result. For example:

(*H1*) Select time_step, eqps, avg(velz), avg(acclz) from crushingCan where time_step between 9 and 21 ... group by time_step 10, 1, eqps 0, 0.5 having min(displz) > 0; ⇒ select a group (partitioned by eqps $\in [0, 0.5)$, $[0.5, 1.0)$, $[1.0, 1.5)$, etc. and each time step 10–20) only if min(displz), calculated over all selected mesh regions in the group, >0.

(*H2*) Select time_step, avg(velz), avg(acclz) from crushingCan where time_step between 9 and 21 ... group by time_step 0, 5 having min(displz) > 0; ⇒ select a group (partitioned by the time step to $[0,4], [5,9], \dots, [40,42]$) only if min(displz), calculated over all selected regions in the group, >0.

6. Examples of MeshSQL queries

In this section, we give some representative examples of MeshSQL queries based on the crushing can data shown in Fig. 1.

6.1. Combination of query types

MeshSQL can express any combination of the query types described in Section 4:

(*C1*) *SR.NT.MV*: Select time_step, min(velx), max(velx), min(vely), max(vely), my_aggr(eqps) from CrushingCan where time_step $\leqslant$ 190 and x between 200 and 300 and $y > 100$ and z between 50 and 80 group by time_step 0,1; $\Rightarrow$ returns the time step and spatial aggregations over the selected regions at each time step.

(*C2*) *RP.NT.MV*: Select region from CrushingCan where time_step < 100 and x between 200 and 300 and $y > 100$ and z between 50 and 80 and eqps between 0.1 and 0.5; $\Rightarrow$ returns the regions bounded by the spatial predicate condition at each time step 0–99.

(*C3*) *SP.NT.MV*: Select my_summary(square(velx-100) + square(vely-32)) from CrushingCan where x between 200 and 300 and $y > 100$ and z between 50 and 80 and ((eqps between 0.1 and 0.5) or (vely between 32 and 212)); $\Rightarrow$ returns the spatiotemporal summary calculated over all selected regions and all time steps. (All time steps are considered if there is no predicate condition on time_step.)

(*C4*) *SP.1T.MV*: Select avg(velx), max(velx) from CrushingCan where time_step = 50 and x between 200 and 300 and $y > 100$ and z between 50 and 80 and eqps between 0.1 and 0.5; $\Rightarrow$ returns the spatial summary (i.e., avg(velx) and max(velx)) in each persistent region at one time step 50.

(*C5*) *SP.NT.MV*: Select avg(velx) – min(velx) from CrushingCan where time_step between 350 and 450 and eqps between 0.1 and 0.5; $\Rightarrow$ returns the spatiotemporal summary (i.e., avg(velx) – min(velx)) calculated over all selected regions and across all time steps.

(*C6*) *SP.NT.1V*: Select avg(velx) from CrushingCan where time_step between 350 and 400; $\Rightarrow$ returns the spatiotemporal aggregation (i.e., avg(velx)) over all mesh data in the selected time interval.

6.2. Visualization queries

As mentioned in Section 1, visualization tools support primitive query operations such as finding points, iso-surfaces, and orthogonal or oblique slices. The following examples demonstrate how MeshSQL can express those visu-

alization queries. We specify an arbitrary single time step 10 in all examples. If no time step is specified, each query returns the results in all time steps, displayed in sequence or in overlays.

(*V1*) *Point*: Select velx from CrushingCan where time_step $=$ 10 and $x = 200$ and $y = 100$ and $z = 60$; $\Rightarrow$ retrieve the value of a field velx at a point defined by the three spatial coordinates x, y, and z at time step 10. The result is null if data is defined at mesh nodes (i.e., node-centered) and there is no node at the specified coordinate.

(*V2*) *Iso-surface*: Select region from CrushingCan where time_step = 10 and velx = 530; $\Rightarrow$ retrieve the mesh regions in which velx = 530 at time step 10.

(*V3*) *Orthogonal slice*: Select region from CrushingCan where time_step = 10 and ontheplane$(x, y, z, 1, 0, 0, 100)$; $\Rightarrow$ retrieve the mesh region sliced by a plane perpendicular to the x-axis at $x = 100$ at time step 10. Here, "ontheplane $(x, y, z, a_x, a_y, a_z, d)$" is a function that returns true if and only if the mesh zone containing the point at $\langle x, y, z \rangle$ intersects a plane defined by the normal vector basis $\langle a_x, a_y, a_z \rangle$ and length d of the projection line from the origin to the perpendicular drop point on the plane. Note that the following simpler statement

select region from CrushingCan where time_step = 10 and $x = 100$;

is valid only if a mesh region is a set of *zones*, not nodes, because the probability of a mesh node intersecting a plane is zero.

(*V4*) *Oblique slice*: Select region from CrushingCan where time_step = 10 and ontheplane$(x, y, z, 0.5, 0.2, 0.6, 3.5)$; $\Rightarrow$ retrieve the mesh region sliced by a plane not perpendicular to any axis.

6.3. Similarity queries

As mentioned in Section 3, a similarity query aims at finding regions within a certain distance from a reference region. In MeshSQL, only the persistent regions are considered. We show here the examples of two popularly used queries—a range query and a k-nearest neighbor query. In the following examples, some_distance is the distance function shown in Section 5.1.

(*D1*) *Range*: Select similar region from CrushingCan where time_step between 10 and 20 and x between 100 and 300 and y between 200 and 400 and some_distance(123) < 3.0; $\Rightarrow$ at each time step between 10 and 20, retrieve all persistent regions whose some_distance from the region numbered 123 is less than 3. (The notion of a region number is introduced in Section 7.1.)

(*D2*) *Nearest neighbor*: Select five similar regions into database from CrushingCan where time_step between 10 and 20 and x between 100 and 300 and y

between 200 and 400 order by some_distance(123); ⇒ at each time step between 10 and 20, retrieve the first five persistent regions ordered in an increasing order of some_distance from the region numbered 123 and store them in the database. (The keywords "into database" is included for use in Section 7.2.)

7. Implementation considerations for MeshSQL

In this section we discuss three implementation issues: mesh regions, translation of MeshSQL into SQL99, and indexes for MeshSQL query processing. As MeshSQL is for an object-relational database system, its implementation makes use of user-defined types, functions, and indexes.

7.1. Mesh regions

As mentioned in Section 3, a mesh region is defined as a set of connected mesh elements (i.e., nodes or zones) and is considered an instance of the abstract data type region. In Section 7.2.1, the type region is realized as an SQL99 type region_t, which is instantiated by the table mesh_regions. Each instance of the type region consists of a region number and the associated set of references to mesh elements, and is implemented as a composite primary key ⟨region_no, element⟩ of the table mesh_regions.

Fig. 3 illustrates generating, storing, and using mesh regions. A region query (of the type RP) applies a query condition on the columns of the table mesh_elements and retrieves regions, each of which contains references to records of the type element_t. With the keywords "into database", the elements are inserted into the table mesh_regions paired with a new region_number.

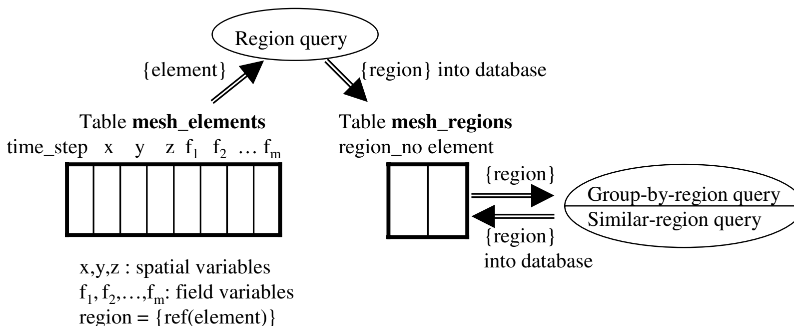

Fig. 3. The generation, storage, and usage of mesh regions.

These regions are then used as the units of grouping in the group-by-region queries or as the units of comparison in the similarity queries.

7.2. Translation from MeshSQL into SQL99

7.2.1. Creating mesh data

We show below the SQL99 types and tables created as a result of executing the "create meshdata" statement in Section 5.1. Once created, the two tables mesh_elements and mesh_regions are populated with mesh data read from the file /usr/meshsql/data/crushingcan.dat.

```
create schema CrushingCan authorization lee;
//Mesh element type (either node or zone)
create type element_t as (
  time_step integer,
  x double, y double, z double,
  displx double, disply double, displz double,
  velx double, vely double, velz double,
  acclx double, accly double, acclz double,
  eqps double);
//Mesh element table
create table CrushingCan.mesh_elements of element_t
  (ref is element_id system generated);
//Mesh region type
create type region_t as (
  region_no integer,
  element ref(element_t))
  method some_distance(rno integer) returns double;
create method some_distance(rno integer) for region_t begin ...end;
//Mesh region table
create sequence region_no;
create table CrushingCan.mesh_regions of region_t (
  primary key (region_no, element),
  scope for element_t is CrushingCan.elements)
```

7.2.2. Querying mesh data

As mentioned in Introduction, MeshSQL requires SQL99 to support a partitioned grouping. Since this features is unavailable in the current SQL99, we should simulate it by augmenting the "*where*" clause with an incremental range condition on the partitioning variables. Note this is not applicable to "group-by region" because "region" is not a variable.

For example, the "group-by time_step 10, 1" of query G2 in Section 5.2.4 is simulated as a sequence of partitioned selections on time_step as follows.

```
declare
  init1 integer = 10;
  increment1 integer = 1
begin
  while (:init1 < 21) loop
    select region, time_step, avg(eqps), stddev(eqps) from CrushingCan
    where time_step between :init1 – 1 and :init1 + :increment1 group by re-
    gion;
    :init1: =  :init1 + :increment1;
  end loop;
end
```

We can easily extend this one partitioning-variable case to a multi-variable case.

Suppose SQL99 were extended to support grouped partitioning by attributes (e.g., "group by displx, −5, 1"). Then, it is straightforward to translate MeshSQL on CrushingCan into SQL99 on mesh_elements and mesh_regions. There are three cases: select-to-select, select-to-select&insert, and select-to-script.

Case 1 (*select-to-select*): If a mesh region is specified as the target of either a qroup-by-region query or a similar-region query statement, then use mesh_regions as the target table of the corresponding SQL99 statement. The query type is RP in this case. Otherwise, use mesh_elements as the target table. The query type is either SR or SP in this case. For example, the following SQL99 statements are those translated from the corresponding MeshSQL statements in Section 5.2.

(*G3: group-by-region*): Select distinct r.region_no, r.element → time_step, r.element → displx, avg(r.element → eqps), stddev(r.element → eqps) from CrushingCan.mesh_regions r where r.element → time_step between 9 and 21 ... group by r.region_no, r.elment → time_step 10, 1, r.element → displx −5, 1;

(*G4: group-by* (*not region*)): Select e.time_step, avg(e.eqps), stddev(e.eqps) from CrusingCan.mesh_elements e where e.time_step between 9 and 21...

(*C1: SR.NT.MV*): Select e.time_step, min(e.velx), max(e.velx), min(e.vely), max(e.vely), my_aggr(e.eqps) from CrushingCan.mesh_elements e where e.time_step ⩽ 190 and e.x between 200 and 300 and e.y > 100 and e.z between 50 and 80 group by e.time_step 0,1;

(*C2: RP.NT.MV*): Select e.element_id from CrushingCan.mesh_elements e where e.time_step < 100 and e.x between 200 and 300 and e.y > 100 and e.z between 50 and 80 and e.eqps between 0.1 and 0.5;

(*C3: SP.NT.MV*): Select my_summary(square(e.velx-100) + square(e.vely-32)) from CrushingCan.mesh_elements e where e.x between 200 and 300 and e.y > 100 and e.z between 50 and 80 and ((e.eqps between 100 and 500) or (e.vely between 32 and 212));

(*V4: oblique slice*): Select e.element_id from CrushingCan.mesh_elements e where e.time_step = 10 and ontheplane(e.x, e.y, e.z, 0.5, 0.2, 0.6, 3.5);

Here, we assume the SQL99 function ontheplane has already been created: create function ontheplane(*x* double, *y* double, *z* double, *ax* double, *ay* double, *az* double, *d* double) returns Boolean begin ... end;

(*D1: range*): Select distinct r.region_no from CrushingCan.mesh_regions r where r.element ⇒ time_step between 10 and 20 and r.element → $x = 100$ and r.element → y between 200 and 400 and r.some_distance(123) < 3;

Case 2 (*select-to-select&insert*): If the keyword "into database" is specified in the select clause, insert the retrieved regions (specifically, the tuples comprising region_no and element) into the table mesh_regions. For example, the MeshSQL statement

```
select region into database from CrushingCan where...;
```

is translated into the SQL99 statement

```
insert into CrushingCan.mesh_regions
  select region_no.nextval, e.element_id from CrushingCan.mesh_elements e
  where...;
```

Case 3 (*select-to-script*): If a MeshSQL statement cannot be translated into a single SQL99 statement, it is translated into a script written in SQL99 Procedural Language Extensions. Templates are used for this purpose. For example, the similarity query D2, which retrieves five regions into the database, is translated into the following cursor program [3] that processes the regions one by one.

(*D2: k-nearest neighbors*)

```
declare
  cursor c1 is
    select distinct r.region_no, r.element from CrushingCan.mesh_regions r
    where r.element → time_step between 10 and 20
      and r.element → x = 100 and r.element → y between 200 and 400
    order by r.some_distance(123);
  c1_rec c1%rowtype;
  rno integer;
  count integer;
begin
  if not c1%open then open c1; end if
  fetch c1 into c1_rec;
  rno:= region_no.nextval;
```

[3] The syntax is based on Oracle PL/SQL. (Oracle is the trade mark of Oracle Corp.)

```
  count := 0;
  while (count < 5 and c1%found) loop
    insert into CrushingCan.mesh_regions values (rno, c1_rec.element);
    fetch c1 into c1_rec;
    count := count + 1;
    end loop;
    close c1;
end
```

7.3. Index usage for different query types

We sketch here the probable usage of database indexes based on the query classification criteria described in Section 4: $\{SR, RP, SP\} \times \{1T, NT\} \times \{NV, UV, MV\}$.

Query target forms and conditions: Query condition expressions in the "where" clause may include any of the temporal, spatial, and field variables. We do not need an index to implement ϕ in Eq. (7) and ψ in Eq. (8) because, given a node index, we can *calculate* the address of the mesh record containing the values of the variables. In contrast, indexes on spatiotemporal or field variables are useful for identifying nodes given predicates on those variables. (See Eqs. (9) and (10).)

- For an *SR* query, since the selection condition in the "where" clause includes spatiotemporal variables only, we use indexes available on the columns time_step, x, y, and z of the table mesh_elements, preferring composite (i.e. multi-column) indexes to simple (i.e., single column) indexes. The index search returns one or more sets of references to mesh_elements records (or tuple identifiers of), and each reference is de-referenced to retrieve the field values of the variables specified in the "select" clause. The values are then used to calculate the target summaries.
- For an *RP* query, since the selection condition in the "where" clause includes not only spatiotemporal but also field variables, we use indexes available on any of the columns in the table mesh_elements, again preferring composite indexes. Indexes return sets of references to mesh_elements records, and each set makes an instance of the type region. If the query is a group-by-region query or a similar-region query, then we may well create nested indexes [28] on the table mesh_regions with the columns of mesh_elements as the index keys. These indexes are currently not considered.
- For an *SP* query, which is equivalent to *RP* followed by *SR*, the sets of references to mesh_elements records are found using indexes on spatiotemporal and field variables (RP part), and the summaries are computed from the data in the dereferenced mesh_elments records.

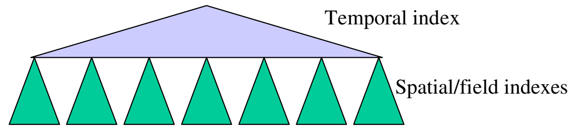

Fig. 4. A two-tiered partitioned index on simulation mesh data.

The number of time steps in a query condition: As far as physical storage is concerned, the temporal variable may well be distinguished from spatial variables. Fig. 4 shows a two-tiered index structure, where the upper level is a one-dimensional temporal index on time step and is used to identify a multi-dimensional spatial/field index to be searched. This two-tier index structure can be used by 1T and NT queries as follows:

- For a *1T* query, search the lower level index found from the upper level index given the specified time step.
- For an *NT* query, search all lower level indexes found from the upper level index given the specified interval of time steps.

Note that, as indicated in Section 2.2, it is subject to debate whether a temporal variable should be treated distinct from spatial variables [15]. Thus, we include the option of supporting one index on the composition of time_step and the coordinate columns (i.e., x, y, z) of the table mesh_elements. Such an index covers spatial data across a range of time steps with *one* index search.

The number of variables in a query condition:

- For an *NV* query, obviously no index is applicable. The only possibility is an exhaustive scan of the data (at one or multiple steps depending on 1T or NT).
- For a *UV* query, use an index if available on the column (of the table mesh_elements) corresponding to the variable. Otherwise, perform an exhaustive scan of the data.
- For an *MV* query, use a composite index applicable to the largest set of columns (of the table mesh_elements) corresponding to the variables specified in the query condition. If no composite index is available, use as many simple indexes as available. Otherwise, perform an exhaustive scan of the data.

8. Conclusion

We have presented a query language MeshSQL proposed for simulation mesh data. Under a mathematical data model simplified to a set of records

containing spatiotemporal and field variables, MeshSQL is founded upon two algebraic operators: finding the values of fields given a predicate on spatiotemporal coordinates and finding the values of spatiotemporal variables given a predicate on field variables. Due to the nature of the data, MeshSQL exhibits the characteristics of temporal, spatial region, statistical, and similarity queries. In consideration for these characteristics and the implementations of queries, we have classified MeshSQL into three categories: query target and condition forms, the number of time steps in a query condition, and the number of variables in a query condition.

We have provided the mesh data creation specification and the query specification of MeshSQL with a focus on the extensions required of SQL99. Then, we have shown examples of queries combining the classified types, queries embodying conventional visualization operations, and queries that incorporate similarity search techniques in the form of user-defined functions.

For implementing MeshSQL in SQL99, we have considered three cases that, respectively, handle translating one MeshSQL select statement into one SQL99 select statement, one MeshSQL select into one SQL99 select followed by insert, and one MeshSQL select into an SQL99 Procedural Language script. In addition, we have outlines useful indexes on mesh data and discussed how a query processor should choose those indexes for each query type in each of the three categories of MeshSQL.

Acknowledgements

The authors thank Robert Snapp at the University of Vermont and Roy Kamimura at the Lawrence Livermore National Laboratory for their comments on the draft of the language design, and the anonymous referees for their invaluable comments for improving the initial manuscript of the paper. This research was partially supported by the University of Vermont through UCRS Faculty Research Grant No. PSCI00-1 and the US Department of Energy through Grant No. DE-FG02-ER45962. For LLNL authors, work was performed under the auspices of the US DOE by LLNL under contract No. W-7405-ENG-48.

Appendix A. Grammar of MeshSQL select statement

In this appendix we summarize the grammar of MeshSQL select statements described in Section 5.2. In its current shape, MeshSQL grammar is a subset of SQL99 grammar, augmented with the features of queries characterized in Section 3.

A.1. Syntactic notations

In the syntactic rules described below, we use the following notations:

- Keywords are shown in bold face.
- An optional expression is enclosed within square brackets as in [expression].
- An expression repeated zero or more times is denoted as [expression]*.
- Alternative expressions are separated by a vertical bar (|) between them.

A.2. Terminal expressions

We assume the following expressions are given.

A.2.1. Constants

- integer: denotes a constant integer number.
- real: denotes a constant real number.
- string: denotes a constant character string.

A.2.2. Variables

- spatial_variable: a variable denoting a spatial coordinate (e.g., x, y, z) of mesh.
- temporal_variable: the time step of simulation, denoted by the keyword **time_step**.
- field_variable: a field variable defined at each mesh node or zone.

A.2.3. User-defined functions

- UD_nonaggr_function: a user-defined non-aggregate function.
- UD_aggr_function: a user-defined aggregate function.

A.3. Query expressions

We describe the grammar of a query expression using an abbreviated Backus–Naur Form.

query ::=
select [number] [**similar**] **region** [**into database**] | select_list
from mesh_data
[**where** condition]

[**group by** grouping_list
[**having** group_selection_condition]]
[**order by** ordering_key_list];

A.3.1. Select clause

select_list ::= select_element [, select_element]*
select_element ::= expr_list | aggr_function(expr_list)
expr_list ::= expr[,expr]*
expr ::= variable | number | nonaggr_function(expr_list)
variable ::= spatial_variable | field_variable | **time_step**
number ::= integer | real
aggr_function ::= avg | min | max | count | sum | stddev | UD_aggr_function
nonaggr_function ::= +| − | * | / | square | sqrt | UD_nonaggr_function

A.3.2. From clause

mesh_data ::= string

A.3.3. Where clause

The syntax of a condition expression in the "where" clause is identical to that of ANSI SQL99. Therefore, we omit the specification and refer the readers to other resources like the Ref. [27].

A.3.4. Group-by clause

grouping_list ::= grouping_attr [, grouping_attr]*
grouping_attr ::= **region** | variable init_value increment
init_value ::= number
increment ::= number

A.3.5. Having clause

group_selection_condition ::= aggr_function(expr_list) rel_op number
rel_op ::= >|>=|<|<=|==|!=

A.3.6. Order by clause

ordering_key_list ::= ordering_key [, ordering_key]*
ordering_key ::= variable | aggr_function(expr_list) | nonaggr_function (expr_list)

References

[1] NetCDF, University Corporation for Atmospheric Research, Available from: <http://www.unidata.ucar.edu/packages/netcdf/index.html>.

[2] The NCSA HDF Home Page, The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Available from: <http://hdf.ncsa.uiuc.edu/>.

[3] Silo Documentation Version 4.0, Lawrence Livermore National Laboratory, Available from: <http://www.llnl.gov/meshtv/manuals.html>.

[4] MeshTV: Scientific Visualization and Graphical Analysis Software, Available from: <http://www.llnl.gov/bdiv/meshtv/>.

[5] ENVISION: an interactive system for the management and visualization of large geophysical data sets, Available from: <http://www.atmos.uiuc.edu/envision/envision.html>.

[6] A. Eisenberg, J. Melton, SQL:1999— formerly known as SQL3, ACM SIGMOD Record 28 (1) (1999) 131–138.

[7] R. Cattell et al., The ODMG Data Standard: ODMG3.0, Morgan Kaufmann Publisher, 2000.

[8] W. Wang, J. Yang, R. Muntz, STING: A statistical information grid approach to spatial data mining, in: Proceedings of International Conference Very Large Data Bases, 1997, pp. 86–195.

[9] W. Wang, J. Yang, R. Muntz, STING+: An approach to active spatial data mining, in: Proceedings of International Conference on Data Engineering, 1999, pp. 116–125.

[10] A. Bauer, W. Lehner, The Cube-query-language (CQL) for multidimensional statistical and scientific database systems, in: Proceedings of International Conference on Database Systems for Advanced Applications, 1997, pp. 263–272.

[11] G. Wiederhold, R. Jiang, H. Garcia-Molina, An interface language for projecting alternatives in decision-making, in: Proceedings of AFCEA Database Colloquium, 1998.

[12] G. Ozsoyoglu, Z. Ozsoyoglu, Statistical database query language, IEEE Transactions on Software Engineering 11 (10) (1985) 1071–1081.

[13] A. Shoshani, F. Olken, H. Wong, Characteristics of scientific databases, in: Proceedings of International Conference on Very Large Data Bases, 1993, pp. 147–160.

[14] Scientific Data Management (SDM) Overview, Available from: <http://www.xdiv.lanl.gov/XCI/PROJECTS/SDM/>.

[15] C. Davies, B. Lazell, M. Hughes, L. Cooper, Time is just another attribute—or at least, just another dimension, in: Proceedings of International Workshop on Temporal Databases, 1995, pp. 175–193.

[16] R.T. Snodgrass (Ed.), The TSQL2 Temporal Query Language, Kluwer Academic Publishers, 1995.

[17] J.Y. Lee, R.A. Elmasri, An EER-based conceptual model and query language for time-series data, in: Proceedings of International Conference on Entity-Relationship Data Model, 1998, pp. 21–34.

[18] H. Shatkay, S.B. Zdonik, Approximate queries and representations for large data sequences, in: Proceedings of International Conference Data Engineering, 1996, pp. 536–545.

[19] S. Acharya, P.B. Gibbons, V. Poosala, S. Ramaswamy, The aqua approximate query answering system, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1999, pp. 574–576.

[20] B. Lee, R. Snapp, R. Musick, T. Critchlow, Ad hoc query support for very large scientific data: the metadata approach, in: Proceedings of Brazilian Symposium on Databases, 2001, pp. 199–212.

[21] G. Abdulla, C. Baldwin, T. Critchlow, R. Kamimura, I. Lozares, R. Musick, N. Tang, B. Lee, R. Snapp, Approximate Ad-hoc query engine for simulation data, in: Proceedings of ACM + IEEE Joint Conference on Digital Libraries, 2001, pp. 255–256.

[22] S. Abad-Mota, Approximate query processing with summary tables in statistical databases, in: Proceedings of International Conference on Extending Data Base Technology, 1992, pp. 499–515.

[23] V. Poosala, V. Ganti, Fast approximate query answering using precomputer statistics, in: Proceedings of International Conference on Data Engineering, 1999, p. 252.
[24] M.-C. Chen, L.-P. McNamee, On the data model and access method of summary data management, IEEE Transactions on Knowledge and Data Engineering 1 (4) (1989) 519–529.
[25] G. Amato, G. Mainetto, P. Savino, A query language for similarity-based retrieval of meltimedia, in: Proceedings of East-European Symposium on Advances in Databases and Information Systems, 1997, pp. 196–203.
[26] W.G. Aref, D. Barbara, S. Johnson, S. Mehrotra, Efficient processing of proximity queries for large databases, in: Proceedings of IEEE International Conference on Data Engineering, 1995, pp. 147–154.
[27] J. Melton, A.R. Simon, J. Gray, SQL: 1999—Understanding Relational Language Components, Morgan Kaufmann Publishers, 2001.
[28] E. Bertino, W. Kim, Indexing techniques for queries on nested objects, IEEE Transactions on Knowledge and Data Engineering 1 (1) (1989) 196–214.