A Scalable Database Approach to Computing Delaunay Triangulations

Tiankai Tu

CMU-CS-08-138

June 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David R. O'Hallaron, Chair
Anastassia Ailamaki
Jacobo Bielak
Todd C. Mowry
Jonathan R. Shewchuk (University of California at Berkeley)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CMS-9980063, EAR-0122464, and IIS-0429334. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: unstructured scientific data sets, tabular data representation, structural mismatch, computational cache, translation mechanism, Delaunay triangulation

Abstract

As we ride on a wave of technological innovation towards the age of petascale computing, massive input data models and voluminous output data sets involved in large-scale parallel computer simulations and scientific instruments will soon (if not already) overwhelm our ability to generate, manipulate, and interpret them. Although efficient and effective for processing relatively small data sets, traditional main-memory–based algorithms lack the necessary scalability to deal with massive data sets that require more memory than that is available. To address the new data challenge, we must seek a more scalable solution.

Inspired by the success of database management systems in managing huge information stores for the commercial and governmental sectors, we propose a database approach to computing and managing large-scale scientific data sets. Although unstructured scientific data sets do not map to the tabular representation of a database system naturally, we resolve the mismatch by introducing a *computational cache* on top of a standard database buffer pool and providing a mechanism to translate data between the inherent unstructured representation (stored in the computational cache) and the native database format (stored in database pages). Scientific application codes then operate directly on the computational cache instead of on the native database format.

We have implemented our methodology in a prototype system called *Abacus* that targets Delaunay triangulations, a commonly used unstructured data representation used by many scientific and engineering applications. Abacus can not only store and index existing Delaunay triangulations but also generate large-scale Delaunay triangulations from scratch. Evaluation results show that:

- Computing a Delaunay triangulation using Abacus is more than *three orders of magnitude faster* than an implementation that uses standard database techniques
- The performance of Abacus matches that of the state-of-the-art 2D and 3D Delaunay triangulator (*Triangle* and *Pyramid*) when triangulating data sets that fit in main memory
- Abacus achieves scalable performance even when triangulating data sets *four orders of magnitude larger* than the main memory (while other software has stopped working long ago)

Furthermore, we demonstrate the scalability of Abacus in the context of a grand challenge application—earthquake ground motion modeling, where we use Abacus to compute a series of large-scale 3D Delaunay triangulated finite element meshes with multi-billion tetrahedral elements.

Although some of the techniques presented in this dissertation are specific to Delaunay triangulations, the proposition of a database approach is more general; it points to a promising new way of how to leverage and extend existing database techniques to compute and manage large-scale unstructured scientific data sets of the coming era.

Contents

About this Thesis

The idea of using databases to support scientific applications originated when I was taking a graduate database class taught by Anastassia Ailamaki. David O'Hallaron encouraged me to extend the idea further to target dynamic datasets and to implement an unstructured octree mesh generator called *Weaver* that operates directly on databases. Weaver was used by the Carnegie Mellon Quake Project to generate record octree-based hexahedral meshes with billions of elements. The success of Weaver (which helped the Quake project to win the 2003 Gordon Bell Award for Special Achievement) made me wonder whether the idea would apply to more unstructured datasets such as Delaunay triangulations. The product of the research is this dissertation.

My thanks go to Anastassia Ailamaki, Jacobo Bielak, Guy Blelloch, Christos Faloutsos, Omar Ghattas, Mor Harchol-Balter, Benoît Hudson, Julio López, Gary Miller, Todd Mowry, David O'Hallaron, Todd Phillips, Jonathan Shewchuk, Daniel Spoonhower, and Hongfeng Yu for comments and discussions that improved this document. Among these, Jonathan must be singled out. His two high-quality software packages (available to the general public), *Robust Predicates* and *Triangle*, greatly facilitated my implementation and evaluation efforts.

Double thanks to David for his wholehearted support throughout my graduate study.

Chapter 1

Introduction

We are at an exciting juncture in the history of computing: high-performance scientific simulations are leaping towards the petascale; commodity processors are rushing to multi-core/many-core; alternative technologies such as GPGPU and Cell processors are gaining momentum. The list goes on. Yet behind the spectacle of the current wave of innovation looms a massive data avalanche. The vast quantities of data required and produced by scientific simulations and instruments will soon—if not already—overwhelm our ability to generate, manipulate, and interpret them. Database Management Systems (DBMSs), though capable of managing huge information stores for the commercial and governmental sectors, have lagged in supporting core scientific applications [1, 32]. The need to bridge the gap is urgent.

While a number of ongoing research efforts are dedicated to the management and query processing of *static* data sets, this dissertation takes a step further and targets the generation and manipulation of *dynamic* data sets. The proposed database approach and the relevant techniques introduce a new dimension to the design space of scientific data management.

This introductory chapter describes the challenges facing scientists and engineers in the new era of computing, and surveys existing solution strategies. A detailed preview of the main results of the dissertation concludes the chapter.

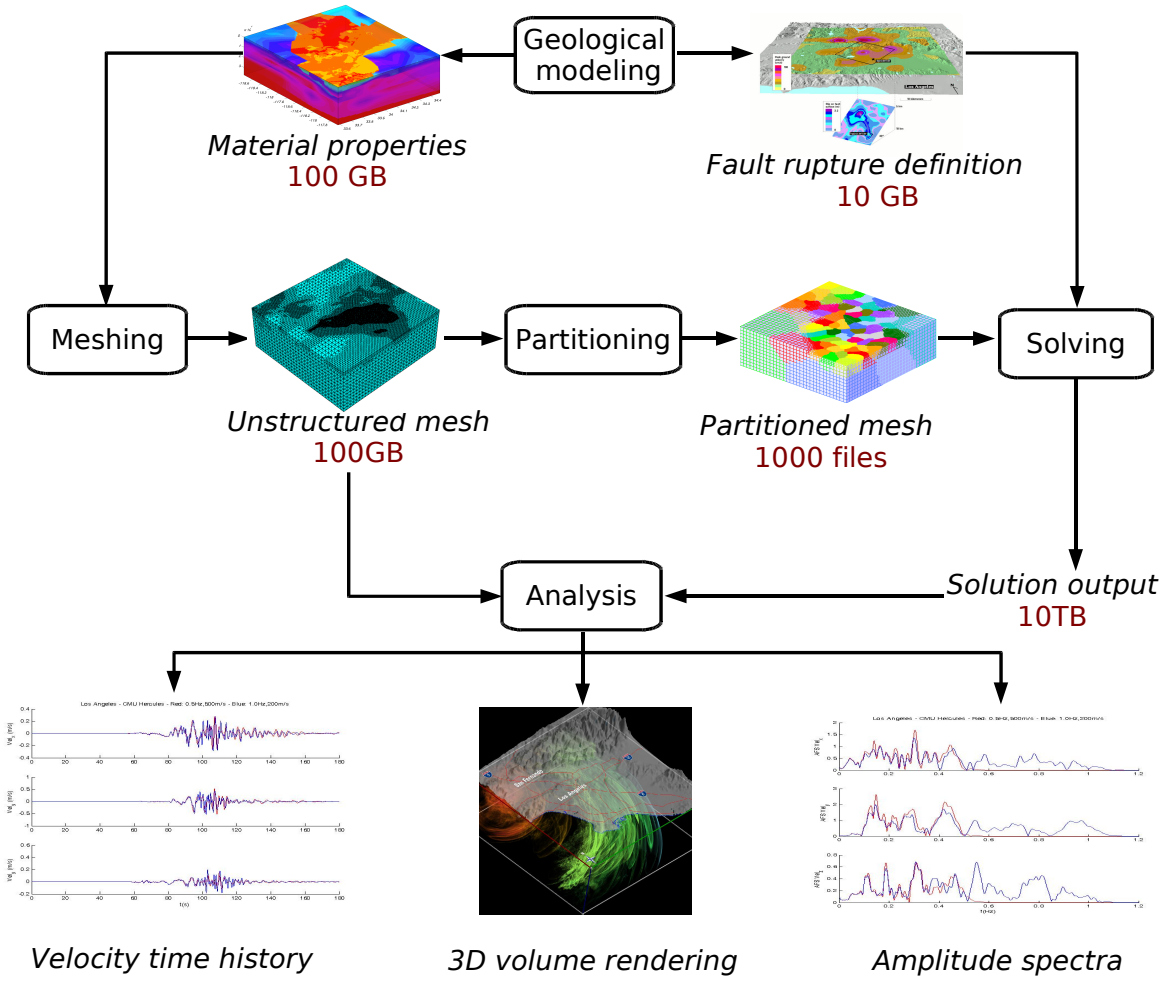


Figure 1.1: **A typical scientific simulation process.**

1.1 Data Avalanche

If Einstein would roam the campus of a university today, he would probably be surprised to find a new breed of researchers who call themselves computational scientists. Indeed, the landscape of scientific research has been significantly transformed in the past 50 years. A new form of science is emerging, thanks to rapidly-evolving, and sometimes revolutionary, computing technologies. Computational science and engineering is now widely accepted, along with theory and experiment, as a crucial third mode of scientific investigation and engineering design. Large-scale physical simulations enabled by high-performance parallel computers have led to new scientific understanding and discoveries that were previously unconceivable.

Einstein would also be amazed to learn how versatile and powerful scientific instruments have become. All disciplines, ranging from physics to astronomy to biology to earth science,

have developed innovative and high-precision new instruments to collect data and conduct experiments.

Dazzled by the overwhelming progress, Einstein would then ask how his contemporary colleagues would generate, manage, and interpret of the huge quantities of data involved in computer simulations or produced by scientific instruments. The answer, however, would not be what he would expect: even though there have been individual heroic efforts undertaken by domain scientists to address data management problems, there exists no generic solution as of today.

For a genius like Einstein, the so-called *data avalanche* [34] triggered by technology breakthroughs might not drown his thoughts on the theory of relativity. But for mortals, generating, organizing, and interpreting massive scientific data sets imposes a serious challenge.

For example, the Carnegie Mellon Quake group has been working on modeling earthquakes in large sedimentary basins on parallel supercomputers for over a decade [5, 8, 9, 12, 99]. The goal of the research is not to predict when an earthquake will occur, but instead to understand the consequences of a particular hypothetic earthquake's occurrence. For instance, if a large rupture occurs on the southern portion of the San Andreas fault in Southern California, which regions of Los Angeles will experience the most severe ground motion? Which faults and rupture scenarios will have the greatest effect on populated regions? Answers to these questions will help city planners establish building codes, structural engineers design buildings, emergency management officials prepare response plans, and insurance companies estimate potential losses.

We model seismic wave propagation in the earth via Navier's equations of elastodynamics and apply a standard Galerkin finite element method to seek the numerical solution to the equations. Figure 1.1 illustrates the process of conducting an earthquake ground motion simulation at the operational level. Given a description of the material properties of the earth, we construct a 3D seismic wavelength adaptive mesh (i.e., a graph) to resolve the local spatial resolution requirement. Roughly speaking, the softer the soil is in a particular region (e.g., the ground close to the top surface), the more mesh nodes (i.e., graph vertices) are needed to capture the behavior of the propagating seismic waves. Unknowns such as displacements and velocities are defined and associated with the mesh nodes. The mesh is partitioned into smaller pieces so that each can be loaded into the memory of a processor on a distributed-memory supercomputer (the prevailing parallel architecture today). A parallel numerical solver then takes the partitioned mesh and a postulated rupture source as input, and computes the solutions to the unknowns for each simulation timestep. Typically, the solver runs for several days to a couple of weeks on a parallel computer. During the long-running process, snapshots of the displacement and velocity fields are output to disk at prescribed intervals, for example, every 10 timesteps. After the run is completed, the 3D mesh and the time series of snapshots are queried by seismologists and civil engineers to understand the impact of the simulated earthquake.

We migrated our solver code to multi-thousand-processor terascale supercomputers five years ago in response to increasing resolution requirements, which continue to grow into the billions of elements and beyond. As a result, the problem description phase now requires generation of massive unstructured meshes on the order of hundreds of gigabytes; the output analysis phase involves manipulating and querying gigantic solution field data sets that are in the range of multi-terabyte and beyond.

At this scale, the traditional and still widely-used file-based *ftp-grep* model for data analysis breaks down. Scanning a multi-terabyte data set to locate a particular data object is time-

consuming. Moreover, the unstructured nature of the data sets often demands a more complicated data retrieval scheme than finding a single data object.

For example, retrieving a seismograph at a particular location requires (1) finding the mesh element that encloses the query location, (2) retrieving the solutions associated with the nodes of the enclosing element, and (3) interpolating the results to compute the requested seismograph. This procedure is dictated by the mathematics of the underlying numerical method of (linear) finite elements. We are not allowed to return an interpolation result of k arbitrary nearest neighboring nodes.

Figure 1.2 shows the procedure for retrieving the seismograph out of a file-based data set. First, two mesh files are consulted. The element file contains the topology of the mesh. Each element is identified by an element id and a list of 4 node ids (assuming a tetrahedral mesh is used). The node file contains the geometry of the mesh. Each node is identified by a node id and the 3D coordinate of the node. Finding the element that encloses the query location begins with a scan from the first record of the element file. For each element, the coordinates of the corresponding nodes are fetched (via random seeks) from the node file. Four computationally expensive *side-of-face* tests are conducted to determine if the element is the target. The process continues until we find the enclosing element. Using the four node ids of the hit element (and some magical way of mapping node ids to file offsets), we seek into a series of snapshot files for the field, fetch the displacement values, and compute the interpolated results at the query location.

When billions of elements and hundreds of thousands of timesteps are involved, the procedure of scanning an element file and seeking into a node file and a large number of field data files slows down to a crawl. Nevertheless, querying for points is the easy case. Much more difficult is the task of data summarization. To obtain insight, scientists need to "see" the data, for example, using 3D time-varying volume rendering visualization; "annotate" the data, for example, performing iso-contour extraction; and "convert" the data, for example, applying a Fast Fourier Transform to the entire domain. All these operations are memory intensive since large, sophisticated, incore data structures are required by the high-level algorithms. Simply scanning and seeking in flat files can hardly accomplish these tasks in a reasonable amount of time. On the hand, let us optimistically expect that memory size should keep growing according to Moore's Law, that is, doubling every 18 months. It will take 15 years for most desktop computers to have 1 terabyte memory (assuming 2 GB memory as of today). But by that time, the amount of data to be processed will be on the order of a petabyte (10^{15}) or even an exabyte (10^{18}).

Orthogonal to the difficulties associated with read-only queries, the other major challenge is how to modify the data sets on demand. For example, as more information becomes available regarding the geological structure in a particular region, part of the mesh structure needs to be refined or adjusted to incorporate the new knowledge. On these occasions, the flat files are completely useless. A new mesh has to be re-generated from scratch (a memory- and computation-intensive task that often requires the use of a parallel computer by itself), though most of the new mesh structure is identical to the old one.

The problems just presented are merely an example of the wide-spread data avalanche that has cut across all disciplines of science and engineering [11, 34, 56, 89]. The specifics of earthquake modeling are not important here. Other domains may have different material or geometry models, obey different governing equations, or use different numerical methods. But the overall

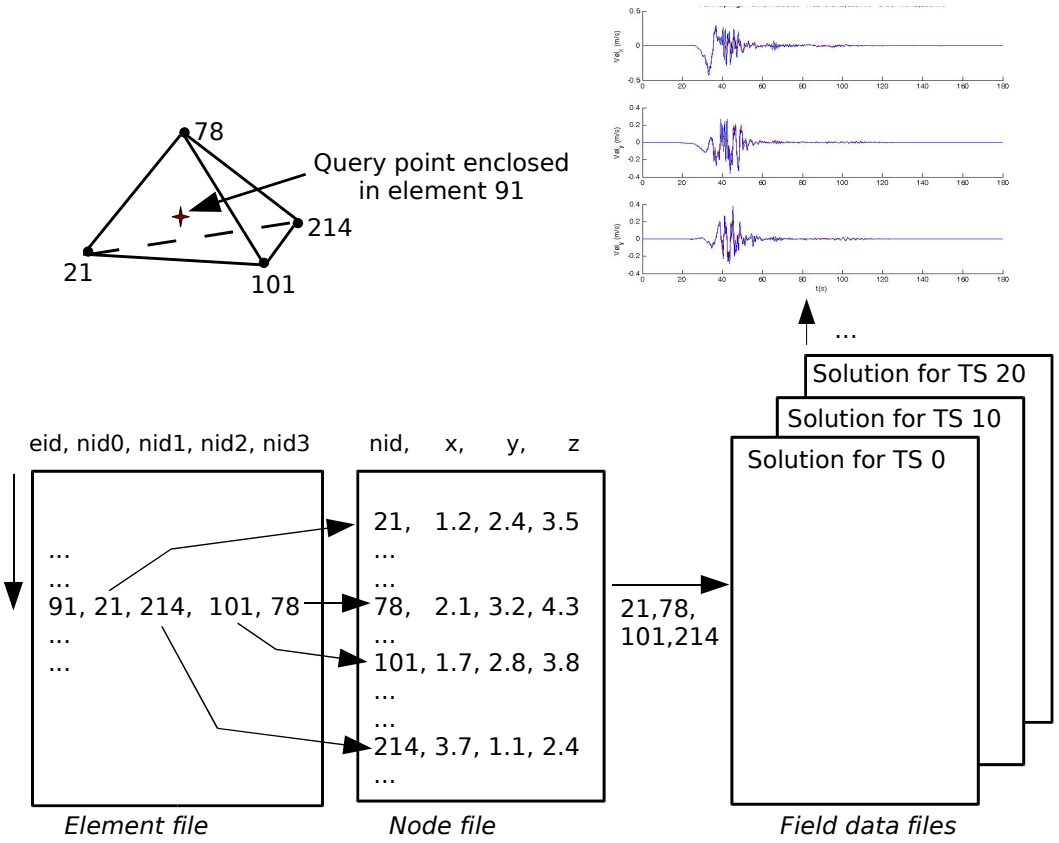


Figure 1.2: **Querying for the seismograph at a particular location.** "Eid" stands for element id and "nid" for node id. "TS" is an abbreviation for timestep. For clarity, only the enclosing element is drawn; the rest of the mesh is omitted. The seismograph at the upper-right corner is computed by interpolation of the seismographs associated with the four corresponding nodes.

simulation methodology, the scale and the complexity of the data sets, and the needs of data querying and data modification are similar and are comparable to those encountered in earthquake modeling.

1.2 Solution Approaches

Active research and software development are under way to deal with the data avalanche. A number of solutions have emerged. What follows is a brief discussion of the strengths and the weaknesses of three strategies that have shown promise and created new functionality for scientists and engineers. Note that it is not a taxonomy of all existing solutions. Rather, the classification reflects the relevance of flat files with respect to the solutions from a programmer's perspective. In order of decreasing level of relevance, the strategies are (1) *a supercomputing approach*, (2) *a portable file format approach*, and (3) *a database approach*.

1.2.1 A Supercomputing Approach

The basic idea of a supercomputing approach is to use the aggregated memory, processors, network and I/O bandwidth of a supercomputer to solve the data problem and conduct high-performance data analysis online. Depending on how the data analysis is conducted, a supercomputing approach can be further categorized as either *decoupled* or *coupled*.

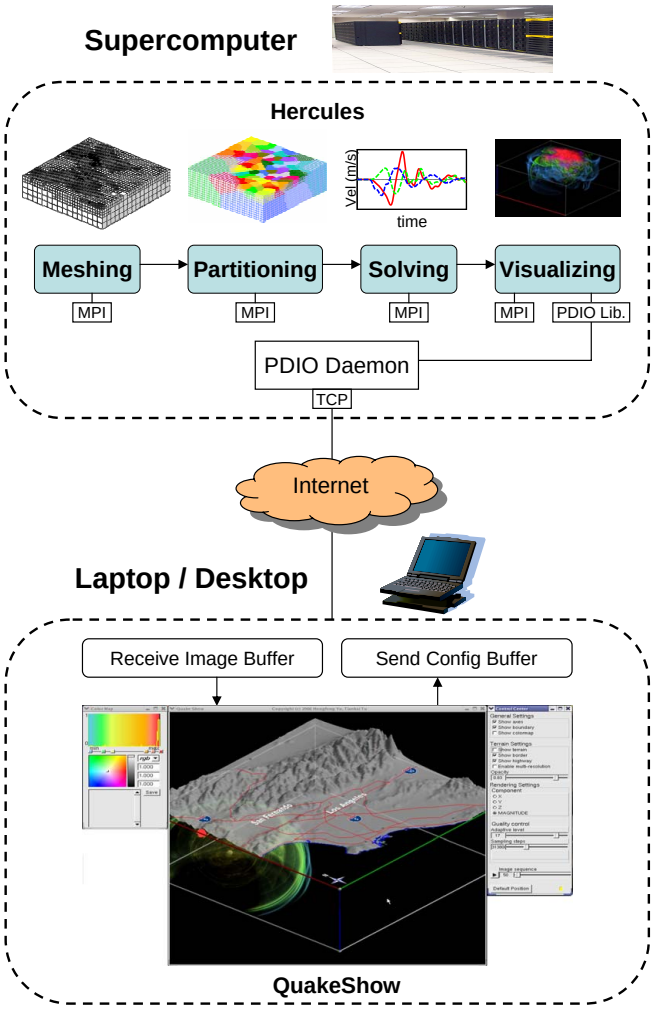


Figure 1.3: **A coupled supercomputing approach.** *Hercules* is a coupled simulation framework that executes on a multi-thousand-processor supercomputer. MPI stands for message-passing interface, the standard programming interface on parallel computers. PDIO stands for Portals Direct I/O, a special-purpose middleware infrastructure that supports external interaction with parallel programs running on Portals-enabled supercomputers such as the Cray XT3. QuakeShow is a client program that runs on a remote user's computer and communicates with a PDIO daemon using a TCP/IP connection.

A traditional supercomputing approach decouples parallel data processing and analysis from the rest of the simulation pipeline [19, 54, 55, 104].The input are flat-file data sets associated with scientific applications. The output is whatever a parallel data analysis tool is designed

to produce, which is often another massive data set. If the size of the data sets exceeds the aggregated memory of the parallel computer, sophisticated parallel I/O strategies are deployed to bulk-load the data in a certain order to avoid thrashing. In essence, a decoupled supercomputing approach attempts to solve the data problem by loading everything into memory. If that does not work, parallel out-of-core methods are used.

I have proposed and implemented a new coupled supercomputing approach [51, 97, 99] that visualizes solution data simultaneously while a parallel solver executes, as shown in Figure 1.3. The solution field is pipelined directly into the data analysis module, which uses the same processors that compute the solution. Thus, simulation results are retrieved directly from each processor's cache or main memory. Data reduction and summarization take place instantly.

A coupled supercomputing approach avoids the bottlenecks associated with transferring and storing large quantities of output data, and is particularly applicable whenever a user's ultimate interest is visualizing the 3D volume output, as opposed to retaining it for future analysis. Figure 1.3 shows an example in the context of the earthquake modeling problem where 3D volume rendering is coupled with the meshing, partitioning, and solving components. At runtime, the solution fields are visualized as they are being computed. The output JPEG images are several orders of magnitude smaller than the volume data and can be sent to a remote user via a low bandwidth TCP/IP network connection. A middleware layer (i.e., PDIO [85]) provides the infrastructure to facilitate the communication between the parallel simulation pipeline (i.e., Hercules) and the remote client (i.e., QuakeShow).

The biggest limitation of a coupled approach, however, is its inability to support postmortem data analysis. Once a simulation is completed, a user cannot conduct any further investigation into the data sets.

Both the decoupled and coupled supercomputing approaches have alleviated the data problems of scientific applications to a certain extent. But neither has addressed the fundamental issue of how to organize and index scientific data sets and provide efficient query support. After all, the carriers for the both the input and output data sets are still flat files.

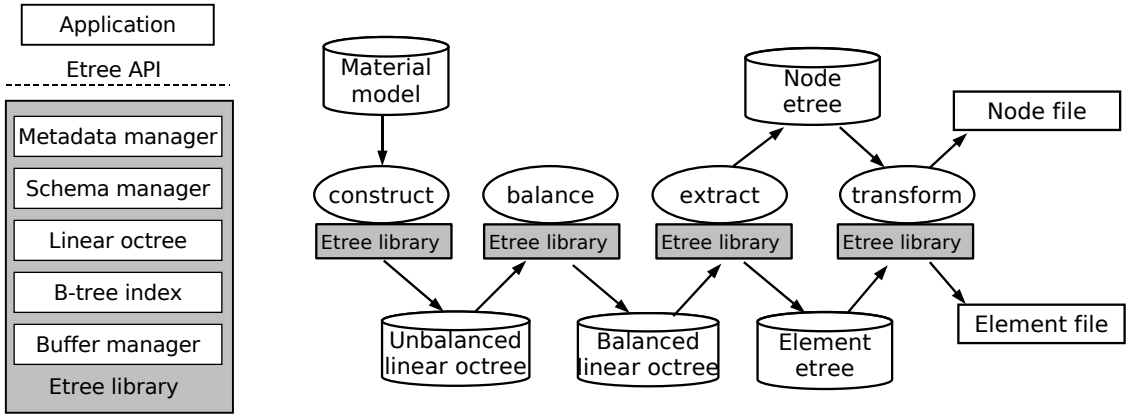
1.2.2 A Portable File Format Approach

A portable file format approach overlays a data set-specific structure on top of a flat file and provides a general purpose library to manipulate the data sets. Hence, application programs (either parallel or sequential) operate on an underlying data set via the library Application Programming Interface (API) instead of on the raw data directly.

The most commonly used portable file formats support multidimensional array data sets. Typical examples include HDF [40], NetCDF [57], and FITS [28]. These formats are different from one another and are in fact competing standards. They provide a platform-independent way of creating, modifying, and sub-setting array data sets. Generic open-source or commercial analytical/visualization tools such as AVS, MATLAB, and ParaView are able to manipulate the encapsulated data via the respective library interfaces. In addition, these standards can record data lineage and other metadata within data files, thus making it possible to share array data among scientists.

We have proposed and developed a complementary portable file format called *etree* [93] that targets octree-based data sets such as large-scale octrees or point sets embedded in a 3D

integer domain. Similar to array-oriented models, an etree allows applications to manipulate an encapsulated octree-based data set via a library API. Figure 1.4 (a) shows the components of the etree library.[1] A schema (i.e., the names and types of data fields) and other metadata (e.g., a narrative description) can be defined and stored within an etree file. The runtime ensures the portability of an etree across different platforms (e.g., little-endian vs. big-endian). Unlike other models, an etree stores an octree as a sequence of fixed-size octant records sorted by a special key called locational code, a technique generally referred to as *linear octree* [31]. The order obtained is equivalent to a preorder traversal of the tree or a Z-order traversal through the domain [27, 60, 61]. The sorted records are indexed by a conventional B-tree index and queried using fixed-length locational code keys. Spatial queries such as point in an octant and 3D range queries can be implemented conveniently on top of an etree.



(a) Etree library components. (b) Generating octree-based hexahedral meshes on etrees.

Figure 1.4: **Etree library and its application.**

Etree has been adopted by the United States Geological Survey (USGS) and the Southern California Earthquake Center (SCEC) to store the velocity models [83, 101] of the San Francisco Bay Area and the Los Angeles Basin, respectively. Thanks to the adaptivity of the underlying octree, an etree is able to represent a velocity model compactly. As a comparison, storing a velocity model using a uniformly spaced 3D grid would require 2 or 3 orders more data points.

In addition to storing and indexing static data sets such as material properties of the earth, the etree library also supports efficient dynamic updates to massive octree data sets stored on disk. We have built an octree-based hexahedral mesh generator called *Weaver* to exploit this feature [93, 94, 95, 96, 98]. Weaver has been used by the CMU Quake project to generate record sized unstructured hexahedral meshes to simulate the 1994 Northridge Earthquake in California.

Figure 1.4(b) shows the process of generating an octree mesh using the etree library. The *construct* step builds an indexed linear octree on disk. The sizes of the octants are determined by an application, for example, by the density of the material they enclose. Since only leaf octants are transformed to finite elements, the interior octants are not stored in an etree. The output is an an *unbalanced octree* because the neighbors of an octant may be arbitrarily larger or smaller than

[1]Strictly speaking, the etree library is more than a portable file format. It has features of a nascent database system such as a B-tree index structure, a page buffer manager, and a schema manager.

the octant. To guarantee good element quality, the next step performs a *balance* operation, which repeatedly decomposes large leaf octants into smaller sizes until a so-called *2-to-1 constraint* is satisfied. The 2-to-1 constraint requires that two leaf octants sharing a face or an edge be no more than twice as large or small. After a *balanced octree* is derived, an *extract* procedure is invoked to collect mesh topology and geometry information, which are stored in two different etrees, one for the mesh elements, and the other for the mesh nodes. Finally, an optional *transform* step queries the data in the mesh etrees and generates an element file and a node file as illustrated in Figure 1.2. The output flat files can be used directly by existing mesh partitioning tools [47] and solver packages [49].

The main strength (and also the limitation) of portable file formats is their optimized design and implementation for a particular type of data sets. They provide a level of *physical data independence* so that applications are isolated from the physical data layout in the files. How the data are stored on disk and how to access the data are handled by the library runtime system. On the downside, the query capabilities provided by portable file formats are quite limited. Except for extracting sub-arrays, array-oriented file formats do not support efficient temporal, spatial, and associative searches. The etree file format does support certain spatial queries, though underneath the hood it is actually using a database index structure (i.e., a B-tree) to provide the services. As such, an interesting question arises: Can we use database systems as a generic framework to solve data problems in large-scale scientific applications instead of re-inventing the wheels (i.e., access methods, buffer manager, disk space manger, etc.) for each particular type of data sets?

1.2.3 A Database Approach

The idea of using database systems to manage the scientific data avalanche is appealing. Today's DBMSs are the product of decades of innovative research and intense software development. They provide the backbone of infrastructure applications ranging from banking, retailing to supply chain management and national security. Although designed and optimized mainly for business applications, DBMSs have been used successfully to support scientific applications such as workflow management [4]. For the purpose of this dissertation, the following discussion focuses only on how database systems have been used to manage massive data sets that are acquired via scientific instruments or involved in computer simulations.

Typically, a database approach stores and indexes massive scientific data sets in a relational (or object-relational or object-oriented) database [44] and uses a data manipulation language (DML) exported by the database system such as SQL to operate and query the data. Application programs are thus completely shielded from the file abstraction.

A number of large-scale international and national data acquisition and archiving projects, including the Sloan Digital Sky Survey (astronomy) [82, 89], BaBar (high energy physics) [11, 84] and GenBank (biology) [58], have adopted a database approach. The driving motivation is to leverage existing database capability to perform queries, analyze correlations, manage metadata, and automate parallel execution. However, since most existing database systems target applications of a different nature, mapping large-scale scientific applications and data sets to databases imposes real challenges. For example, Objective/DB, an object-oriented database, is used in the Sloan Digital Sky Survey (SDSS) project to convert C++ objects to and from persistent database

structures. But its performance is disappointing, delivering only about $1/10$ of the expected speed [89]. On the other hand, loading the SDSS data into a relational database exposes the difficulties of using SQL to implement application logic. For example, there is no support for arrays and the support for user-defined types, and hierarchical data is limited. Such mismatches have made application development more difficult and the software products execute slower. In order to improve the performance, a novel automated schema design algorithm [62, 63] and a new index method [88] have been proposed.

Recently, Heber and Gray have led an effort to investigate how to support parallel finite element simulations using a relational database [41, 42, 43]. The main idea is to represent a finite element mesh and the associated solution fields using an SQL database. First, a relational schema is defined for the mesh topology (i.e., an element file) and geometry (i.e., a node file). A pre-generated mesh is then bulk-loaded into a database. Before a parallel solver starts running, each compute node queries the database to fetch its share of mesh data. During the execution of the solver, the database is not queried or updated. Output data from the solver are stored in flat files on local disks attached to compute nodes. After the solver finishes running, the solution files are copied from the compute nodes' local disks to a scratch space close to the database and then bulk loaded into the database. During the analysis phase, the database is queried to support certain types of visualization.

Note that the relational database is not tightly coupled with the simulation process. Its use is limited to the management of the finite element mesh and the solution fields *after* they are computed and stored in flat files. Besides, since no performance measurements are reported, it is unclear how efficient and scalable this approach is. Nevertheless, the work represents a significant *first* step towards supporting parallel simulation pipelines using databases.

Regardless of the context, the previous examples are pioneering works undertaken by domain scientists (with the help of computer scientists) to tackle the data problem using databases. But as a general rule, the vast majority of scientists do not use database systems in their work [32]. There is a long list of reasons why databases have not been more useful for scientific applications. For example, databases do not support important scientific data types such as arrays, meshes, and fields; it is hard to formulate sophisticated spatial-temporal queries in SQL; the performance of loading and querying data is abysmal; there are no quality visualization or analysis tools built on top of databases. To change the status quo and make databases the workhorse for scientific applications, significant research challenges must be overcome to qualitatively enhance the competence of today's database systems.

In summary, each of the three solution strategies has solved some aspect of the data problem in large-scale scientific applications. As the size and complexity of scientific data sets move into the realm of the terascale and beyond, these disparate, ad-hoc solution strategies become increasingly untenable. We need a systematic framework to attack the data problem at its core. Towards that goal, we envision a *data-centric framework* anchored on databases.

1.3 A Data-Centric Framework

Facilities that host the high-performance computers are commonly referred to as *supercomputing centers*. The term, by itself, reflects a unique mental model of the funding agencies, user groups,

and on-site hardware and software engineers: A supercomputing center is first of all a prime CPU cycle-provider; everything else comes second. In fact, crafting optimal parallel numerical solvers (the core algorithms in scientific computing) has traditionally been the primary goal of computational scientists who are the main users of supercomputing centers. Great effort has gone into the design, evaluation, and performance optimization of scalable parallel solvers. Data management and analysis have taken a back seat when it comes to attention paid to scalability and performance. But as the amount of data reaches into the terabyte to petabyte scale and the content of data becomes more unstructured and complex, this *computation-centric* view must be re-assessed and adjusted.

Outside of the scientific computing community, there is a complementary view of computing, which is also coined in the term referring to the hosting facilities—*data centers*. Enterprises rely on data centers to conduct mission critical operations such as airline ticket reservation, e-commerce, web search, and online auctions. Although the computer systems at data centers are high-end commodity clusters similar in capacity to those found at supercomputing centers, the focus here is on the data. Efficient, reliable, and secure manipulation of data is the primary goal. Processor cycles are consumed to retrieve, convert, summarize, and format the data in order to make them accessible and useful to the users.

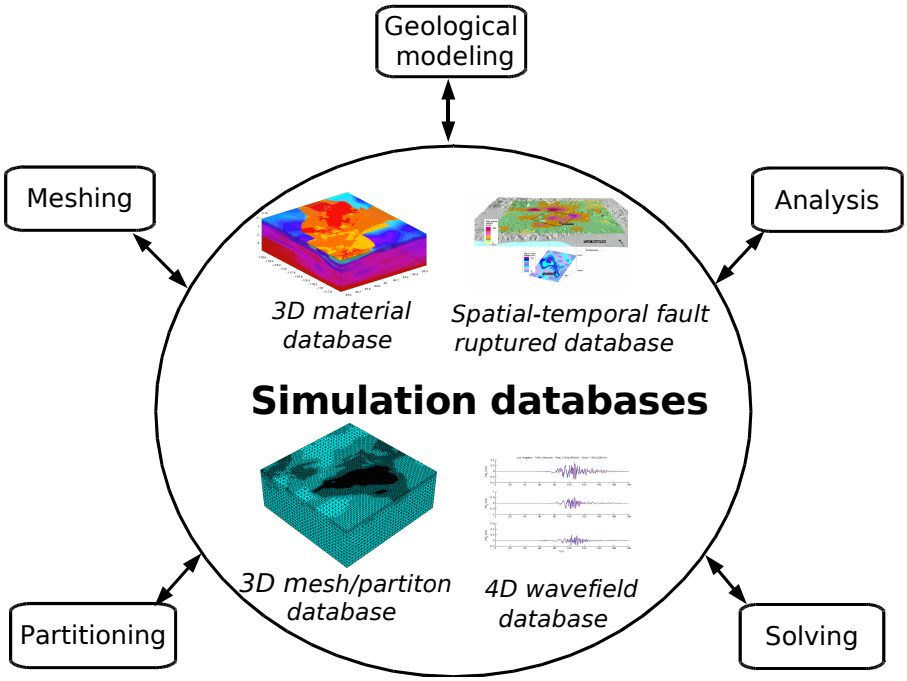


Figure 1.5: **Scientific simulation within a data-centric framework .**

Borrowing the idea from enterprise computing and using databases as the building block, we can recast the simulation pipeline (Figure 1.1) into the a *data-centric framework* as shown in Figure 1.5. The database system at the center is the hub for all data exchanges. It is also an oracle to satisfy all the data queries issued by different simulation components. No intermediary flat files or portable file formats are involved. The database is tightly coupled with meshing, partitioning, solving, and online or offline data analysis, and thus becomes an integral part of

the simulation process. Note that the data-centric framework proposed here does not require the use of a parallel supercomputer. As long as there is enough disk space to store the database, a simulation can be performed and interpreted by scientists or engineers. Of course, the more resources (i.e., larger memory and more processors), the faster the simulation and the analysis.

Similar vision has been proposed recently. Gray and colleagues have envisioned *science centers* to deal with petascale data sets produced by modern scientific instruments [32]. Bryant and colleagues have proposed *Data-Intensive Super Computer* (DISC) systems, placing emphasis on data, rather than raw computation, as the core focus of the system [15].

1.4 Thesis Statement

To build flexible, high-performance parallel simulation database systems that run on multi-thousand-processor terascale or petascale supercomputers requires innovative research in all areas within computer science. This dissertation demonstrates the scalability of a database approach to computing a particular kind of unstructured scientific data sets called Delaunay triangulations. Compared with other types of scientific data sets, Delaunay triangulations are one of the most flexible and unstructured.[2] Thanks to their superior geometry-resolving power, they are used by a large number of important applications such as finite element simulations, computer graphics, and geographic information systems.

Chapter 2 presents a case study of mapping Delaunay triangulation to database structures in a traditional way. A solution that builds a Delaunay triangulation construction algorithm on top of an R-tree index structure—a popular spatial data access method supported by a number of production database management systems—turns out to be ineffective. It runs more than three orders of magnitude slower than a quality incore Delaunay triangulator. Performance analysis shows that most of the running time is spent searching and updating the R-tree index. Further analysis reveals a *structural mismatch* between the unstructured properties of Delaunay triangulations and the built-in tabular representation of data in typical database structures such as an R-tree.

Based on this discovery, Chapter 3 proposes a new technique to make a database approach feasible for computing large-scale Delaunay triangulations. The main idea is to add a *computational cache* on top of a standard database buffer manager and provide a mechanism to translate data between the unstructured representation in the computational cache and the tabular data layout on slotted database pages. Scientific codes then operate directly on the computational cache instead of on the native storage format. This proposition diverges from the traditional translation-free buffer management scheme used by today's DBMSs. The conventional wisdom has been that such data "marshaling" and "un-marshaling" operations is computationally too costly to be practical.

Chapter 4 describes the implementation of the methodology within a prototype system called *Abacus* that is designed to deal with 2D/3D Delaunay triangulation data sets. Abacus is capable of storing and indexing pre-generated triangulations and supporting read-only applications such as iso-contour extraction and volume rendering. More importantly, Abacus has the unique ability

[2] Structured data sets (e.g., a regular grid) exhibit a uniform topological structure that unstructured data sets (e.g., an arbitrary triangulation) lack. See more details in Chapter 2.

to support dynamic data sets, generating and indexing massive Delaunay triangulations from scratch. In order to prevent other DBMS's overhead from affecting performance evaluation, I have built Abacus from a clean slate without using an existing database system. However, no special assumptions are made; almost all production DBMSs have the required software modules, such as a page buffer pool manager and a B-tree index, to support the implementation.

Chapter 5 presents the performance evaluation of Abacus. The main findings are summarized as following:

- Computing a Delaunay triangulation using Abacus is more than 5000 times faster than the implementation (of Chapter 2) that uses an R-tree, a standard spatial access method.
- The performance of Abacus matches that of *Triangle* and *Pyramid*, the state-of-the-art 2D and 3D Delaunay refinement mesh generators, [3] respectively, when triangulating data sets that fit in memory.
- When Triangle and Pyramid start thrashing (out of physical memory) and stop working (out of virtual memory), Abacus continues to achieve (almost linearly) scalable performance. For instance, it is capable of triangulating a data set of 93 GB using only 8 MB physical memory.

Besides the experimental cases, Abacus further demonstrates its scalability in the context of a grand challenge application (earthquake ground motion modeling) where it supports the generation of a series of large-scale 3D Delaunay triangulated finite element meshes with multi-billion tetrahedral elements. This is a breakthrough new capability on commodity servers, which rivals parallel algorithms running on supercomputers that have hundreds of gigabytes to terabyte aggregated physical memories.

Building on the evidence, Chapter 6 claims the intellectual and practical contributions of the research and points out how to leverage new technologies such as multi-core processors.

Together, the analysis, design, implementation, and evaluation presented in this dissertation corroborate the following thesis statement:

> Extending existing database techniques with an application-specific computational cache is a scalable solution to computing large-scale Delaunay triangulations.

[3] 2003 James Hardy Wilkinson Prize in Numerical Software.

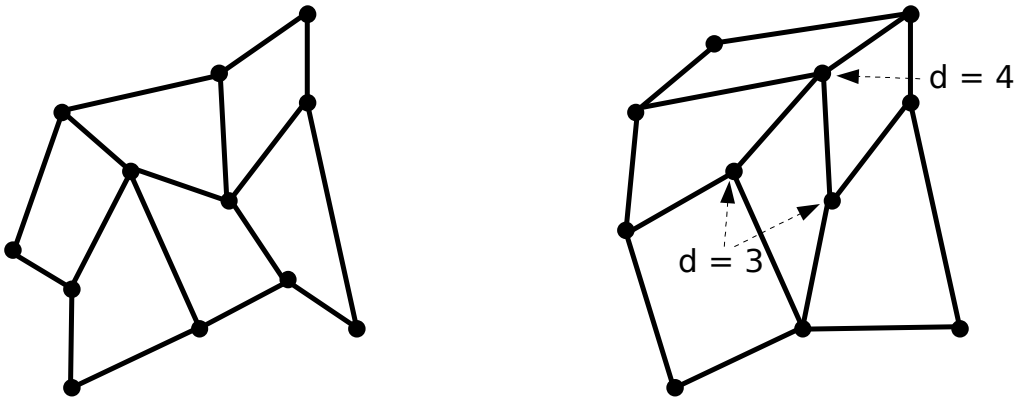
Chapter 2

Structural Mismatch

A problem that has persistently vexed database researchers is why mapping scientific applications to conventional DBMSs is so difficult. As early as the late 1980s, the term *impedance mismatch* was coined by the object-oriented database community to refer to the differences in both the programming models and the type systems between the general-purpose imperative programming language such as PL/I and the declarative databases data manipulation language such as SQL [7, 105]. Recently (*circa* 2005), researchers have extended the meaning of the term to refer to the mismatch between the programming model of scientific applications and the existing database capabilities [32, 41]. Propositions have been made, for example, putting scientific data types and programs into databases, to ameliorate the problem.

The insight of impedance mismatch is important. However, it accounts for only part of the essential hurdle between scientific applications and database systems. As will be shown in this chapter, merely integrating types and codes with data (and thus eliminating the impedance mismatch) is not sufficient to resolve the mapping problem. The main hurdle, I believe, is due to *structural mismatch*, that is, the unstructured nature of a large class of scientific data sets and algorithms do not match the rigid built-in tabular abstraction of database systems.

This chapter uses Delaunay triangulation as a running example to demonstrate that directly implementing a scientific algorithm on top of a database system is a dead end. Compared to an efficient incore implementation of the same algorithm, the database version runs three orders of magnitude slower. Performance evaluation and synthetic analysis lead to the discovery of structural mismatch as the root cause of the problem.



(a) A structured mesh. (b) An unstructured mesh.

Figure 2.1: **Structured and unstructured mesh.** The structured mesh (a) has the same topology as a uniform rectangular grid, though it is deformed enough to appear to be unstructured. The arrows point to mesh nodes of different degrees (abbreviated as “d=3” and “d=4”).

2.1 Structured and Unstructured Data Sets

There is no formal ways to define a data set as structured or unstructured. In practice, 2D/3D topological and geometric properties are often used to assess how structured a data set is.

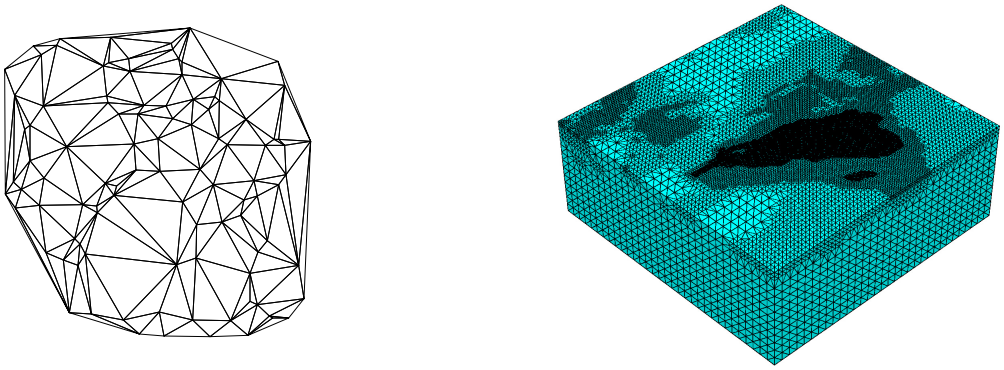
In scientific applications, a physical problem domain is usually discretized with a finite number of points and partitioned into small pieces of simple shape. Adopting the terminology in numerical simulations, we call the points *nodes* and simple shapes *elements*. Together, the nodes and elements define a mesh. Topologically, a mesh is either structured or unstructured. Figure 2.1 shows an example of each. Structured meshes exhibit a uniform topological structure that unstructured meshes lack. A functional definition [81] is that in a structured mesh, the indices of the neighbors of any node can be calculated using simplex addition, whereas an unstructured mesh necessitates the storage of a list of each node’s neighbors. Note that geometrically, neither example is structured (regular); the coordinates of the irregularly distributed nodes of both meshes have to be explicitly listed.

Under this criterion, only uniform grids are strictly structured; all others are unstructured, either topologically or geometrically or both. It turns out that the vast majority of data sets used in scientific applications are unstructured. Among them, one of the most flexible and commonly used is the Delaunay triangulation.

2.2 Delaunay Triangulation

Delaunay triangulation [24], first introduced by Delaunay in 1934, is of paramount contemporary importance in numerical analysis, computer graphic, and geographic information systems. Given a vertex set, there are numerous different ways to construct a triangulation. A Delaunay

triangulation is a special instance in which the circumcircle of every triangle is *empty*. A circumcircle is defined as empty if it does not enclose any vertices. Note that vertices are allowed on a circumcircle. This property is called the *empty circumcircle property*. Figure 2.2 shows two examples of Delaunay triangulations.



(a) A 2D Delaunay triangulation. (b) A 3D Delaunay triangulation.

Figure 2.2: **Delaunay triangulation data sets.** The interior of the 3D Delaunay triangulation is not shown in the figure.

It can be proven that a Delaunay triangulation always exists and is unique if all the vertices are in *general position*, meaning that non 4 vertices lie on a common circle. It can also be proven that a Delaunay triangulation is optimal (among all the triangulations of a vertex set) in that it maximizes the minimal angle in the triangulation [23]. Roughly speaking, most triangles in a Delaunay triangulation are well shaped. There exist fewer long, skinny triangles than any other triangulation of the same vertex set. The practical application of this optimality is to minimize numerical interpolation errors.

(a) A non-Delaunay triangulation. (b) A Delaunay triangulation.

Figure 2.3: **Triangulations of a vertex set** V_1, V_2, V_3, V_4, V_5**.** The non-Delaunay triangles $\triangle V_1V_2V_3$ and $\triangle V_1V_3V_5$ are removed by flipping the edge V_1V_3.

In 2D, when no four vertices lie on a common circle, the Delaunay triangulation is unique. (The uniqueness generalizes to higher-dimensions.) For example, Figure 2.3(a) shows a non-

Delaunay triangulation of 5 vertices where V_2 is enclosed by the circumcircle of $\triangle V_1V_3V_5$. Flipping the edge V_1V_3 to V_2V_5, we obtain a new triangulation as shown in Figure 2.3(b). The empty circumcircle property holds for every triangle in this triangulation. Hence, it is a Delaunay triangulation.

Over the years, a number of efficient computational geometry algorithms have been invented to construct 2D Delaunay triangulations including the incremental insertion [13, 50, 103], the divide-and-conquer [26, 35, 52], and the sweepline [30] algorithms. Among these, the simplest is the incremental insertion algorithm, which has the advantage of generalizing to arbitrary dimensionality. We use this algorithm to illustrate the mismatch between Delaunay triangulation and database structures.

Proposed by Bowyer [13] and Watson [103] at the same time, the incremental insertion algorithm inserts vertices into a Delaunay triangulation one at a time. For each insertion, two steps are performed to maintain the empty circumcircle property:

1. Cavity creation: Find the triangles whose circumcircles enclose the new vertex and form an insertion polygon, called a *cavity*, from these triangles.
2. Re-triangulation: Connect the new vertex to each edge of the cavity to create new triangles.

(a) Creation of a cavity (the shaded area). (b) Re-triangulation of the cavity.

Figure 2.4: **Insertion of a new vertex V_6 into a Delaunay triangulation.**

Figure 2.4 illustrates the insertion of vertex V_6. Before the insertion, we have a valid Delaunay triangulation of $\{V_1, V_2, V_3, V_4, V_5\}$. The cavity creation step identifies two triangles, $\triangle V_1V_2V_5$ and $\triangle V_2V_3V_5$, whose circumcircles enclose V_6. The shared edge V_2V_5 is canceled out. The remaining edges V_1V_2, V_2V_3, V_3V_5 and V_5V_1 form the edges of the cavity. The re-triangulation step then simply connects V_6 to the four edges and create four new triangles. The other triangle $\triangle V_3V_4V_5$ is not involved with the cavity and thus remains intact.

The incremental insertion algorithm is usually implemented using a pointer-based topological structure to keep track of the evolving Delaunay triangulation. The cavity creation step first locates a triangle that encloses the new vertex and then conducts a depth-first or breath-first search to expand the cavity. The re-triangulation step deletes the *encroached triangles*, that is, those triangles whose circumcircles enclose the new vertex, and inserts the new ones. In many circumstances, the dominant cost is the time required for *point location*: finding the triangle in which a vertex lies, so that a cavity can be expanded from a base.

In practice, the incremental insertion algorithm has been implemented in a number of high-quality free software packages including *Triangle* [92], *Pyramid* [81], *Qhull* [67], and *Tetgen* [90]. Although highly efficient in triangulating relatively small data sets, these implementations are subject to two major limitations. [1]

First, running the programs requires massive amount of memory. Roughly speaking, a 2D triangulation requires about 100 bytes per vertex on average (including the memory usage by the associated triangles), and a 3D triangulation 400 bytes on average (including the memory usage by the associated tetrahedrons). On a commodity server with 8 GB memory, this translates to about 80 million 2D vertices or 20 million 3D vertices. Such capacities fall short of the requirements of large-scale real-world applications where billions of vertices need to be triangulated. Consequently, parallel computers must be used to aggregate hundreds of gigabyte physical memories. However, scalable parallel algorithms and implementations for large-scale Delaunay triangulation based mesh generation are *significantly* more difficult than their sequential counterparts.[2]

Second, after a triangulation is computed and stored to disk, there is no way to *efficiently* query the triangulation or insert more vertices. Flat file format, as described in Chapter 1, is the standard representation for Delaunay triangulations. The massive topological structure used to build the triangulation collapses into a nondescript sequence of records. Even simple queries such as point-in-a-triangle would require chasing out-of-core pointers, as would more complicated ones such as range queries or iso-contour queries. Adding more vertices is equally non-trivial. The entire triangulation must be loaded from flat files back to memory to restore the structure in order to continue the incremental insertion operations—even when only a handful vertices are to be added.

Intuitively, database systems appear to be a natural solution to overcome these limitations. In particular, database systems are built to store and index data that are orders of magnitude larger than the memory size. Interactive queries and incremental updates are their strong suits by design. The question is, how do we map Delaunay incremental insertions to operations on databases?

Directly porting an incore implementation to a database system would be quite infeasible. Following a pointer in memory takes a few nanoseconds (when the target object is in the cache) to tens of nanoseconds (when the target object is in the DRAM), while following a pointer on disk takes time on the order of 10 milliseconds (if the target object is on an un-cached disk page). Furthermore, regardless of the efficiency of an implementation, the inherent application logic of Delaunay triangulation is fundamentally in conflict with the design philosophy of databases. In contrast to record-at-a-time operations performed by the incremental insertion algorithm, database systems are optimized for set-at-a-time operations.

[1] Note that these software packages are capable of other sophisticated geometry computation besides Delaunay triangulation.

[2] In a report [65] identifying the prospects of scalability of a variety of parallel algorithms to petascale architecture, mesh generation and associated load balancing are categorized as Class 2—"scalable provided significant research challenges are overcome."

2.3 The Database Approach Revisited

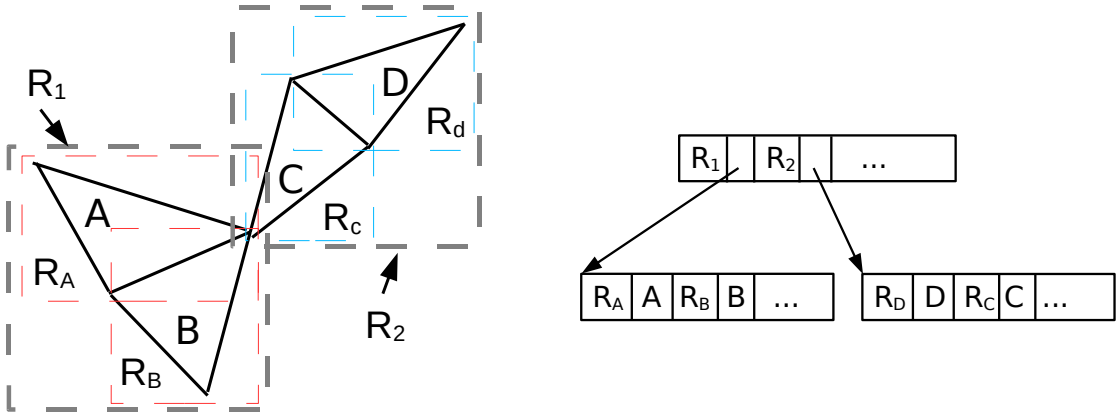
The case of Delaunay triangulation exemplifies the difficulty of mapping scientific applications to database systems. A widely accepted view attributes such a difficulty to a problem called *impedance mismatch* [7, 32, 41, 105]. Namely, the programming model of scientific applications (i.e., data types and procedural interfaces) does not match the existing database capabilities (i.e., tables, SQL, and non-procedural set-oriented accesses). The recommended solution is to incorporate user-defined data types (UDT) and user-defined functions (UDF) into the database to ameliorate the mapping problem.

Following this advice, I have developed an incremental insertion program on top of an R-tree index. The implementation eliminates the impedance mismatch completely by interacting directly with a low-level R-tree API, thus avoiding the overhead of SQL and other database mechanism. In a way, this is the best performance one can expect from an existing database system. [3]

2.3.1 A Trial with R-trees

Before discussing the technique of implementing Delaunay triangulation on an R-tree, we first briefly explain what an R-tree is and how it works.

An R-tree [36] is a spatial access method invented by Guttman in 1984. Since then, a number of variants [10, 46, 75] have been proposed to improve the performance. R-trees are generally regarded as the most flexible and powerful structure for indexing spatial data and have been incorporated in both open-source and commercial database systems such as PostgreSQL and Oracle.



(a) Spatial distribution of four triangles. (b) An R-tree index for the triangles.

Figure 2.5: **Indexing triangles using an R-tree.** Minimum bounding rectangles (MBRs) are defined for the triangles and indexed in an R-tree.

[3] An earlier implementation on a open-source object-relational database system, PostgreSQL, runs orders of magnitude slower due to additional overhead.

An R-tree is an index that organizes multi-dimensional objects in a tree structure similar to that of an B-tree. There are two types of tree nodes in an R-tree: the leaf nodes and the index nodes. A leaf node contains spatial objects to be indexed. Each entry in a leaf node is a record of `<R, data>`, where `R` is the N-dimensional *minimum bounding rectangle* (MBR) of an object and `data` describes the geometry span and other attributes of the object. An index node, on the other hand, contains routing information for guiding data object searches. Each entry in an index node is a record of `<R, ptr>`, where `R` is the MBR that encloses all the MBRs contained in a child node and `ptr` points to where the child node is stored on disk. Both the leaf and the index nodes are mapped to disk pages. Figure 2.5(a) illustrates the MBRs of four triangles A, B, C, and D in an R-tree. Assuming each leaf node and index node can accommodate two records, Figure 2.5(b) shows the corresponding R-tree structure.

An R-tree support three basic operations: search, insertion, and deletion.

- `R-tree-search`: The search operation takes a query MBR (which can degenerate to a point) as input and returns a collection of objects whose MBRs overlap with it. A search starts from the root node. If the current node is an index node, the MBR of every entry is examine to check if it overlaps with the query MBR. If so, recursively search the child node pointed at by the associated `ptr`. If the current node is a leaf node, return the objects whose MBRs overlap with the query MBR. Note that multiple descending paths may be traversed to return all the qualified objects.
- `R-tree-insert`: The insertion operation inserts a new object into an R-tree. The operation first descends an R-tree from the root node. If the current node is an index node, find the entry whose MBR needs the least enlargement to include the MBR of the new object. Ties are resolved by choosing the entry whose MBR has smaller area. Descend to the child node pointed to by the `ptr` associated with the chosen MBR. Repeat the process until a leaf node is encountered. Note that only one path from the root to a leaf node is traversed. If the leaf node has extra space for the new object, insert the new object and adjust the MBRs of the ancestors of the leaf nodes recursively. If the leaf node is full, a split operation is carried out to distribute the existing entries and the new object into two leaf nodes. The newly created leaf node is inserted into the parent (index) node. If the parent node is full, recursively split the parent node until reaching the root node. When a root node is split, the height of the R-tree increases by 1.
- `R-tree-delete`: The delete operation first searches for the object whose MBR matches the MBR to be deleted. The routine is almost identical to that of `R-tree-search` except that at a leaf node only the MBR that exactly matches the target MBR is chosen. If there is no match, return immediately. Otherwise, remove the entry from the leaf node and recursively adjust the MBRs of the ancestor nodes. If a node becomes under-utilized, for example, the number of entries on the node drops below 50%, the node is eliminated from the R-tree. The remaining entries in this node are then re-inserted into the R-tree.

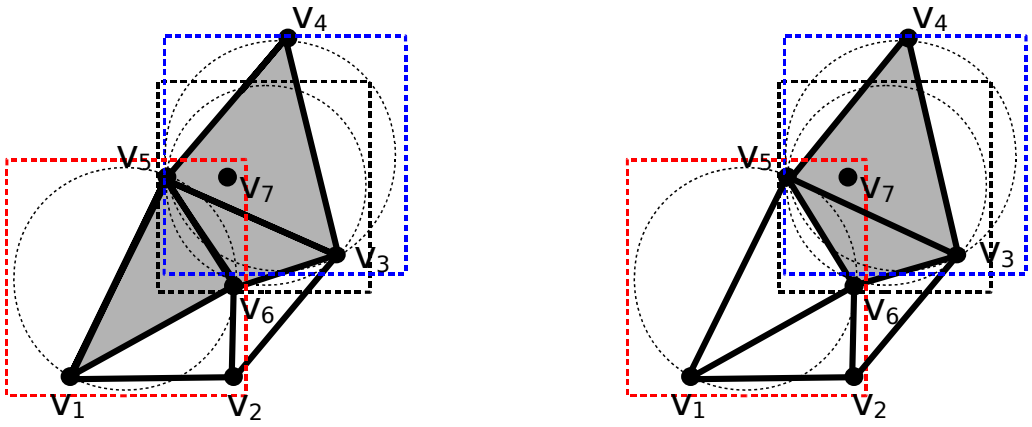
Note that the insertion and deletion operation dynamically adjust the structure of the tree in such a way that all leaf nodes are at the same level, which means an R-tree is a balanced search tree structure, similar to a B-tree.

The main innovation of the R-tree is that it allows the MBRs of the objects and the index nodes to overlap. For example, R_1 and R_2 in Figure 2.5 overlap with each other. As a result, the

maintenance of the tree structure is greatly simplified. But the downside of this feature is that the search operation may need to explore more than one subtree since their MBRs may all overlap with a query MBR. In the worst case, the entire tree must be searched. Therefore, unlike a B-tree where the search time is bound by $O\log(n)$, the worse case search time of an R-tree is $O(n)$, where n is the number of data objects being indexed.

Now, let us study how to construct Delaunay triangulation on an R-tree using the incremental insertion algorithm. A seemingly simple way is to directly index the triangles in an R-tree, and make use of the R-tree's search and update capabilities to carry out the cavity creation and re-triangulation steps. However, although we can use an R-tree to locate the triangle that encloses a new insertion vertex, there is no means for us to expand from the base triangle to create the cavity since there is no connectivity information available.

A better way to make use of an R-tree is to index the circumcircles of the triangles. After all, when we create a cavity, we are only interested in retrieving those triangles whose circumcircles encloses a new insertion vertex. For simplicity, we refer to the MBRs of the circumcircles of triangles as *circumrectangles*. The algorithm works by first querying an R-tree to retrieve a collection of candidate triangles whose circumrectangles enclose a new insertion vertex, and then eliminating the false hits using a robust in-circle geometry test [78, 80].



(a) Searching for candidate triangles. (b) Dismissal of false hit.

Figure 2.6: **Insertion of a vertex V_7 into a Delaunay triangulation indexed by an R-tree.** The R-tree index returns three candidate triangles. One of them, $\triangle V_1V_6V_5$, is a false hit. Although its circumrectangle encloses the new insertion vertex, its circumcircle does not.

Figure 2.6 shows an example. Assume V_7 is a new vertex to be inserted into a Delaunay triangulation. We first search the R-tree and find that $\triangle V_1V_6V_5$, $\triangle V_6V_3V_5$, and $\triangle V_3V_4V_5$ are the triangles whose circumrectangles enclose V_7. Robust geometry predicate tells us that the circumcircle of $\triangle V_1V_6V_5$ does not really enclose V_7. So we ignore it as a false hit. The remaining two triangles, $\triangle V_6V_3V_5$ and $\triangle V_3V_4V_5$, pass the test and are deleted from the R-tree to create the cavity. The re-triangulation step produces four new triangles, $\triangle V_7V_4V_5$, $\triangle V_7V_5V_6$, $\triangle V_7V_6V_3$, and $\triangle V_7V_3V_4$, which are then inserted in the R-tree. The general procedure of inserting a new vertex is shown in Figure 2.7.[4]

[4]The new vertex is assumed to fall inside the convex hull of the existing Delaunay triangulation.

1.	`Search the R-tree`:	Find triangles whose circumrectangles enclose the new vertex
2.	`Incircle tests`:	Apply a robust geometry predicate to dismiss the false hits
3.	`Delete from the R-tree`:	Create a cavity that comprises the true hits
4.	`Insert into the R-tree`:	Insert the new triangles into the R-tree

Figure 2.7: **Inserting a new vertex into a Delaunay triangulation indexed by an R-tree.**

A technical detail that has been conveniently side-stepped till now is how to define the circumrectangles for triangles. Mathematically, we can compute the center (O_x, O_y) and the radius $R_{circumcircle}$ of the circumcircle of a triangle $\triangle abc$ using the formulae:

$$O_x = c_x + \frac{\begin{vmatrix} (a_x - c_x)^2 + (a_y - c_y)^2 & (a_y - c_y) \\ (b_x - c_x)^2 + (b_y - c_y)^2 & (b_y - c_y) \end{vmatrix}}{2 \times \begin{vmatrix} (a_x - c_x) & (a_y - c_y) \\ (b_x - c_x) & (b_y - c_y) \end{vmatrix}} \tag{2.1}$$

$$O_y = c_y + \frac{\begin{vmatrix} (a_x - c_x) & (a_x - c_x)^2 + (a_y - c_y)^2 \\ (b_x - c_x) & (b_x - c_x)^2 + (b_y - c_y)^2 \end{vmatrix}}{2 \times \begin{vmatrix} (a_x - c_x) & (a_y - c_y) \\ (b_x - c_x) & (b_y - c_y) \end{vmatrix}} \tag{2.2}$$

$$R_{circumcircle} = \frac{L_a L_b L_c}{2 \times \begin{vmatrix} (a_x - c_x) & (a_y - c_y) \\ (b_x - c_x) & (b_y - c_y) \end{vmatrix}} \tag{2.3}$$

where L_a, L_b, and L_c are the edge lengths of $\triangle abc$.

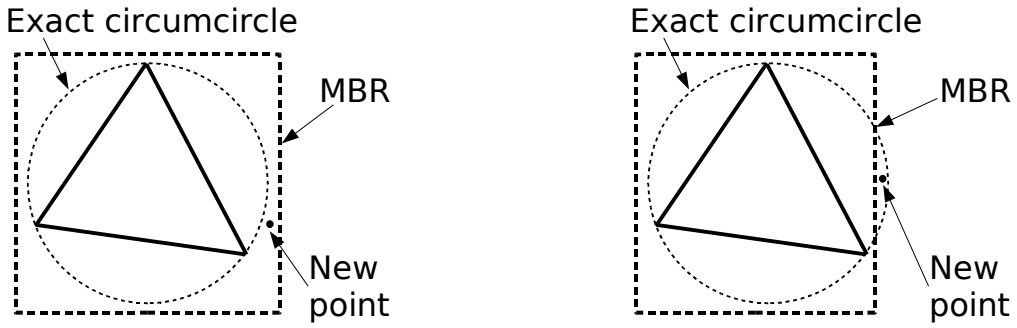
We then compute the lower-left corner and upper-right corner of the circumrectangle of a triangle as follows:

$$(O_x - R_{circumcircle}, O_y - R_{circumcircle})$$

$$(O_x + R_{circumcircle}, O_y + R_{circumcircle})$$

However, due to floating-point arithmetic round-off errors, we cannot obtain the exact values by evaluating these formulae. The results are merely approximations, which may cause circumrectangles to shift, expand or shrink slightly. When we search for the enclosing circumrectangles, we can encounter two types of errors: a *false hit* or a *false miss*, as shown in Figure 2.8. A false hit is acceptable since we can dismiss it along with other false hits using a robust geometry predicate. However, a false miss is detrimental. If the R-tree does not return a triangle whose circumcircle truly encloses the new vertex, we would not be able to find the triangle in other ways, and the resulting triangulation would be incorrect (for example, cross-over triangles would appear).

To fix this problem, we use the interval arithmetic technique [14] to slightly expand the circumrectangles to ensure that they *always* fully contain the circumcircles. Although there will be (a few) more false hits, the correctness of the algorithm is guaranteed.



(a) False hit (acceptable). (b) False miss (detrimental).

Figure 2.8: **Round-off errors may result in either a false hit or a false miss.** For illustration purpose, the errors (shift, expansion, shrinking) associated with the circumrectangles are magnified to show the effects.

2.3.2 The Missing Performance

We have built an R-tree implementation in C that includes a standard page buffer pool manager [33] and that supports the `R-tree-search`, `R-tree-insert`, and `R-tree-delete` function calls. Seeing that Delaunay triangulation always inserts more triangles into an R-tree, it is unnecessary, during a delete operation, to eliminate an under-utilized R-tree node and re-insert its remaining entries. The newly created triangles will soon refill the node. This observation has been incorporated in the implementation to improve the performance of the delete operations.

On top of the R-tree, we have developed an incremental insertion Delaunay triangulation program using the techniques described in the previous section. There is no SQL query processing, logging, locking, or any other overhead associated with a full-fledged database system. There is no mismatch of the programming models or type systems. (The triangulation program was also written in C.) In short, there is no impedance mismatch.

Unfortunately, the performance of this implementation, which we call *Goose*, turns out to be abysmal. Compared to the state-of-the-art incore Delaunay triangulator *Triangle*, Goose runs more than 3 orders of magnitude slower. Computing the Delaunay triangulation of 1 million random points takes Triangle 1 minute, but takes Goose 25 hours 43 minutes and 35 seconds.

The experiments were conducted on a server with two Intel 3.6 GHz Xeon processors running Linux 2.6.17. (Only one processor was used in the experiments.) The memory subsystem consisted of 8 GB physical memory and 18 GB swap space. Both Triangle and Goose were compiled with `gcc` using the `-O2` optimization flag. In the experiments, the data sets fit completely in the physical memory (8 GB). The Goose implementation never accessed disk to swap in or out R-tree node pages, except for storing the final results.

Given that both Triangle and Goose are operating on in-memory structures, why is there such a huge difference in performance? What is Goose doing that makes it run so much slower? Where does the time go?

To answer these questions, we conducted an experiment for triangulating 100,000 random points and used `GNU gprof` to obtain a running time profile of Goose. The percentage contri-

bution of different operations to the running time of inserting a new vertex is listed in Figure 2.9.

`Search the R-tree:`	16%
`Incircle tests:`	0.1%
`Delete from the R-tree:`	74%
`Insert into the R-tree:`	4%
`Other operations (including I/O stalls):`	5%

Figure 2.9: **The percentage contribution of inserting a new vertex using Goose**

The biggest surprise is the cost of the delete operations, which account for 74% of the total running time. The cost of searching the R-tree to retrieve a collection of candidate triangles uses 16% of the time. Following that is the cost of inserting new triangles, which accounts for 4% of the time. The seemingly expensive robust in-circle geometry test is responsible for only 0.1% of the total time. Simply put, the vast majority of the running time is spent searching and updating the R-tree structure.

A closer look at the running time of the *R-tree-delete* operation further reveals that more than 98% of the cost is attributed to searching the R-tree to find the true-hit triangles that need to be removed. The removal operation itself accounts for less than 2% of the time (thanks to the optimization mentioned earlier). Note that the search operation involved in *R-tree-delete* is different in nature to the search operation carried out when we create a cavity. The former searches for an exact match for a circumrectangle in order to delete it; the latter searches for all circumrectangles that enclose a new vertex in order to create a cavity.

If we aggregate the cost of searching the R-tree, regardless of the nature of the searches, the overall contribution of searches is 89% (i.e., 16% + 74%$\times$98%) of the total running time. The total contribution of updates (i.e., insertion and deletion) is about 5.5% (i.e., 4% + 74%$\times$2%).

Besides the running time, two other metrics are also measured. First, the R-tree search efficiency is only 10%. That is, out of 10 MBRs checked, only 1 results in a hit. This of course makes sense. Since a triangulation is a dense spatial structure, the circumrectangles are naturally overlapping with one another. As a result, the MBRs in the R-tree index nodes also tend to overlap, leading to multiple search paths down the tree structure, which, in turn, increases the chance of wasted searches. Second, as many as 67% of the triangles inserted are later deleted. On average, the insertion of a new vertex deletes 4 existing triangles and inserts 6 new triangles. Hence the incremental insertion algorithm removes about two thirds (i.e., 67%) of the total triangles that are ever generated. These two metrics quantitatively reflect the unstructured property of Delaunay triangulations and the dynamic nature of the incremental insertion algorithm.

From the timing and the statistics, we make three observations. First, frequent searches of the R-tree hurt the performance most. We cannot afford to conduct associative key searches on the R-tree to create cavities. Second, the cost of updating an R-tree (i.e., insertion and deletion of triangles), though much smaller than the search cost, is still non-trivial. Even if we could completely eliminate the search cost, the remaining 5.5% running time (for insertion and deletion) would remain one to two orders of magnitude larger than an efficient incore implementation. Third, most of of triangles should not have been stored in the R-tree in the first place since they are to be deleted later. The unnecessary insertions cause more deletions, clutter the MBRs of the

index nodes, and slow down the search operations. If compelled to draw a conclusion, we must admit that mapping Delaunay triangulation and the incremental insertion algorithm to operating on R-trees appears to be a dead end.

The Delaunay triangulation problem presented here is just one instance. Other scientific data sets may take the form of an array, a tree, or a graph. Mapping such data sets to traditional database structures imposes equally difficult challenges. For example, recent research [39] has shown that database systems such as MySQL and BerkeleyDB are not suitable for efficient storage and on-demand query processing of massive graph data sets.

2.4 Structural Mismatch

So, what are the causes of such difficulties? Impedance mismatch—the discrepancy in the programming models between scientific applications and database systems—is certainly one culprit. The proposition of integrating user-defined data types (UDT) and user-defined functions (UDF) into databases helps resolve the issue of how to efficiently manipulate individual data objects. However, when a subset is to be manipulated by a user-defined function, for instance, incremental insertion of a new vertex into a Delaunay triangulation, a standard database query still has be to executed in order to retrieve the data objects of interest. Unfortunately, as shown in the case of Delaunay triangulation, frequent database queries (associative searches, insertions, and deletions) slow down a scientific application to a crawl.

The root cause of the problem, I believe, lies in *structural mismatch*, that is, *the discrepancy between how data are stored and indexed in databases and how data are accessed and manipulated by scientific applications.*

A figurative way to illustrate structural mismatch is to imagine squeezing an octopus into a cubic box (of equal volume). We can certainly do that with due efforts. But the octopus can hardly be moved afterwards. If we want to turn it sideways or take a biopsy sample from one of its tentacles, a large number of maneuvers have to be carried out. In the interim, cares must be taken to prevent the octopus from slipping out of the box. Now map an unstructured data set to an octopus and a traditional database to a box. We can surely manage to load a data set into a database by defining a relational schema. But if we want to expand, modify, or query part of the data set, expensive database operations need to be performed. All the while we have to prevent the data set from "slipping" out of the database and using up the main memory.

Going back to the Delaunay triangulation example, we have chosen to map a triangulation to an R-tree, thus "flattening" the data set to fit the database. While doing so, we lose important topological information about the data set and hence pay a significant performance penalty for manipulating the triangulation. On the other hand, we could have chosen to preserve the topological information by introducing out-of-core pointers in a relational schema, thus "stretching" the database to accommodate the data set. But then, we would have to rely on record-at-a-time operations (which we have decided against). Consequently, the performance would degrade to no better than random accesses to flat files.

More generally, scientific data sets often exhibit strong spatial and/or temporal correlation among the compositing parts/regions. The natural representation of a scientific data set can be an array, a tree, a graph, or some other non-trivial data structure. When storing such a data set in a

database, we have to serialize (flatten) the data structure into a sequence of records and map the records to disk pages, which are organized by databases in a format referred to as *slotted pages*.

Figure 2.10 shows an example. In this particular format used within the PostgreSQL database system [66], a page (e.g., 4 KB) is partitioned into 5 regions. The `Page Header` field contains general information about the page, including relative offsets recording the start and the end of the free space. An array of `Item Pointers` point to the actual items (data records or index entries). `Free Space` represents the un-allocated space. New item pointers are allocated from the start of the this area and new items from the end. The `Items (records)` (i.e., data objects) are stored in space allocated backwards from (almost) the end of a page. The `Special Space` at the end of a page stores data specific to an access method if the page is an index page. The field is unused in a regular table data page. Other database systems use some variants of the page layout.

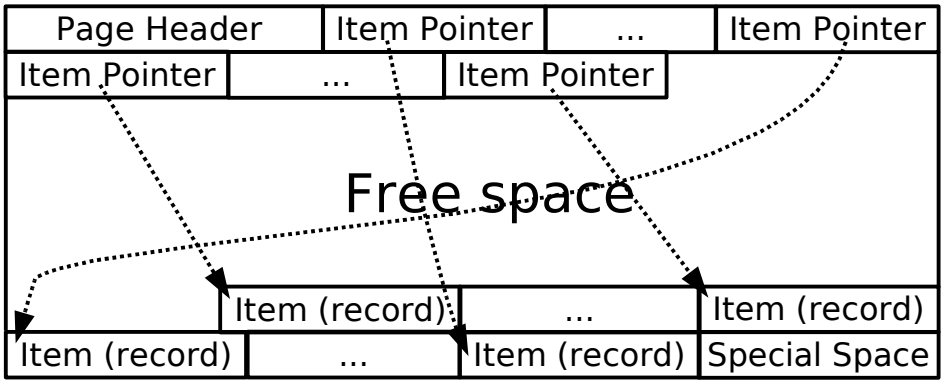


Figure 2.10: **Data layout on a slotted database page.** This particular example shows the page format used within PostgreSQL tables and indices.

When we read the data back from disk, we should ideally reverse the serialization and reconstruct the correlation among the data. But the problem is that scientific data sets are huge. Loading a massive data set into the memory of a supercomputer can be a problem, let alone on commodity servers. The simple way out—and the only way out in a traditional database system—is to cache data in main memory in slotted page unit. Without a means to partially reconstruct the natural data structure, we are left with a collection of "structureless" tabular chunks stored in slotted pages.

In order to operate on the tabular format, we have to adapt algorithms originally designed for the natural representations, as we did for the incremental insertion algorithm. The biggest problem is how to efficiently access individual data objects. Sequential scan incurs an $O(n)$ cost for every single data access. Unless an algorithm needs to traverse an entire data set in an arbitrary order, sequential scan is useless for implementing any non-trivial scientific algorithms. The only other choice we have is to use database index structures. Still, the cost of executing a search operation, assuming it is $O(\log n)$, could be tens of thousands to millions of instructions (and memory references). In contrast, accessing a data object following a pointer, as does a typical incore implementation operating on a natural representation, costs only two instructions and memory references: one for loading the pointer (address) and the other for loading the data

object pointed to by the pointer. The huge difference in data access time implies that even if we are able to develop a scientific algorithm using the best tools available in current databases, the implementation is destined to be several orders of magnitude slower than its incore counterpart, as is the case with Goose and Triangle.

In summary, a large class of unstructured scientific data sets simply does not fit with the built-in tabular abstraction of traditional databases. Forcing scientific data sets and applications into databases could only result in poor performance and cause user frustration.

Chapter 3

A Scalable Database Approach

Structural mismatch is a fact of life. We cannot expect scientists to change their data sets to fit databases. Even if they want to, they are probably not able to, since most data sets are generated or collected according to a natural process or a physical problem. Meanwhile, we can hardly afford to make any major changes to the database systems cores, which are built on decades of intense research and software development. Therefore, the only option we have is to come up with a work-around to bridge the structural mismatch.

This chapter presents a new scalable database approach that is based on a simple idea: (1) deliver data to scientific applications in such a way that the data can be efficiently manipulated, and (2) deliver data to databases in such a way that the data can be efficiently stored and indexed.

Our proposal adds a *computational cache* on top of a standard database buffer pool manager and provides a mechanism to translate data between the inherent unstructured representation (stored in the computational cache) and the native database format (stored in slotted data pages). This approach diverges from the translation-free buffer management scheme used by today's DBMSs [44]. The conventional wisdom has been that "marshaling" and "un-marshaling" data from and to disk is computationally too expensive to be practical. We believe, however, that it is a reasonable price to pay to mitigate the effects of structural mismatch and to speed up scientific applications operating on databases.

Although the first of its kind, the proposed database approach contains a number of ideas that can be traced back to as early as the 1970s. This chapter concludes with a discussion of the related work and points out the similarity and difference between the proposed solution and the earlier works.

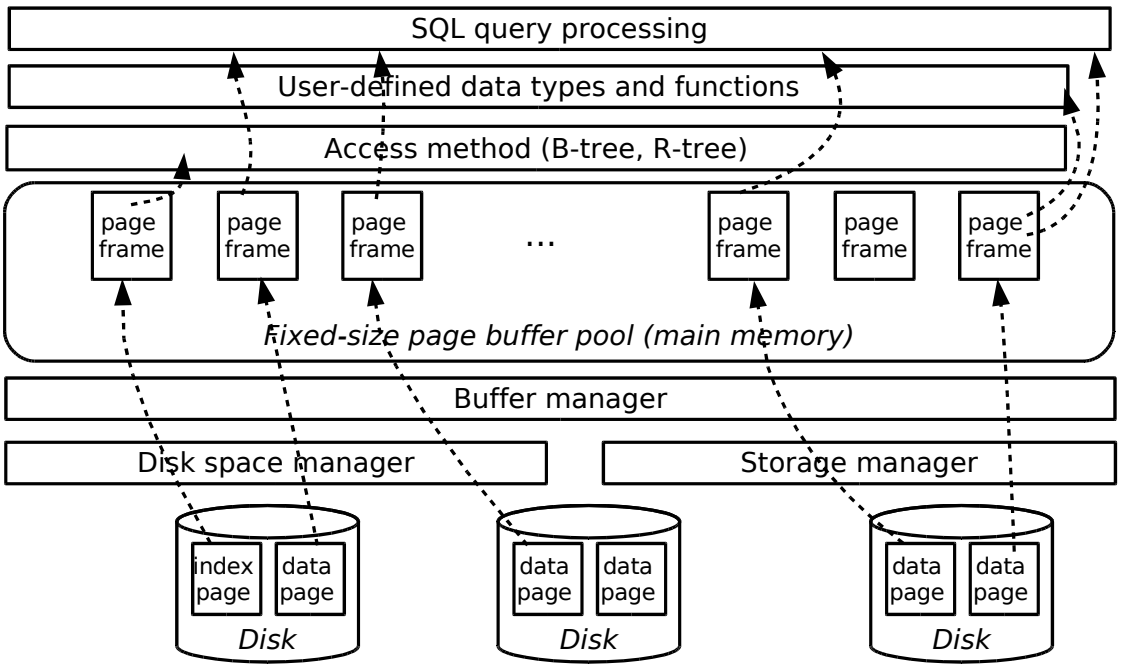


Figure 3.1: **A traditional database system.**

3.1 Framework Overview

Today's databases, refined and optimized over the past four decades, are highly sophisticated software systems. They are particularly good at managing massive tabular data commonly used in the commercial and governmental sectors. Our goal is to make them also good at managing massive unstructured data sets commonly used in scientific and engineering applications. Structural mismatch is the key obstacle we have to overcome.

Our solution strategy is to organically integrate a new mechanism into standard databases, rather than building systems from scratch or attaching a kludge to existing systems. The motivation is two-fold. On the one hand, fine-tuned and robust software components of a database system, such as the buffer pool manager and access methods, are (almost) always needed regardless of the nature of a solution. We should capitalize on this existing capability instead of reinventing the wheel. On the other hand, structural mismatch is a problem *within*. Any external solution would be a veneer fix at the best; the inherent conflict would still persist.

This section briefly illustrates the working of traditional databases and explains how to incorporate a new functionality into the established machinery to solve the structural mismatch problem.

3.1.1 Traditional Database Management Systems

Figure 3.1 shows an abstract overview of today's database systems, including IBM DB2 Universal Database, Microsoft SQL Server, Oracle Database, and PostgreSQL. An effort has been

made to highlight the data paths between disk storage and high-level SQL query processing. Other important database core modules such as lock management, logging systems, memory management, and replication services are omitted for clarity of illustration.

At the top level, SQL query processing [74, 77] encapsulates all the complex operations of parsing an SQL statement, building a query tree, rewriting and optimizing a query plan, and executing a query. A database derives its non-procedural, set-at-a-time power from this layer.

User-defined data types and functions [87] enable extensions to a database. Using a database internal structure called *system catalog*, users can add new data types, functions, and operators to a database. At runtime, the database engine consults the system catalog to retrieve the relevant information and executes the requested functions on the target data types. This is usually the layer where the impedance mismatch problem is addressed.

To efficiently retrieve data and deliver them to the upper layers, database systems make extensive use of highly optimized access methods such as B-trees [21, 69], R-trees [10, 36, 75], and hashing [53]. Note that there may be cases (illustrated in the figure as bypasses) when a query optimizer (contained in the SQL query processing layer) chooses not to use an index or any other access methods if it determines that doing so can improve the performance of a particular query. But in general, index structures are heavily relied on to execute queries. It cannot be emphasized too much how important they are to the success of database systems.

The next level in the software hierarchy is a database buffer manager [20, 59], which oversees a main memory buffer pool consisting of a large array of fixed-size page frames. The size of the frames (e.g., 8 KB) equals the size of database disk pages. All database pages, either data pages storing records of a table or index pages containing search routing information, are mapped to page frames in the buffer pool. Pages are copied in native format from disk directly, manipulated in memory in native format, and written out to disk if necessary. This translation-free buffer management scheme is the norm in today's databases.

At the bottom level, the disk space manager allocates new database pages, deallocates unused pages, and reorganizes disk page layout when requested to improve I/O performance. The storage manager caches disk pages, reorders I/O request sequences, and issues disk read/write commands. This layer gains performance by understanding the characteristics of the storage system rather than the semantics of user-defined functions or SQL queries.

Zooming in on the data paths, we can see that data travel straight through the different software layers without changing representation. Data layout in memory is identical to that on disk. [1] This makes sense. Typical database workloads such as online transaction processing (OLTP) and decision-support systems (DSS) operate on tabular data sets. The only reasonable way to organize the data in memory is to *keep* them as a list of discrete records, which is exactly the data layout on disk.

Unfortunately, the simplicity of direct data transfer gives way to the inefficiency of structural mismatch when it comes to dealing with unstructured scientific data sets and applications.

[1] An exception is presented in Section 3.4.

3.1.2 A New Framework

To bridge the structural mismatch, we propose to (1) add a *computational cache* on top of the standard page buffer pool, and (2) provide a translation mechanism to convert data between the representation optimized for applications and the representation suitable for storage and indexing. Most of the existing machinery of a database system does not need to be modified. The only exception is that user-defined functions should now be able to operate on a computational cache.

Figure 3.2 shows the new solution framework called a *computational database system*, which encodes a triple meaning. First, the overall solution is aimed at managing and manipulating large-scale unstructured scientific data sets used in *computational sciences*. Second, the new cache structure is intended to optimize the performance of *computational algorithms* operating on databases. Third, the translation mechanism introduced is a *computational task* by itself.

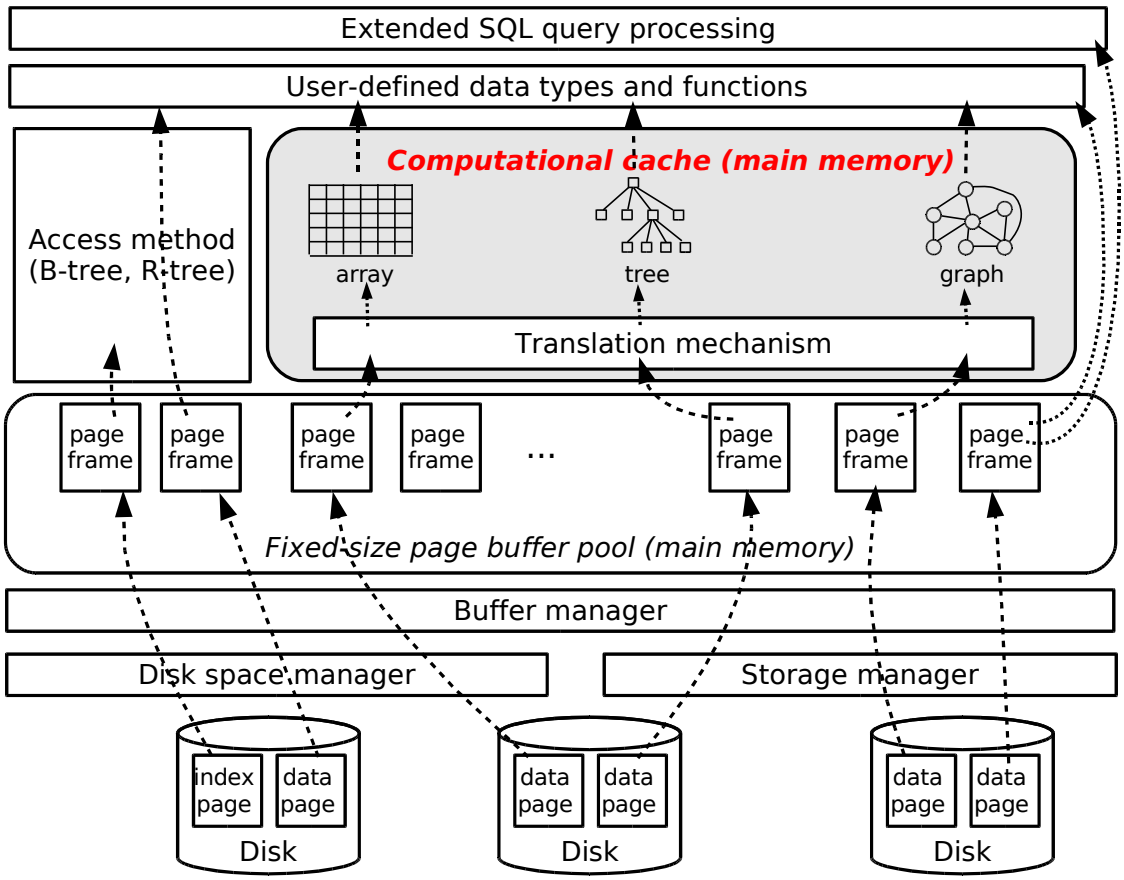


Figure 3.2: **A computational database system.**

Compared to a traditional database management system, a computational database system provides additional venues of accessing data. Besides the existing data paths, a data flow can take a detour through the memory hierarchy and convert into a natural form that is suitable for scientific applications. The placement of the new software module (in the gray box) is dictated by two factors: *functionality* and *performance*. Since the new module's function is to provide

alternative data paths parallel to the ones going through the index structures, it is a natural choice to place it alongside the access method module. A second and more important factor is performance. We need to access data objects in groups to amortize CPU and I/O overheads. The page buffer pool manager is just the right layer to provide the service. By situating a computational cache on top of the buffer manager (the fixed-size page buffer pool), we can retrieve a pageful of existing data objects, allocate an empty page to store a group of new data objects, or delete a page to invalidate a batch of old data objects.

One huge potential application of a computational database system is to conduct large-scale scientific simulations directly on databases. As shown in Figure 1.1 (Chapter 1), a scientific simulation consists of a number of components (algorithms) that operate on massive data sets. Since structural mismatch has prevented implementing scientific algorithms on top of databases, it is a common practice to implement these simulation components on memory-resident data structures. Consequently, the size of the problem that can be modeled, meshed, solved, and analyzed is bound by the main memory size. By bridging structural mismatch, a computational database system allows us to recast a simulation pipeline as a sequence of database queries. In particular, we can implement the simulation components as user-defined functions and rely on a computational cache to provide efficient data accesses to the underlying data sets stored in a database. Memory size should no longer be a limiting factor. As long as there is enough processing power (multicore processors) and sufficient storage (RAID disks), we can conduct a scientific simulation. As a result, we can (1) defer the time when a simulation has to be executed on a supercomputer, (2) monitor and steer an ongoing simulation by issuing database queries, (3) have a coherent data store recording all aspects of a simulation, and (4) explore models, meshes, and solution fields interactively by querying databases. All these are new capabilities that are previously unconceivable and sought after by scientists and engineers.

Exciting as it is, the proposed solution begs two questions. First, is it feasible? Can we really bridge the structural mismatch between scientific data sets and traditional databases by adding a computational cache? The rest of this dissertation is devoted to answering this question. It will be shown that not only is the technical approach feasible, but it is also capable of delivering the necessary performance and scalability.

The second question is, how general is the solution? Can we build one computational database system and use it for all scientific data sets? What we propose is a generic solution framework, which clearly identifies where the structural mismatch problem should be attacked. Nevertheless, there are numerous scientific data sets with widely different spatial and temporal properties, it is impossible to hardwire *one* cure-all solution. That said, it should be also be noted that the *types* of different scientific data sets in terms of the fundamental data structures are quite limited. Vendors or the open-source community could develop one computational database system for each type of data sets. Or, we can develop an extensible framework that allows users to define their own computational cache and translation mechanism. Similar proposal for database extension has been adopted in the past. Namely, the user-defined functions (UDF) and the user-defined data types (UDT). All of today's major database systems support UDF and UDT. The main difference, however, is that a user-defined computational cache (UDC) needs to interact more intimately with the inner working of a database system. The challenge is how can we extend an existing database system to export the right level of abstraction to support UDC extension. There is already an indication of promise as shown in Figure 3.2: The computational

cache module is well isolated from the rest of the system. The clear logical separation should greatly simplify the design and implementation of an extensible computational database system. However, how to develop an extensible framework is beyond the scope of this dissertation and is left as future work.

3.2 Computational Cache

The proposition of converting data between different representations is different from the translation-free buffer management scheme adopted by today's DBMSs. This section discusses the rationale and the design issues of a computational cache.

3.2.1 Rationale

A standard caching scheme delivers data verbatim. Figure 3.3 shows how caching works in a modern computer system [16]. Data (and code) are fetched from disk in fixed-size pages (e.g., 4 KB) and cached in main memory, and then fetched from main memory in fixed-size blocks (e.g., 64 bytes) and cached in processor cache, and finally fetched from processor cache in fixed-size words (e.g., 8 bytes) and delivered to registers where the processor can directly manipulate the data. Note that regardless of the granularity of data transfer between the different levels, data traverse through the memory hierarchy in one format. There is no transformation of any kind.

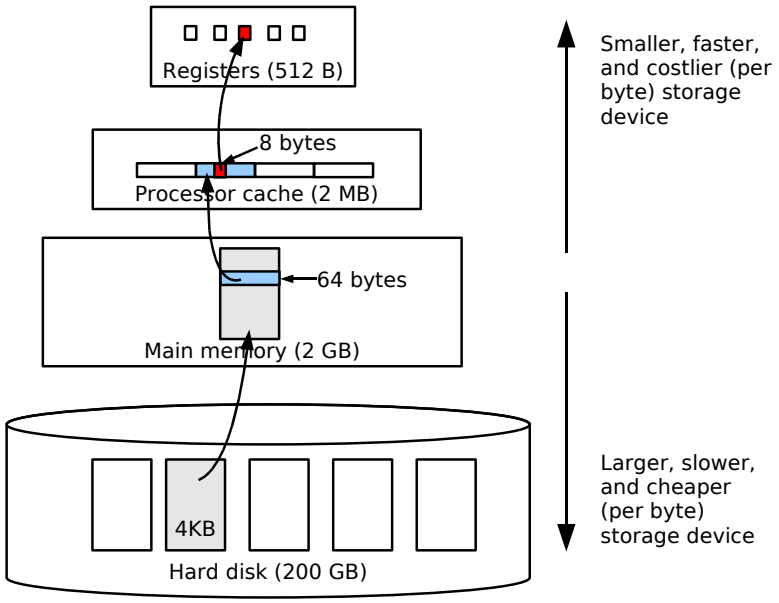


Figure 3.3: **A typical translation-free caching scheme.**

However, there certainly exist cases where converting and caching data in a different format can improve performance. (For the purpose of this dissertation, we focus only on the main memory cache.) For example, suppose we have a data set consisting of a collection of unordered

records and we need to conduct key searches. If there are only a small number of searches, we can cache the data set in main memory (assuming it fits) in its native format as a sequence of pages, and go through the list multiple times (assuming we are given a search key only after we finish the previous search). The running time of this brute-force method is $O(kn)$, where k is the number of searches and n is the number of records. However, if k is large, for example, $k \approx n$, the total cost becomes quadratic $O(n^2)$. In this case, we can afford to build a balanced binary search tree upfront in $O(n \log n)$ time when we load the data into memory, and then search the binary tree to answer the queries. The total cost is $O(n \log n + k \log n)$. When $k \approx n$, the running time is $O(n \log n)$, which outperforms the non-conversion case. Better, we can simply build a hash table in $O(n)$ time when loading the records. Since the average cost of searching a hash table is $O(1)$, the total running time improves to $O(n + k)$, or $O(n)$ when $k \approx n$. Now let us extend our example further and intermix range queries (i.e., finding all the records whose keys are between a specified range) with individual key searches. In this case, the balanced binary tree becomes indispensable since (simple) hashing does not support range queries. Of course, if there is sufficient memory space, we would prefer to keep both data structures to speed up all the queries.

This simple example illustrates an important rationale: Whether a data set needs to be converted to a particular format depends largely on how an application operates on the data. The overhead (memory usage and CPU cycles) of conversion, such as constructing a balanced binary search tree or building a hash table, is justified if the overall performance (running time) of an application is improved. Therefore, there is no right or wrong when it comes to caching data in a native format or in a converted format. The conventional wisdom of avoiding "marshalling" and "un-marshalling" data from and to disk is based on years of experience of the database community working with traditional database applications. Scientific data sets and applications are different. The data sets are more unstructured; the applications are more computation intensive. Hence, "marshalling" and "un-marshalling" scientific data is often worth the effort.

3.2.2 Design issues

A *computational cache* is a main memory structure that stores the converted representation of a tabularized scientific data set. The purpose of a computational cache is to provide a faster cache than the database buffer pool cache. Instead of searching a database index structure to locate data objects, we should access data mostly from a computational cache.

Functionally, a computational cache is similar to a processor cache. Both provide faster access to data and and shield an application (i.e., the processor) from the lower-level slower memory, unless there is a cache miss. But the two derive performance from two different sources. A processor cache, made of SRAM, is inherently faster than a DRAM main memory. Regardless of the application, a cache hit is a *system optimization* that cuts short the data access path. In comparison, a computational cache uses the same DRAM technology as the buffer pool cache. If we could know in advance where in the buffer pool we should access the data, then there would no need for a computational cache. However, as we do not have the knowledge, we have to search a database index structure, which, in most cases, costs $O(\log n)$. The data representation stored in a computational cache, on the other hand, matches more closely to the natural properties of a data set and the application logic. Accessing a data object may cost only one operation and take $O(1)$

time (e.g., following a pointer or accessing an indexed element). Therefore, the performance of a computational cache is due to an *algorithmic optimization*. Of course, since we can now significantly reduce the number of memory references, it is more likely for a computational cache hit to be also a processor cache hit. In other words, an algorithmic optimization improves our chance of exploiting the system optimization.

Operationally, let us examine the following design issues in turn: *data representation*, *application interface*, *I/O management*, and *memory management*.

- *Data representation.* How data is represented in a computational cache is of paramount importance. It determines how fast we can access the data. Usually, a data set has a natural representation, such as an array, a tree, or a graph. Algorithms operating on a data set are often designed to manipulate the natural representation. If that is the case, a computational cache should store the natural representation. However, if some efficient algorithm is designed to operate on a format different from the natural representation of a data set, a careful evaluation is needed to determine how efficient it is to emulate the new format using the natural representation. If the performance difference is unacceptable, then it may be necessary to implement a second representation of the data set. In practice, such cases may be quite rare.
- *Application interface.* In order for applications to take advantage of the high-performance data representation, a computational cache should export a high-level data access interface (API). For example, if an unstructured data set is a tree, an application may want to conduct tree traversal operations. The interface thus exported is not visible to the end user but only to the user-defined function developers and SQL query processing layer. It is the entry point to the alternative data paths within a computational database system.
- *I/O management.* A computational cache may access disk either directly or indirectly. We are in favor of an indirect design and make a computational cache operate on top of a standard database page buffer pool. Data pages are first fetched by a buffer manager and cached in the buffer pool, and then delivered to the computational cache. In the process, data are converted from the tabular storage format to the natural representation via a translation mechanism (to be discussed in the next section). The advantage of this design is that the buffer manager remains the sole I/O request initiator. All database pages are still managed at a single location: the page buffer pool. Effects of data page manipulations by a computational cache, such as read, write, and update, become visible to the rest of the system without delay. No special synchronization mechanism is needed to advertise the effects of the operations. The drawback is that data may be double-buffered, once in the page buffer pool (in the native database format), and once in the computational cache (in a converted format). As will be shown later in the dissertation, how much memory is used is far less important than how the memory is used.
- *Memory management.* The size of a computational cache is bound by a prescribed value. We cannot expand a data structure (e.g., a tree representation) infinitely. When a computational cache becomes full, we need to make room for new data. Part of the data structure has to be either discarded if it has not been modified, or pushed down to the buffer pool for output if it has been modified. The question is which part of the data structure should be replaced. We certainly do not want to discard or swap out data that will be used again

shortly. Thus, we must use a data replacement policy such as the least-recently-used (LRU) algorithm. However, implementing such a policy at the individual data object level is problematic. First, the memory overhead of keeping track of the use patterns of all data objects is too high. Second, data objects chosen to be swapped out may be *logically* unrelated. That is, they may not be correlated either geometrically, topological, or temporally. They are selected to be evicted from a computational cache simply because (1) they have been modified (e.g., newly created), and (2) they qualify for the replacement criterion (e.g., least recently used). Storing them on the same *physical* data page would result in poor locality of reference in the future. To solve this problem, we can organize logically-related data objects into physical groups (i.e., contiguous regions in main memory) that can be directly mapped to data pages, and make a replacement decision at the group level. A concrete example illustrating the technique will be presented in the next chapter.

In short, the function of a computational cache is to provide applications with the appropriate representations of scientific data sets. The reasoning is that the overhead of maintaining and managing a computational cache will be far offset by the the acceleration achieved by having applications operate on the appropriate data representations.

3.3 Translation Mechanism

In order for a computational cache to work as designed, we use a *translation mechanism* to (1) transfer data in and out of the cache in page unit, and (2) convert the data format. An efficient translation mechanism is critical to the performance of the proposed solution.

Conceptually, a translation mechanism can be thought of as a special type of virtual memory system. When a data access cannot be satisfied from the computational cache, a "page fault" takes place and the translation mechanism is triggered. Similar to a virtual memory system, a translation mechanism fetches *one page* of data at a time instead of just fetching the individual data object that has caused the cache miss. Different from a virtual memory system, data items are extracted from the fetched slotted page and installed into the appropriate data structure in a computational cache rather then mapped to memory verbatim.

But we have a problem: There is no page table for us to look up. Which *one page* should we fetch?

To solve this problem, let us zoom out from the data page level and look at the big picture. On one side, we have an unstructured data set whose optimal (natural) representation is partially stored in the computational cache. On the side, we have a collection of slotted pages that store the serialized version of the same data set. What we need is a systematic mapping between the entire optimal representation and the slotted pages. This mapping must somehow be stored in the database and be efficiently looked up.

A natural choice is database index structures. Because of the spatial and temporal locality universally present in scientific data sets, we can (almost) always cluster logical (e.g., spatially or temporally) correlated data objects on the same physical data page. For each page, we can extract the logical property common to all data objects in the page, for example, they all fall within a particular minimum bounding rectangle (MBR), and assign the property (the MBR) as *the key for the page*. We can then use the keys to build an unclustered index, such as a B-tree or

an R-tree, to keep track of the data pages. The index structure is the "page table" we seek. For simplicity, we refer to such an index as a *page index*.

There is nothing special about a page index. The index entries are still of the form `<key, value>`, where `key` represents a key (property) value common to all the data objects store on a page and `value` is a disk page identifier. The only difference from a regular index is that individual data objects are no longer being indexed. The granularity of the index has coarsened to the page level. Within a data page, there is no need to sort the data objects and organize them in order.

With a page index, we can service a computational cache miss as follows: (1) construct the logical key for the missing object, (2) search the page index to find the data page id, and (3) request the buffer pool manager to fetch the page.

At this point, we are half way through the translation process. The next step is to retrieve the data items from a data page (which is already cached in the buffer pool) and convert their representation from the tabular format to the natural format. In order to achieve good performance, we have to resolve two issues.

- *Data placement.* Logically correlated data objects should be physically clustered in main memory as they are on the slotted data pages. Not only does such clustering facilitate mapping data objects back to disk pages, but it also improves processor cache performance when we manipulate a logical group of data objects. To avoid losing control of where a data object is placed in memory as is the case when calling a standard memory allocation routine such as `malloc`, we should pre-allocate large chunks of memory and manage data placement explicitly.
- *Data correlation.* Just placing a data object in a right memory location is not enough. We must further establish the correlation between the new data object with other data objects that are already stored in a computational cache. For example, two triangles sharing an edge should have pointers pointing to each other. We can search the existing data structure to find out to which other data objects the new object is correlated. However, the cost (e.g., $O(\log n)$) could be quite expensive. An alternative is to trade (memory) space for speed. We can set up auxiliary fast (e.g., $O(1)$) lookup data structures to speed up the association of the new object with the existing ones.

In summary, a translation mechanism enables us to maintain a dynamically evolving optimal data representation in a computational cache. The overhead of translation is just another price we have to pay in order to deliver data to applications in the right format.

3.4 Related Work

The proposed database approach that uses an application-specific computational cache is the first of its kind. Nonetheless, the core idea bears resemblance to that of several previous research efforts, including, in order of relevance, *architecture-conscious databases*, *object-oriented databases*, and *external memory algorithms*.

This section is not intended to be a detailed survey of the related work. But rather, it focuses on the main ideas coming out of the previous work and points out the similarities and differences between these ideas and that of the proposed solution.

3.4.1 Architecture-conscious databases

Architecture-conscious databases [2, 3, 17, 18, 37, 38, 76] represent an exciting new research area. The main focus is on how to exploit system (processor, cache, memory, disk) characteristics to significantly boost the performance of traditional database workloads such as online transaction processing (OLTP) and decision support system (DSS).

The idea of decoupling memory representation from disk page layout should be attributed to two pioneering works in this area: the *PAX model* and the *Clotho* architecture. Both address the issue of how data should be placed on disk pages and how data should be accessed in memory and cache.

In a traditional database, an N-ary Storage Model (NSM) is used to store records one after another on a slotted data page, as shown in Figure 3.4. The attribute values of a record are stored together and occupy a contiguous region on disk (also in main memory and the processor cache). When an SQL query examines the value of one attribute, other attributes of the same record are also loaded into the processor cache. Since the other attributes are not used, they practically pollute the cache and waste memory bandwidth.

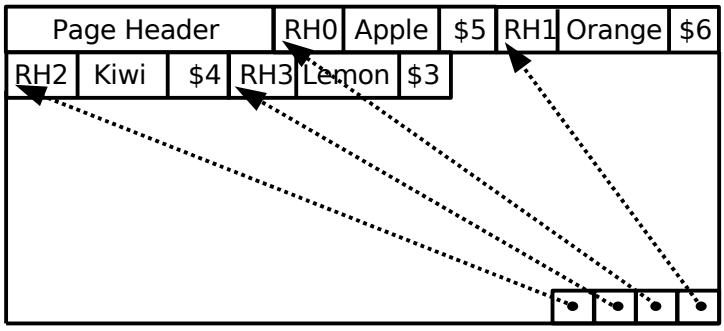


Figure 3.4: **An NSM data page layout.** For clarity, the slotted page is illustrated in a simplified way."RH" represents the record header, a metadata field associated with each record.

The PAX model

Recognizing this pitfall, Ailamaki and colleagues proposed an alternative data organization model called Partition Attribute Across (PAX) [2, 3]. The key new idea is to vertically partition the records on a page and group together all values of each attribute within the page. Figure 3.5 shows the PAX data organization for the records shown in Figure 3.4. Since attribute values are clustered, a cache line is always loaded with attribute values from different records. Since all of them need be accessed if a predicate (e.g., price $<$ \$5) is to be executed on the associated attribute, the cache performance and memory bandwidth utilization become optimal. If a full record needs to be returned, the scattered attribute values are collected and re-assembled.

Similarity. Both the PAX model and a computational cache strive to feed data to applications in the right format to boost performance. The record assembly process of the PAX model is a simple form of online data conversion, which mirrors the more complex translation mechanism required by a computational cache.

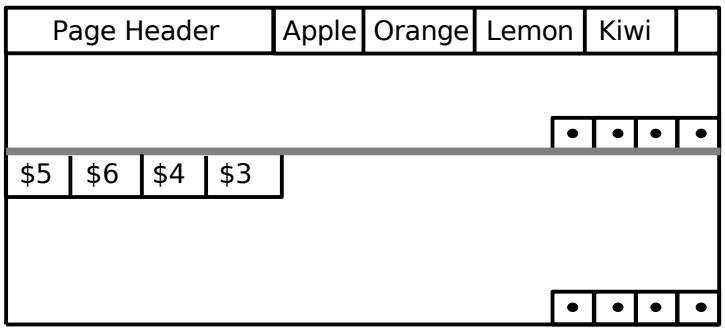


Figure 3.5: **A PAX data page layout.** Attribute values of different records on the same page are clustered together.

Difference. The PAX model stores the optimized data layout statically on disk. At runtime, the PAX data layout is accessed by a database engine without change. In contrast, a computational cache builds an optimized data representation dynamically at runtime.

The *Clotho* Architecture

Exploiting the idea originated from the PAX model, Shao and colleagues introduced a new buffer pool and storage management architecture called *Clotho* [76]. The main idea is to decouple in-memory page layout from data organization on disk to enable independent data layout design at each level of the storage hierarchy.

At the disk storage level, data are organized into *A-pages*. Similar to a PAX page, an A-page organizes data into mini-pages that group values from the same attributes for efficient predicate evaluation. *Clotho* gives storage devices the freedom to place the content of an A-page on disk in an optimal way.

In the main memory, data are re-organized into *C-pages*. A C-page is similar to an A-page in that it also has attribute values grouped into mini-pages. But unlike an A-page, a C-page contains only values of the attributes that a query needs to access. There is no one-to-one mapping between A-pages and C-pages. The content in a C-page can be extracted from a number of A-pages.

The strength of the *Clotho* architecture is that the data page layout in memory is not tied to the data page layout on disk. The particular in-memory page organization at an instance is determined by the data access need of the current query. As a result, *Clotho* is able to efficiently support workloads that have dynamically changing data access needs.

Similarity. The core idea of computational database system is a close cousin to that of the *Clotho* architecture. Both advocate decoupling data representations between memory and disk. Both support dynamic data conversion at runtime.

Differences. Although similar in concept, the *Clotho* architecture and the computational database framework are different in a number of major ways:

- The *Clotho* architecture targets traditional database workloads; the computational database framework targets unstructured scientific applications.

- In the *Clotho* architecture, data are converted into fixed-size C-pages, while in a computational database system, data are converted and attached to a free-form optimal data structure.
- The data conversion between A-pages and C-pages is syntactic. In essence, the operations are a sequence of permutation and sub-setting of the original data records (tuples). The conversion process is relatively straightforward. In contrast, the conversion in a computational database system is at the semantic level. We have to understand the logical property such as the spatial or temporal features of a data set in order to carry out the conversion. As a result, the translation mechanism is much more "heavy-weight" in terms of the CPU cycles and the memory usage.
- The *Clotho architecture* decouples data layout *within* the page buffer pool. A computational database system decouples data layout in a computational cache that sits *on top of* the page buffer pool. This difference exhibits the complementary nature of the two techniques. If we combine the two together, a computational cache will be able to fetch C-pages instead of A-pages. Interestingly, in this case, data traveling from disk to processor cache will undergo a series of non-trivial format transformations. Fortuitously, thanks to the additional computing power available on multicore processors, the computational overhead of data format conversion should not be a problem in the near future as long as we can improve the overall performance of the applications.

Although different from the computational database framework in design and implementation, both the PAX model and the *Clotho* architecture have provided convincing evidence showing the promise of decoupled data representations. This dissertation takes a step further and demonstrates that decoupling data representation at the semantic level can lead to new breakthrough capabilities.

3.4.2 Object-oriented databases

Object-oriented databases (OODB) integrates DBMS functionality with a programming language with the goal to eliminate impedance mismatch [44, 105]. The technical solutions are based on *persistent programming languages* [22, 73] such as persistent C++. The main features of OODBs are: (1) variables of a persistent programming language could represent both disk-based data and main memory data, (2) database search criteria are formulated using the programming language constructs instead of SQL.

The main target of the OODB community is engineering applications.[2] A general form of these applications is to open a large engineering object, for example, an electronic circuit, and then process it extensively before closing it. Such objects are read into virtual memory by a load program. In the process of loading, pointers of the objects are converted from the disk representations to main memory representations. When the code finishes processing the object, the C++ data structure is linearized and stored back to disk.

Similarity. OODB's idea of converting data between disk representation and memory representation is similar to that of a computational database system. The translation (conversion)

[2]Unfortunately, all of the OODB vendors have failed [44].

process requires an OODB to understand both the data layout on disk and the data format of the programming language.

Difference. Different from OODBs, a computational database system converts data representation at the semantic level (translation of inherent data set structures) instead of at the syntactic level (translation of object pointers). Not coupled with a programming language, a computational database system has the flexibility to convert data into any free-form representation that is best for the applications.

A second difference lies in how data are loaded into memory. Instead of loading objects upfront, a computational database system carries out on-demand data conversion at the page granularity. Applications can work on part of the data sets that have already been stored in the computational cache.

3.4.3 External Memory Algorithms

External memory algorithms, also known as out-of-core algorithms solve large-scale computational problems involving massive data [91, 102]. These algorithms incorporate locality directly into the algorithm design and explicitly manage the contents of each level of the memory hierarchy, thereby bypassing the virtual memory system. In practice, a large number of external memory algorithms have been designed and implemented to solve problems ranging from N-body simulations [70] to streamline visualization [100]. In a sense, a database system is also a collection of external memory algorithms and data structures, which happen to have been optimized for dealing with massive tabular data.

Most external memory algorithms organize data on disk a way that can optimize performance of a particular application. Intermediary results are dumped to disk in any way an algorithm sees fit. Neither data persistence nor query capability is a concern. Disks are used as a temporary staging area rather than a permanent data store.

Similarity. Like a computational database system, external memory algorithms provide application programs with optimal incore data representation, which is often far from optimal for storage purpose. To reduce I/O costs, external memory algorithms also manage I/O explicitly.

Difference. In contrast to external memory algorithms, a computational database system stores data in slotted pages. Runtime format translation has to be carried out to map data properly to an optimized incore representation.

While a computational database system supports queries on data sets after computational algorithms finish execution, external memory algorithms produce ad-hoc formatted output that are not suitable for data queries.

Regardless of their relevances, the influence of these prior works has melted into the design of the computational database framework one way or another. The description in this chapter lays out a conceptual model for building real systems. In the next chapter, we demonstrate how to turn the blueprint into the implementation of a prototype system.

Chapter 4

Implementation

The proposed database approach looks promising, at least on paper. To turn this vision into reality, we must address a myriad of technical challenges. The first and the most important of all is whether the proposed database framework is feasible. In particular, what exactly should be stored in a computational cache? How does it work? How can we translate data? What is the overhead of data translation? And above all, can application programs run faster operating directly on a computational cache?

These problems are addressed in this chapter and the next. As mentioned earlier, it is beyond any reasonable expectation to have one cure-all solution. Therefore, we return to our original problem and demonstrate the feasibility of a computational database system solution in the context of Delaunay triangulation.

This chapter describes the implementation of our methodology within a a prototype system called *Abacus* that deals with massive triangulation datasets. The abstract design principles outlined in the previous chapter are materialized into concrete data structures and algorithms.

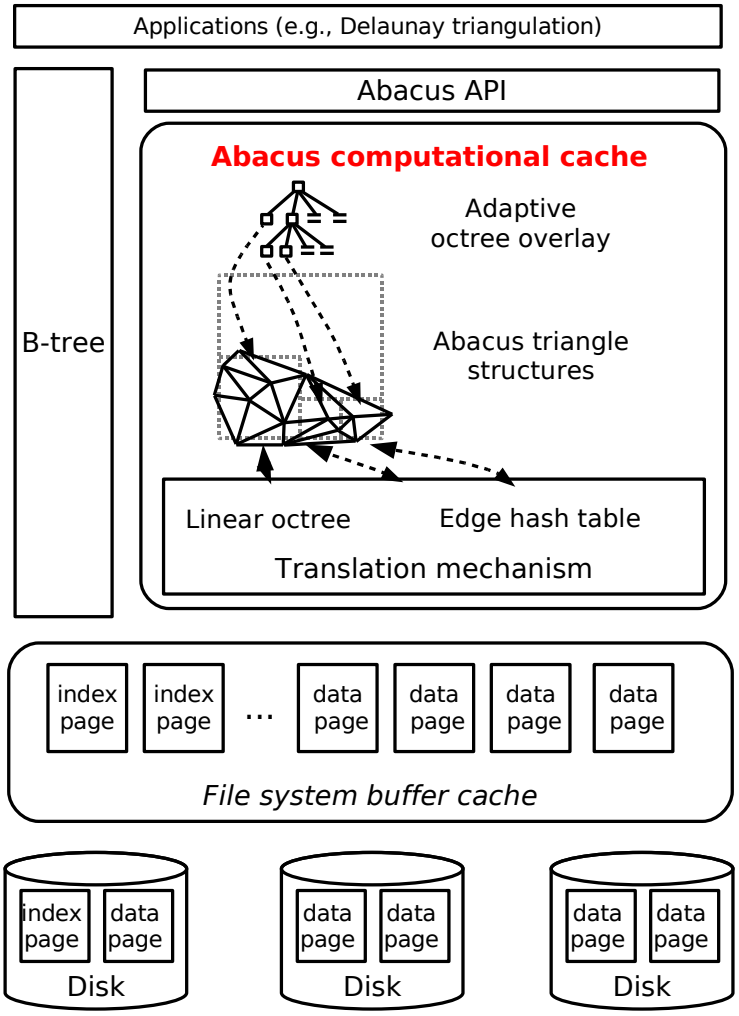


Figure 4.1: **Abacus system overview.**

4.1 The Abacus System Overview

The generic computational database framework we proposed is not tied to any specific scientific datasets or applications. In order to validate its feasibility, we have instantiated it into a system implementation called *Abacus* whose overall structure is shown in Figure 4.1. The Abacus system 2D and 3D Delaunay triangulation datasets. It is mainly designed and implemented to re-attack the Delaunay triangulation problem that has frustrated us in Chapter 2.

A decision has been made to implement the system from a clean slate instead of on top of an existing database system. The benefit of building a standalone system is that we can avoid unrelated database complexity and overhead that could interfere with the implementation and evaluation of the computational cache and the translation mechanism. This may appear to be against the proposed strategy of integrating a computational cache with an existing database

system as explained in Section 3.1. But in fact, it is not. Besides the B-tree index, none of the existing database functions are re-implemented within Abacus. Porting Abacus to an existing system does not require rewriting modules in the current (Abacus) implementation. Besides, since we do not go through the SQL layer of a traditional database system, we completely avoid the impedance mismatch and thus are able to concentrate on the effectiveness of our solution of overcoming the structural mismatch. As a result, the components shown in Figure 4.1 are slightly different from those in the generic computational database framework. The SQL query processing layer is not included; and the user-defined function/user-defined data type layer is replaced with application programs.

The computational cache within the Abacus system consists of a number of optimized data structures and algorithms. A small Application Programming Interface (API) is exported to application programmers. For brevity, we use the terms *computational cache*, *Abacus computational cache*, and *Abacus cache* interchangeably, all of which refer to the box captioned *Abacus Computational Cache* as shown in Figure 4.1.

We have developed a B-tree implementation instead of using the R-tree index we developed in Chapter 2. There are two reason for this design choice. First, the worse-case search time (also the average-case search time) of the B-tree is $O(\log n)$, while the worse-case search time of an R-tree is $O(n)$. Second, all existing database systems have built-in support for B-tree indices, which is not the case for R-trees. (For example, the R-tree is not supported in the IBM DB2 Universal Database.[1]) The practical implication is that when we port Abacus into a full-fledged database system, there will not be technical difficulties due to the lack of a proper index structure.

At the storage level, we have chosen to use the standard file system buffer cache to manage fixed-size database pages. Although it has been pointed out that the OS buffer pool is not the ideal solution for database applications [86], we use it simply for convenience. When a fully functional database page buffer pool manager replaces the file system in the future, we expect to the performance of the Abacus system to improve.

The rest of this chapter provides an anatomy of the Abacus system. In most of the technical discussions that follow, we use a simple 2D triangulation described in the next section to illustrate concepts and motivate solutions. Nevertheless, all the algorithms, data structures and techniques have been designed and implemented for both 2D and 3D cases.

4.2 A 2D Triangulation Example

Figure 4.2 shows a simple 2D triangulation with 6 vertices, numbered from 0 to 5. A triangle is marked by $\mathtt{T_i}$, where `i` is a unique identifier for the triangle.

The example triangulation is embedded in a square with the origin, also called the near end point, at $(0, 0)$ and the far end point at $(8, 8)$. In real-world applications, the dimension of a problem domain is usually known beforehand. For example, the size of a turbine engine or the geographic span of terrain map are always known. Without loss of generality, we assume an application can specify both the near and far end points. The problem domain does not need to be a square, though. It only has to be a rectangle, which we can always conveniently embed into

[1] After IBM acquired Informix, R-tree indices were supported in the IBM Informix Dynamic Server (IDS).

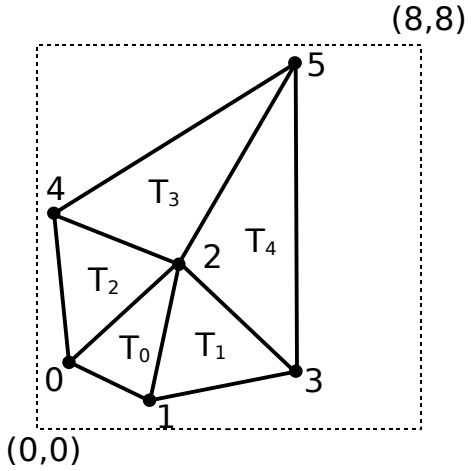


Figure 4.2: **A 2D triangulation embedded in a square.**

a square.

4.3 Database Organization

We organize a triangulation dataset in two database tables, one for the triangles and the other for the vertices. To facilitate the discussion, we assume the size of disk pages is 64 byte. In reality, the size is 8 KB or 16 KB.

4.3.1 Triangle Table

A triangle table contains the triangle records of a triangulation dataset. The table has three attributes, $< \texttt{vid}_0, \texttt{vid}_1, \texttt{vid}_2 >$, where the `vids` are the ids of the three vertices of a triangle. The triplet is stored in the counterclockwise order. It uniquely identifies a triangle and is the *primary key* of the table.

Additional attributes are often required in practice. They usually represent satellite data, which are carried around with the primary keys. For the purpose of this thesis, we ignore these attributes in our discussion.

The triangle table for our example triangulation is shown in Figure 4.3(a). The $\texttt{T}_\texttt{i}\texttt{s}$ listed to the left of the records are for illustration purpose only. They are not stored in the triangle table. Note that when $\texttt{vid}_0$ of a triangle is chosen, $\texttt{vid}_1$ and $\texttt{vid}_2$ become uniquely determined automatically due to the counterclockwise order requirement. Note that $\texttt{vid}_0$ does not need to be the one with the smallest vertex id. For example, triangle $\texttt{T}_4$ is represented as $< 5, 2, 3 >$.

We store a triangle table in slotted disk pages, which we refer to as the *triangle data pages*. We assume the header field of a slotted page consumes 4 bytes. The triangle records are placed one after another after the header field. In addition, we assume the `vids` are defined as 4-byte integers. Hence, each triangle record is 12 bytes in size (i.e., 4 bytes $\times$ 3). At most 5 triangle records can be stored in our hypothetical disk page (i.e., 64 bytes = 4 bytes (header) + 12 bytes

	vid_0	vid_1	vid_2
T_0	0	1	2
T_1	1	3	2
T_2	0	2	4
T_3	2	5	4
T_4	5	2	3

(a) A triangle data.

Header			0	1	2	1	3	2
0	2	4	2	5	4	5	2	3

(b) Triangles on a slotted page.

Figure 4.3: **The organization of a triangle table.**

(triangle record) $\times$ 5). Figure 4.3(b) shows how the triangle records of the 2D example are laid out on a slotted page. (The size of the header field is artificially enlarged for clarify.)

A digression is due to clarify how the counterclockwise order is defined for 3D tetrahedrons. As shown in Figure 4.4, we define a tetrahedron $< \texttt{vid}_0, \texttt{vid}_1, \texttt{vid}_2, \texttt{vid}_3 >$ to be in a counterclockwise order if curling our right-hand palm along the plane of the first three vertices results in the thumb pointing to the fourth vertex.

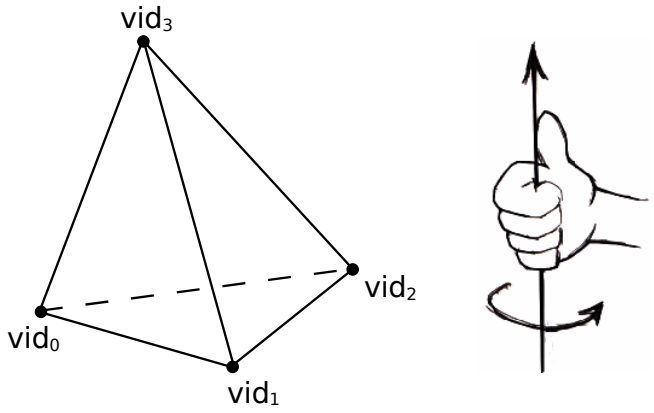


Figure 4.4: **A tetrahedron in the counterclockwise order.**

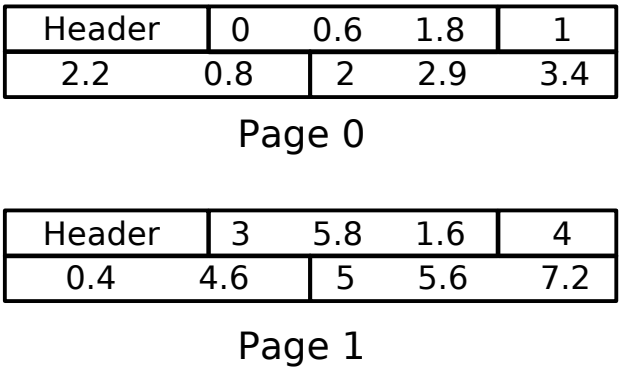
4.3.2 Triangle B-tree Page Index

In order to keep track of the triangles stored in the triangle table, we build a B-tree index. However, we do not use the primary keys ($< \texttt{vid}_0, \texttt{vid}_1, \texttt{vid}_2 >$) to index individual triangles. Instead, we use a technique called *locational codes* to index triangles one data page at a time. We refer to the B-tree index as the *triangle B-tree page index*. Roughly speaking, a locational code encodes the common spatial properties of the triangles stored in a data page.

Since the construction of a locational code is closely related to how we group triangles together in memory, we defer the discussion on how to compute a locational code for a triangle data page to Section 4.8.3.

vid	x	y
0	0.6	1.8
1	2.2	0.8
2	2.9	3.4
3	5.8	1.6
4	0.4	4.6
5	5.6	7.2

(a) A vertex data.



(b) Vertices on 2 slotted page.

Figure 4.5: **The organization of a vertex table.**

4.3.3 Vertex Table

A vertex table contains the vertex records of a triangulation. The table has three attributes, $<$ `vid`, `x`, `y` $>$, where the `vid` is the id of the vertex, `x` and `y` are the coordinate of the vertex. The `vid` attribute is the primary key.

Figure 4.5 shows the vertex table for our example. The vertex table is also stored in slotted disk pages, which we refer to as the *vertex data pages*. We assume the coordinates of a vertex are 8-byte doubles. The size of each vertex record is 20 bytes (i.e., 4 bytes (`vid`) + 8 bytes (coordinate) $\times$ 2). Using the same disk page size of 64 bytes and the same header fields size, we can place at maximum 3 vertex records on a disk page ((i.e., 64 bytes = 4 bytes (header) + 20 bytes (vertex record) $\times$ 3). Figure 4.5(b) shows how the vertex records of the example are laid out on two slotted pages.

Note that when we store the vertex records in a slotted page, we do place them one after another according to an ascending vertex id order. There is no such requirement when we store triangle records.

4.3.4 Vertex B-tree Page Index

We use a second B-tree to manage vertices in a vertex table. Again, we do not index individual vertices but index the vertex data pages. The key assigned to a vertex data page is the vertex id of the first record on that page, which is also the smallest vertex id on the page. We refer to the B-tree as the *vertex B-tree page index*.

Although using a different key for indexing purpose, the vertex B-tree page index is similar to the triangle B-tree page index in that both return one data page at a time instead of individual data objects. As such, we are able to prefetch data objects on the same data page where the target object (i.e., the one we are looking for) is stored and amortize the cost (i.e., $O(\log n)$ per search) of searching B-trees.

Figure 4.6 summarizes the four different database structures we have introduced. Each structure is stored in a separate regular file.

Triangle table	Slotted data pages storing triangle records
Triangle B-tree page index	A B-tree index managing triangle data pages
Vertex table	Slotted data pages storing vertex records
Vertex B-tree page index	A B-tree index managing vertex data pages

Figure 4.6: **Summary of database structures for storing a triangulation dataset.**

4.4 Data Structures in the Computational Cache

From an application's perspective, the natural representation of a triangulation dataset is one that supports efficient traversal of the topological structure. That is, moving from one triangle to another across a shared edge. In practice, there are two popular data structures that effectively represent the topological structure of a triangulation: the *quad-edge* data structure [35] proposed by Guibas and Stolfi, and the *triangle-based* data structure [79] proposed by Shewchuk. Between the two, the triangle-based structure is more memory efficient and is the one we choose use in Abacus. We refer to it as the *Abacus triangle structure*.

Figure 4.7 shows how triangle $\mathtt{T}_0$ is represented within the Abacus triangle structure. A piece of memory called *simplex entry* is allocated for $\mathtt{T}_0$. The entry records three pointers to neighboring triangles, $\mathtt{T}_1$, $\mathtt{T}_2$, and NULL as there is no neighboring triangle sharing edge $(0, 1)$, and three pointers to the vertices, $0, 1, 2$, which are allocated elsewhere within the Abacus cache.

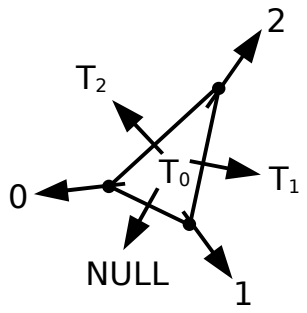


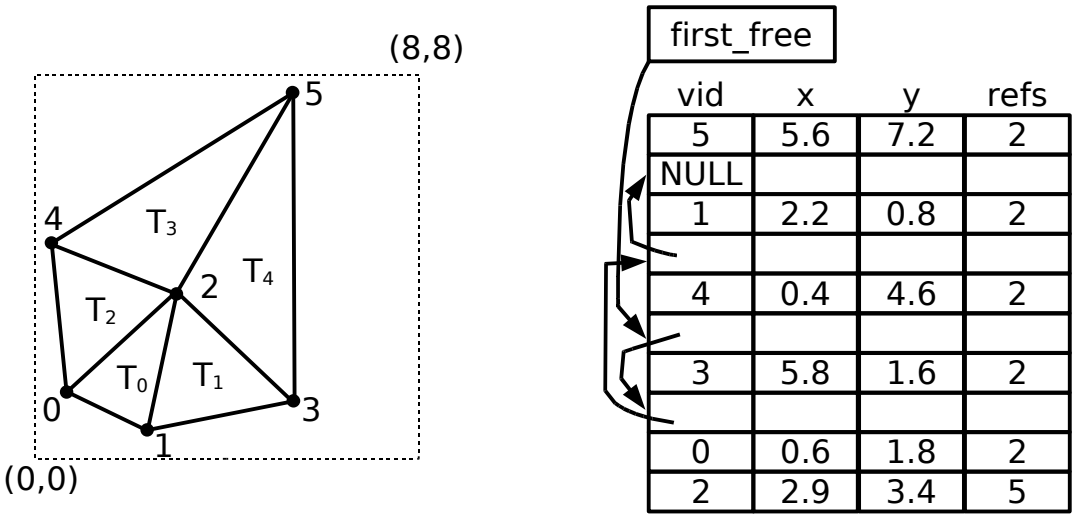
Figure 4.7: **The Abacus triangle structure of $\mathtt{T}_0$.**

This section lists that data structures for the different geometry entities (vertices, triangles, and edges) within the Abacus cache. The next section explains how we manage these data structures.

4.4.1 Vertex Heap

When a vertex record is fetched from the database, it is converted into a slightly different form. Besides the vertex id and the coordinate, a reference count is associated with the vertex, which records how many *cached* triangles share the vertex.

To distinguish the two different representations, we refer to the records in the database as the *out-of-core records* (even though they may have been cached in the buffer cache) and the records



(a) The example triangulation. (b) A vertex heap layout.

Figure 4.8: **Incore vertex records stored in a vertex heap.** We assume the five triangles shown in (a) are all cached in the Abacus cache.

within the Abacus cache as the *incore records*. The same convention is used when we discuss triangle representations.

An incore vertex record is placed on a heap that is allocated at initialization time, that is, at the time an Abacus database is opened (see Section 4.11). The size of the heap is prescribed and does not change over time. We call the heap the *vertex heap*. It consists an array of entries for storing incore vertex records. When incore vertex records are placed on the heap, they are not necessarily clustered in a contiguous region. Unused entries may be scattered across the array. They are managed by a linked list. The pointers used by the linked list are directly installed inside the unused entries.

Figure 4.8(b) shows how the vertices of our example triangulation are stored in a vertex heap with 10 entries. For convenience of reference, the original example is replicated in Figure 4.8(a). We assume all the 5 triangles, $\mathtt{T}_0, \mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3, \mathtt{T}_4$, are already cached in the Abacus cache. The reference count (`refs`) show how many incore triangle records share a vertex, not how many edges are incident to the vertex. For example, vertex 2 has a reference count of 5 since all the triangles share it. The `first_free` pointer in the figure points to the head of the unused entry list.

4.4.2 Vertex Hash Table

When a new triangle is generated or loaded into the computational cache, we need to know if its vertices have been cached in the vertex heap, and if so, the memory address of the cached vertices (so that we can construct the Abacus triangle structure). Because the heap is a slow search structure ($O(n)$ per search), we build a hash table called the *vertex hash table* to keep track of the incore vertex records.

The size of the vertex hash table is prescribed at initialization time. We use a division method

to index into the table and use a doubly-linked overflow chain to manage collision. Each hash entry contains a pointer an incore vertex record.

To avoid the pitfalls of the division method (i.e., not all the bits of a key are used), we scramble a vertex id first before using as a search key. As a secondary optimization, we set the hash table size to a power of 2 and use a bit-mask to extract the lower-order bits of the scrambled keys to avoid division operations.

Figure 4.9 shows how the incore vertex records are tracked by a hash table of size 4. Note that we do not use the vertex ids as keys directly. The keys are the output of a deterministic scrambling process. Hence, vertex 0 and 2 are hashed into the same slot.

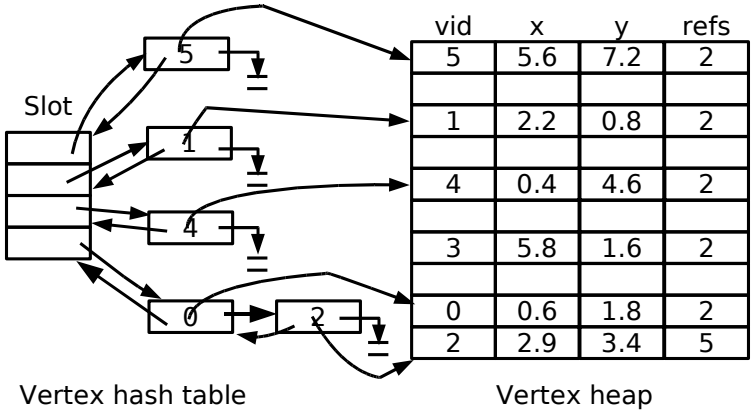


Figure 4.9: **A vertex hash table tracking incore vertex records.** The free entry list within the vertex heap is omitted.

4.4.3 Simplex Blocks

The counterparts of the database triangle data pages inside the Abacus cache are the *simplex blocks*, although the triangle representations of the two are vastly different.

The number of simplex blocks is also determined at initialization time and does not change afterwards. A simplex block is organized in the same way as the vertex heap. The number of *simplex entries* in a simplex block equals to the maximum number of triangle records that can fit in a triangle data page. Each entry holds one incore triangle record with 6 pointers. Three of the pointers point at the vertex hash entries that correspond to the vertices of the triangle. The other three point at neighboring triangles. If a neighbor is not cached, the corresponding pointer is NULL. A neighbor triangle may be stored in another simplex block and it is all right to have cross simplex block pointers.

Figure 4.10 shows how the 5 triangles of our example are cached in one simplex block. To clarity, the pointers of the incore triangle records are not shown in the figure. The ordering of triangles within the simplex block is deliberately shuffled. There is no correspondence between where an out-of-core triangle record is stored in a slotted page and where an incore triangle record is stored in a simplex block. Note that we reserve a small amount of memory at the

beginning of each simplex block to store meta data, which record the number of free entries and other control information.

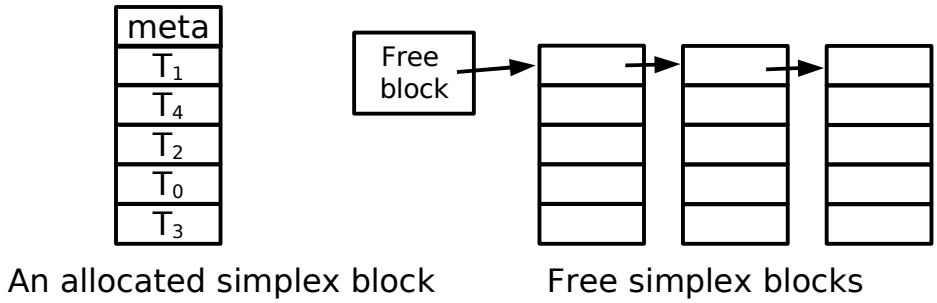


Figure 4.10: **Storing incore triangle records in a simplex block.** There are actually pointers pointing to one another within the simplex block. However, for clarity, those pointers are not drawn. The `meta` field is a small chunk of memory reserved for meta data.

The main use of simplex blocks is to group spatially close triangles together in contiguous memory address space. Section 4.5.2 explains how to assign a triangle to an appropriate simplex block.

4.4.4 Edge Hash Table

We use a hash table called *edge hash table* to keep track of *visible* edges. An edge is said to be visible if there is only one incore triangle that uses the edge. The other triangle sharing the same edge is either on disk or non-existent (for example, a convex hull edge is only used by one triangle). The edge hash table provides a fast means for a newly loaded or created triangle to find its neighbors and set pointers to them.

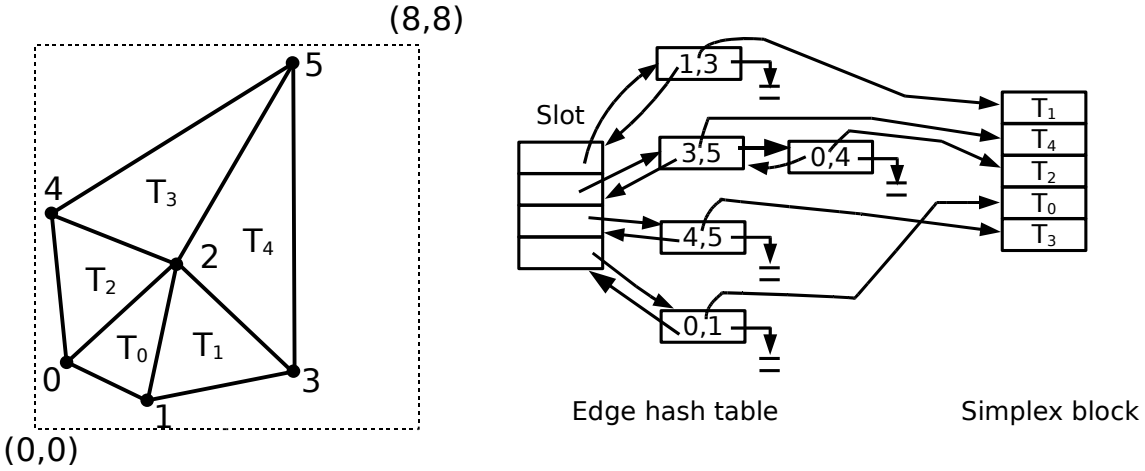
An edge hash entry has three fields $(\texttt{vid}_0, \texttt{vid}_1, \texttt{ptr\_triangle})$. The $\texttt{vid}_0$ and $\texttt{vid}_1$ fields record the vertex ids of the end points of the edge. We require that $< \texttt{vid}_0, \texttt{vid}_1 >$ be an ordered pair such that $\texttt{vid}_0 < \texttt{vid}_1$. The `ptr_triangle` field points to the triangle that shares the edge.

Figure 4.11(b) shows how the status of an edge hash table after the 5 triangles of our example are loaded into a simplex blocks. We assume the size of the hash table is 4. Note that edges shared by two triangles, for example, edge (2,5) and (2,4), are not hashed. Only the ones on the convex hull are hashed.

In summary, the data structures described so far are collectively known as the Abacus triangle structure. The individual components and their usage are listed in Figure 4.12. The most important among these is the simplex blocks. The data structures for storing vertices and edges are updated as a result of changes in the simplex blocks.

4.5 Computational Cache Memory Management

This section provides an overview of how we use the various data structures to allocate and deallocate vertices, triangles, and edges, respectively.



(a) The example triangulation. (b) An edge hash table.

Figure 4.11: **Edge hash table tracking edges used by only one triangles.**

Vertex heap	A heap storing incore vertex records
Vertex hash table	A hash table keeping track of cached vertex records
Simplex blocks	Fixed-size blocks storing incore triangle records
Edge hash table	A hash table keeping track of visible edges

Figure 4.12: **Summary of the Abacus triangle structure.**

4.5.1 Vertex Memory Management

New incore vertices are allocated from from the vertex heap. If there are free entries left on the heap, that is, the `first_free` pointer in Figure 4.8(b) is not NULL, the entry pointed at by `first_free` is returned and the free entry list is updated accordingly. If there are no more free entries, we go through the entire vertex hash table and check each entry's reference count. If the reference count is 0, the vertex heap entry is freed and its associated hash entry is also deleted. After all the hash entries are checked, we either acquire a vertex record entry and proceed or fail to reclaim any memory and have to exit with an out-of-memory error.

Incore vertex records are deleted from the heap indirectly. When a incore triangle record is deleted, either because the triangle is to be swapped to disk or the triangle is deleted, the reference counts of its three vertices decrement by 1. When the reference count of a vertex drops to 0, the incore vertex record is deleted from the heap and the corresponding hash entry removed.

It may appear strange that how there could be vertex records with reference count 0 existing on the heap while we have a policy to reclaim memory whenever the reference count drops to 0. The existence of such vertex records is due to the effect of prefetching, which will be explained in Section 4.7.

Figure 4.13 shows that a new triangle T_5 (6,5,3) is added into the Abacus cache. The details of how the triangle is inserted is explained the next section. For the time being, let us just focus on the operations in the vertex heap and vertex hash table. Figure 4.14 shows the updated vertex heap and vertex hash table. The vertex 6 is allocated the entry pointed at by the `first_free`

pointer as shown in Figure 4.8(b). Its reference count is set to 1 since only T_5 uses it. A new hash entry is created and in vertex hash table and linked into the overflow chain. Note the reference counts of vertex 3 and 5 increment by 1, respectively, as there is now one more triangle sharing them.

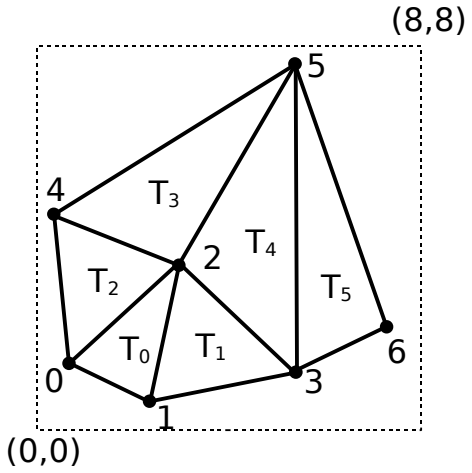


Figure 4.13: **Adding a new triangle into the Abacus cache.**

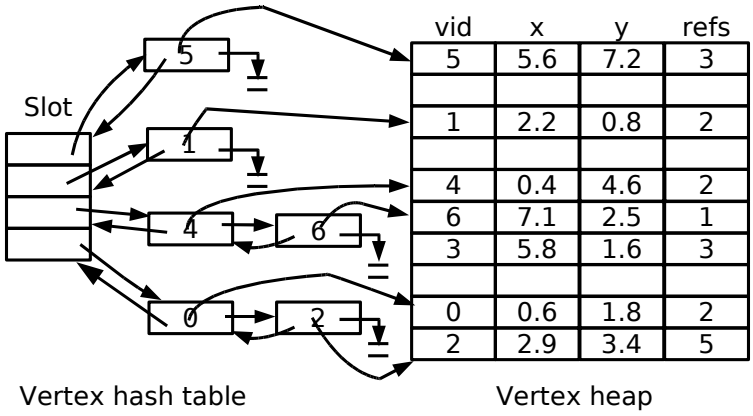


Figure 4.14: **Updated vertex heap and vertex hash table.**

4.5.2 Simplex Memory Management

The design goal of simplex blocks is to cluster spatially close triangles together in contiguous physical memory. There are two major advantages if the goal can be achieved. First, applications operating on a triangulation dataset, such as Delaunay triangulation and iso-contour extraction, exhibit strong spatial and temporal locality of reference. They tend to operate in a small region of the triangulation intensely for a short period of time and then move to the next (nearby) region. By clustering spatially close triangles in contiguous physical memory, we can make better use of

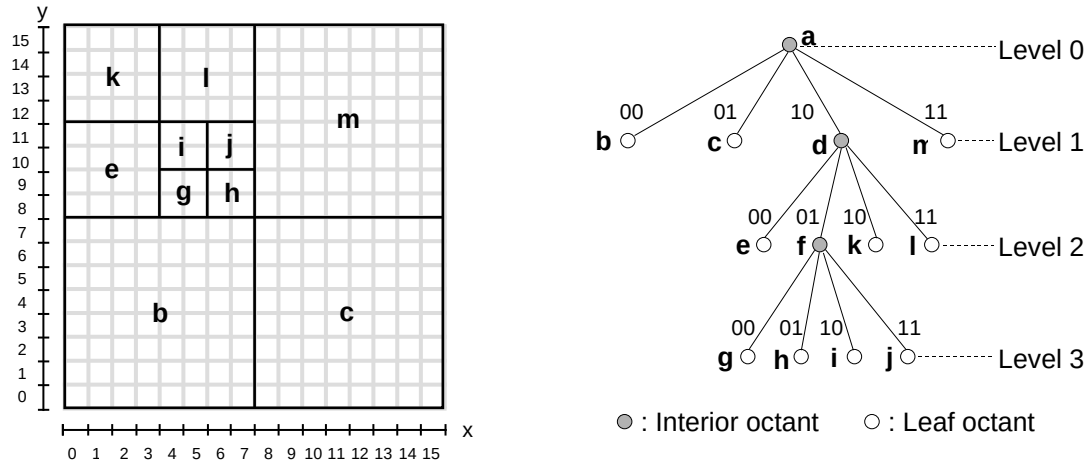
the processor cache. Second, when we map a simplex block to a triangle data page, there is no need to chasing pointers in the memory. All the triangles that need to be swapped out are stored in the same simplex block.

The questions are, (1) How can we group spatially close in the same simplex block? (2) How do we split the data when a simplex block becomes full? and (3) What should we do if all the simplex blocks are used up?

Our solution is to overlay an an octree on top of the Abacus triangle structure and use the octree as a memory management tool.

Octree Structure

An octree recursively subdivides a 3D problem domain into 8 equal size octants until certain criterion is satisfied.[2] We can interpret an octree in two equivalent ways: the *domain representation* and the *tree representation*. A *domain* is a Cartesian coordinate space that consists of a uniform grid of $2^n \times 2^n$ indivisible *pixels*, that is, the smallest octant representable. Figure 4.15(a) shows a domain representation of an octree decomposition of a domain of $2^4 \times 2^4$. The uniform grid in the background consists of the pixels of the domain. The equivalent tree representation is shown in Figure 4.15(b). The *root octant* spanning the entire domain is defined to be at level 0. Each child octant is one level lower than its parent and is half as large (in edge size). Each tree edge in Figure 4.15(b) is labeled with a *directional code* that distinguishes the children of each parent octant. Figure 4.16 shows the interpretation of the directional code in the context of the domain representation.



(a) Domain representation. (b) Tree representation.

Figure 4.15: **Octree decomposition.** Note that the interior octants `a, d, f` are not shown in the domain representation, which contains only leaf octants.

In the rest of this chapter, we will use the directional code to refer to the four sibling octants, with (00) representing the octant in the left-lower corner, (01) the octant in the right-lower corner,

[2]We use a 2D quadtree to illustrates the concepts.

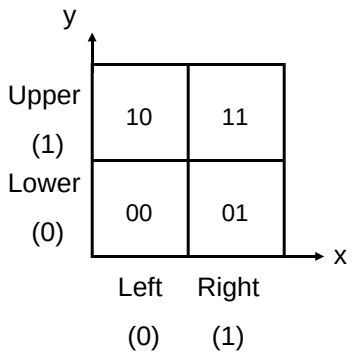


Figure 4.16: **Two-bit encoding of a directional code.**

(10) the octant in the left-upper corner, and (11) the octant right-upper corner. This ordering is the same as the left to right ordering as the siblings are laid out in the tree representation.

How to Group Spatially Close Triangles in the Same Simplex Block?

We determine the closeness of triangles using a simple criterion: Triangles whose *centroids* fall within the same leaf octant in an octree are regarded as spatially close to one another. The centroid (C) of a triangle $\triangle abc$ is defined as follows.

$$C_x = \frac{a_x + b_x + c_x}{3} \tag{4.1}$$

$$C_y = \frac{a_y + b_y + c_y}{3} \tag{4.2}$$

A centroid is the geometry center of a triangle, which gives some indication of where a triangle is in the domain. As stated in Section 4.2, we always embed a problem domain into a square region, which can be mapped to a root octant. Hence, the centroid of *any* triangle consisted of vertices from within the domain must fall inside the root octant. In contrast, as shown in Figure 4.17, the circumcenter of a triangle may be far way from where a triangle is and could be outside of a pre-defined domain. A centroid is also fast to compute. In contrast, computing circumcenters requires evaluation of non-trivial determinants as shown in Equation 2.1 and 2.2.

Since we make use of the octree structure to aggregate spatially close triangles, it is a natural choice for us to use the leaf octants to manage the simplex blocks. Initially, all simplex block are marked as free. At runtime, simplex blocks are dynamically assigned to leaf octants. Each leaf octant can be assigned one simplex block at most. A pointer is installed in a leaf octant to point to the simplex block it manages. Conversely, a pointer is installed in a simplex block meta data field to point to its assigned leaf octant.

To allocate space for a new incore triangle, we compute its centroid and search the octree to find the enclosing leaf octant. If the simplex block associated with the enclosing leaf octant has free entries, we allocate an entry for the new triangle. Otherwise, we split the simplex block as will explained next.

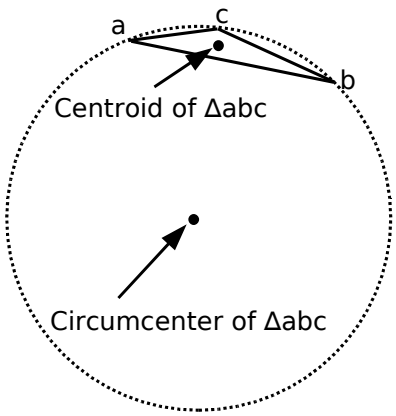
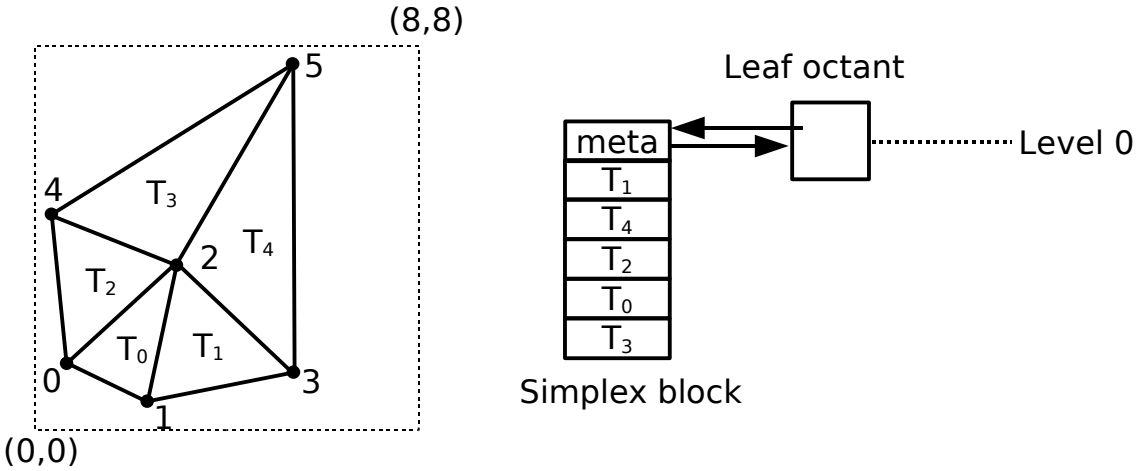


Figure 4.17: **The position of the centroid of $\triangle abc$ vs. that of the circumcenter.**



(a) The original example. (b) The root (leaf) octant manages a simplex block.

Figure 4.18: **Managing simplex blocks using leaf octants.**

Figure 4.18(a) shows the original triangulation (without the addition of the new triangle T_5). We map the domain (from (0,0) to (8,8)) to the root octant, which is also the sole octant of the octree shown in Figure 4.18(b).

How to Split a Full Simplex Block?

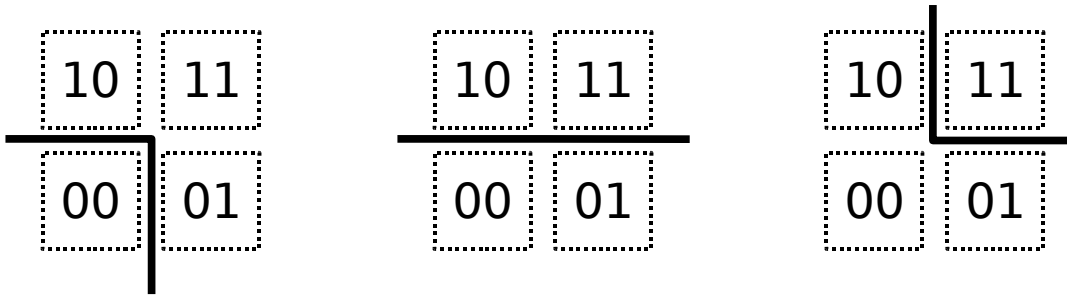
Let us study the case of inserting a new triangle T_5 (as shown in Figure 4.13) into the Abacus triangle structure. We first compute the centroid of T_5 and search the octree. Since there is only one octant, the root. We arrive at a leaf octant trivially. Examining the simplex block associated with the leaf octant, we find there are no more free simplex entries available. Recall that the capacity of a simplex block equals to that of a triangle data page, which, in our simple example, is equal to 5. At this point, we must split the full simplex block and grow the octree properly. By "splitting", we mean allocating new simplex block(s) and distributing the incore triangle records

across the old and new simplex blocks.

Octree structural change requires a leaf octant be split into 4 children in 2D and 8 in 3D. However, if we allocate 3 new simplex blocks and assign them along with the old (splitting) simplex block to the 4 new children octant, we make a one-to-four split, which results in an average memory and disk utilization rate of 25% if the triangles are uniformly distributed in the domain. Worse, if the distribution is extremely skewed, we may end up with a large number of almost empty simplex blocks. Not only are the memory space and bandwidth wasted, but the disk space and I/O bandwidth are wasted since we need to map simplex blocks to triangle data pages.

To avoid this pitfall, we emulate the one-to-two split method used in the B-tree. We organize the new children octant in two groups such that *the centroids* of the triangles stored in the full simplex block are distributed (roughly) evenly between the two groups. Each group is assigned a simplex block that is shared among the siblings of the group. The incore triangle records are then physically re-distributed between the two simplex blocks. As such, we improve the memory and disk utilization rate (immediately after a split) to 50%.

For reasons to be explained Section 4.10, we require that a split must be placed along the directional code ordering of the children, that is, a cut is made somewhere between the leftmost child and the rightmost child. Figure 4.19 shows 3 possible scenarios in 2D.



(a) (00), (01, 10, 11). (b) (00, 01), (10, 11). (c) (00, 01, 10), (11).

Figure 4.19: **Three scenarios of splitting 4 children leaf octants into 2 groups.** We have three choices along the sequence (00, 01, 10, 11).

When a group contains only one octant, the octant has exclusive use of a simplex block. Triangles whose centroids fall inside the octant are distributed to the exclusive simplex block.

When a group contains multiple sibling octants, the group share a simplex block. The first sibling sets a pointer to the shared simplex block; other siblings set pointers to the first sibling and access the shared simplex block indirectly. Triangles whose centroids fall inside any of the octants of the group are distributed to the shared simplex block.

Figure 4.20 shows how the centroids of the 5 triangles of our example triangulation are distributed among the children of the root octant. The centroids are marked as four-pointed stars in the figure. Since the centroids of $\mathtt{T}_0, \mathtt{T}_1, \mathtt{T}_2$ are distributed inside the first child (00) and the two remaining centroids of $\mathtt{T}_3, \mathtt{T}_4$ are distributed inside the other children, we split the children according to scenario (a) of Figure 4.19.

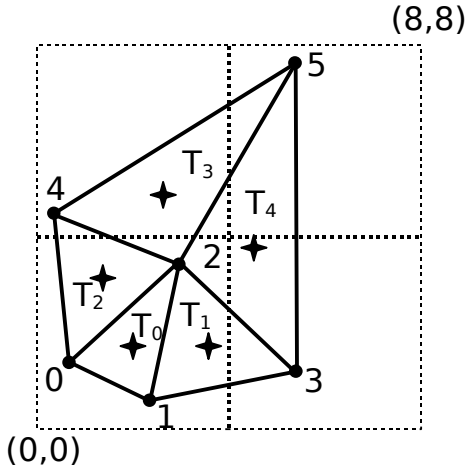


Figure 4.20: **Centroid distribution of existing triangles among the children octants of the root.**

Figure 4.21 illustrates the result of the splitting. The old (splitting) simplex block is transferred from the root octant to the first child (00). A new simplex block is allocated and associated with the first sibling (01) of the second group. The other sibling of the group, (10) and (11), set their pointers to the first sibling (00). The existing 5 triangles are physically re-distributed between the simplex blocks based on where their centroids fall.

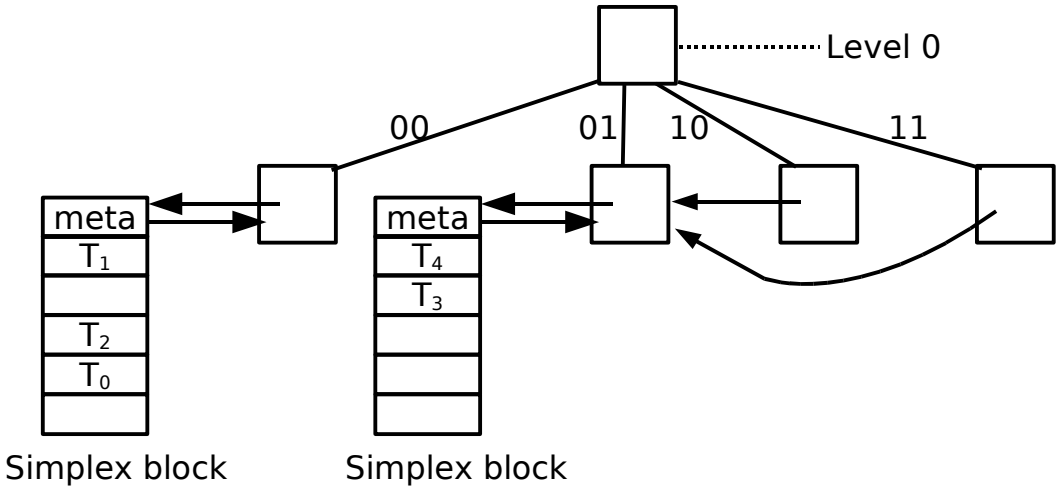


Figure 4.21: **Result of splitting a simplex block.**

After the split is completed, the insertion of the new triangle T_5 proceeds as normal. We search the octree from the root and traverse down to the leaf octant (01), which encloses the centroid of T_5 as shown in Figure 4.22. Then we allocate a simplex entry from the simplex block associated with the octant and initialize the incore triangle record for T_5. The result of the Abacus cache is illustrated Figure 4.23.

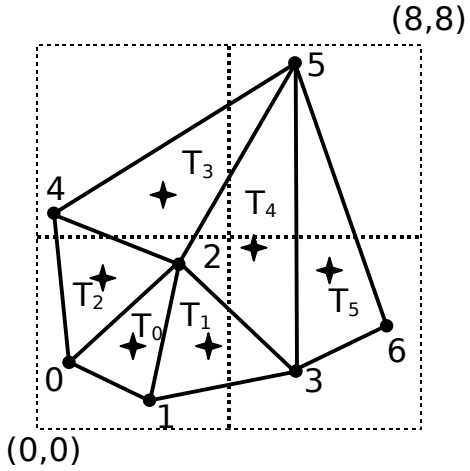


Figure 4.22: **Centroid distribution of all triangles (including T_5 among the children octants of the root.**

In cases when a new triangle's centroid falls inside an octant that is not the first sibling of the group, for example, a centroid falls inside octant (10), we follow the pointer to the first sibling (01) and find the simplex block indirectly.

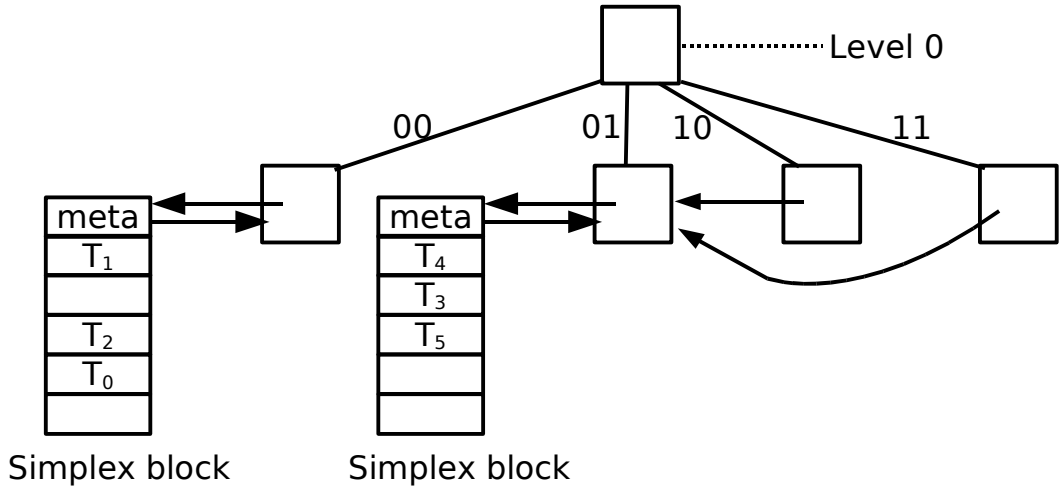


Figure 4.23: **Result of inserting T_5.**

A problem we have conveniently sidestepped is what if the distribution is extremely skewed. The centroid of the triangles in a full simplex block all fall inside one child octant. Figure 4.24 shows an example. Let us assume triangles T_0, T_1, T_2, T_3 and T_4 are already cached in a simplex block and we try to insert a new triangle, T_5. All the centroids, including the new triangle's, fall inside the first child (00) of the root. Using the one-to-two split algorithm just described, we get a *full* simplex block associated with the first group (00) and an *empty* simplex block associated with the second group (01, 10, 11). When we attempt to insert T_5 again, we run into the same full simplex block and have to split it again.

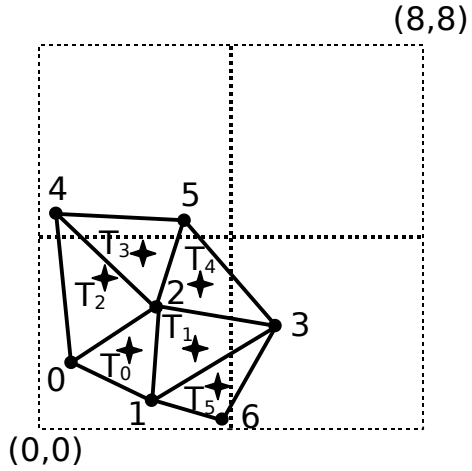


Figure 4.24: **All centroids fall inside one child octant.**

To avoid wasting simplex blocks and to ensure a split operation always effective (i.e., new simplex entries always become available after a split), we perform a *singular split* for the extreme cases as the one shown in the previous example.

In a singular split, a leaf octant is still split into 4 children octant. Except for the child that contains all the centroids, all the children are marked as *phantom octants* and are not associated with any simplex blocks, either directly or indirectly. [3] The phantom octant, however, do exist in the octree structure. The fully packed child is split recursively until a one-to-two split is performed, that is, until a new simplex block is finally allocated. Figure 4.25 shows the result of the singular split. In practice, when a vertex set is reasonably distributed in the space, singular splits occur rarely.

How to Evict Old Simplex Blocks?

After a sufficiently large number of splits, the free simplex blocks will be all used up. If we need to add more triangles into the Abacus cache, we must make room for the newcomers.

We implement a replacement policy by maintaining an LRU list as a doubly-linked list at the octree leaf level. Only leaf octants that have direct association with simplex blocks are linked in the list.

When a triangle is manipulated, we use its memory address to calculate which simplex block it belongs to. Using the pointer stored in the meta data field of the hosting simplex block, we trace back to the leaf octant and move it to the end of the LRU list. That is, the leaf octant becomes the most recently used.

When we use up all available simplex blocks and need a new one, simplex blocks associated with the leaf octants at the head of the LRU list are chosen for eviction. In the current implementation, we evict 10% of the least recently used simplex blocks at a time.

[3]The reason these octants are referred to as *phantom* is that there are no triangle data pages on disk associated with such octants.

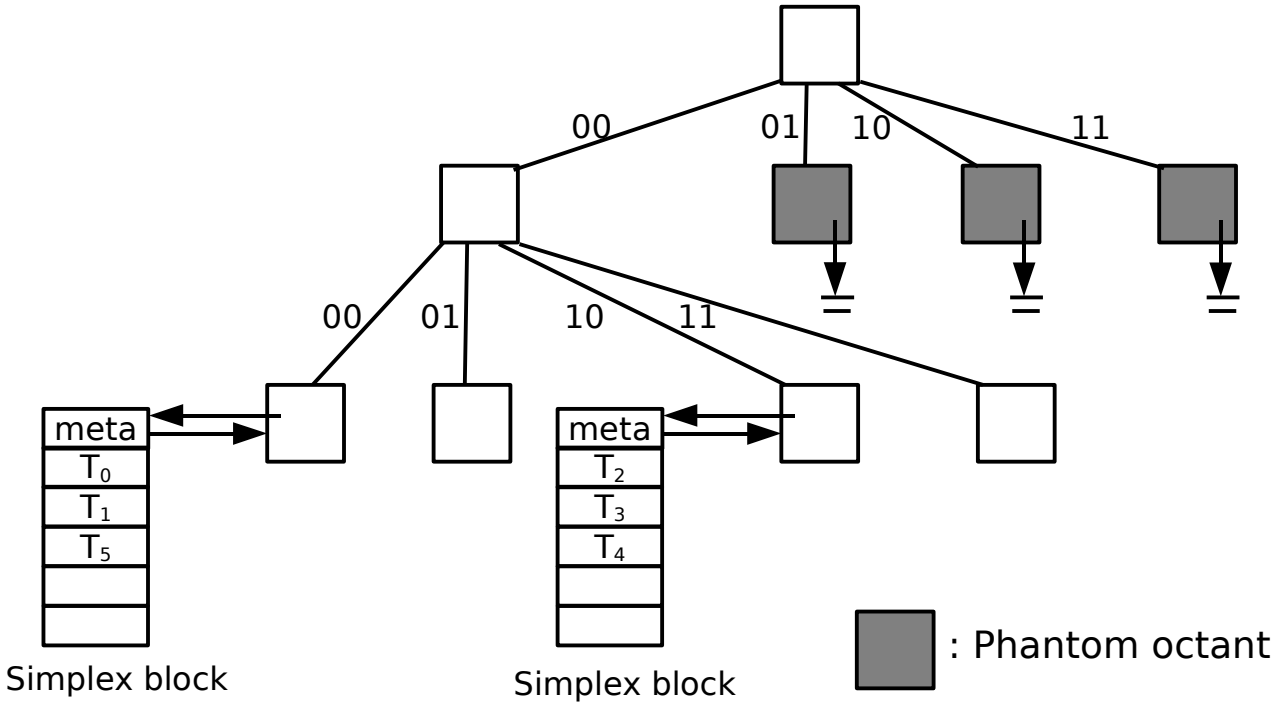


Figure 4.25: **The result of a singular split.**

This scheme works well thanks to the locality of references by applications. When a triangle is manipulated, a large number of its neighboring triangles in the same simplex block are also likely be manipulated. Hence, we can use the access pattern of a leaf octant to approximates the collective access pattern of the triangles stored in the simplex block associated with the leaf octant.

In summary, we make use of an octree structure to (1) choose which simplex block should be used to store a particular triangle, (2) perform one-to-two splits of full simplex blocks, and (3) evict simplex blocks that contain the least recently used triangles.

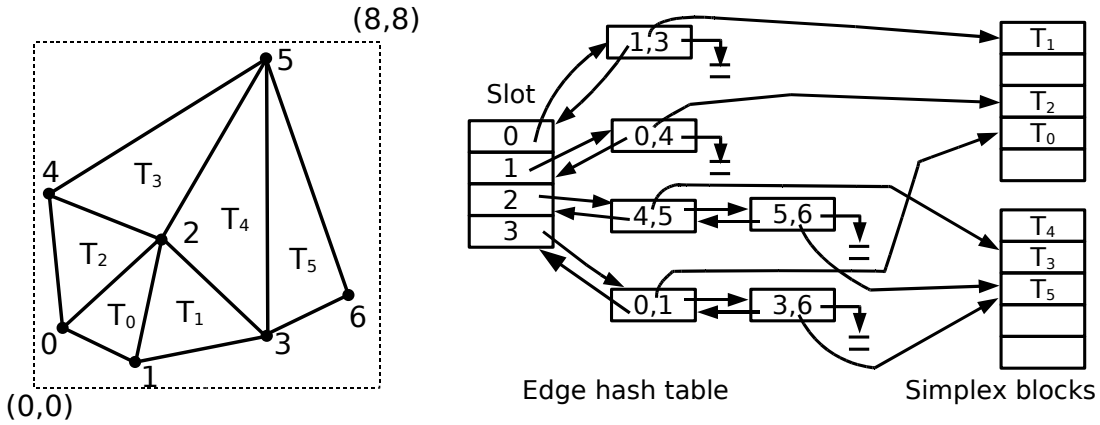
4.5.3 Edge Memory Management

The management of the edge hash table is quite straightforward. When an edge shows up for the first time, we inserted it into the edge hash table; when it shows up for the second time, we remove it from the edge hash table after we retrieve the information stored in the hash entry.

For example, when we try to insert a new triangle T_5 in the Abacus cache as shown in Figure 4.13, we search in the edge hash table for edges (3,5), (3,6), and (3,5), respectively. The status of the edge hash table is shown in Figure 4.11(b). The first two searches result in misses, which means that the edges are seen for the first time in the Abacus cache and we should install them into the edge hash table. The third search, the one for edge (3,5), results in a hit. The hash entry points to triangle T_4. The address is copied into the new incore triangle record for T_5 as a pointer to the neighbor that shares edge (3,5). We then follow the pointer to access the incore record for T_4 and modify one of its neighbor pointers to point to T_5. Thus, we are able to use the

edge hash table to establish correlation between a new triangle and existing triangles. Once the correlation is established, the hash entry for edge (3,5) is deleted from the hash table.

Figure 4.26(b) shows the result of the updated edge hash table after triangle T_5 (6,5,3) is inserted into the Abacus cache. Note that the original simplex block shown in Figure 4.11(b) has now been split into two blocks, as illustrated in Figure 4.23. Two new hash entries are created for edge (3,5) and (5,6), respectively. Both point to the new triangle T_5. The hash entry for edge (3,5) is deleted.



(a) Addition of T_5. (b) Result of the updated edge hash table.

Figure 4.26: **State of the edge hash table after T_5 is inserted into the Abacus cache.**

Note that all the edges of T_4 have become invisible. None of the edge hash entries points to T_4. This is because T_4 has become an *interior* cached triangle, surrounded by other cached triangles (T_1, T_3 and T_5). When we operate on a large chunk of spatially clustered triangles in the Abacus cache, most triangles are interior. Only those on the boundary contribute visible edge to the edge hash table.

By now, it should become clear that we do not need to keep track of *all* the edges of cached triangles, but only those that are at the interface between the Abacus triangle structure in memory and the triangulation database on disk.

4.6 Correlation of the Data Structures

The data structures we have presented so far are correlated in an organic way. Figure 4.27 illustrates the correlation visually. The vertex database, consisting of a vertex table and a vertex B-tree page index, maps to the vertex heap and vertex hash table inside the Abacus cache. The simplex database, consisting of a triangle table and a triangle B-tree page index, associates with the simplex blocks and the octree structure within the Abacus cache. The edge hash table inside the Abacus cache does not have a counterpart in the database. It serves as a glue that ties newly arrived database triangles with existing cached triangles. Inside the Abacus cache, the different data structures are also tightly correlated with one another, with the simplex blocks and the octree being the centerpiece.

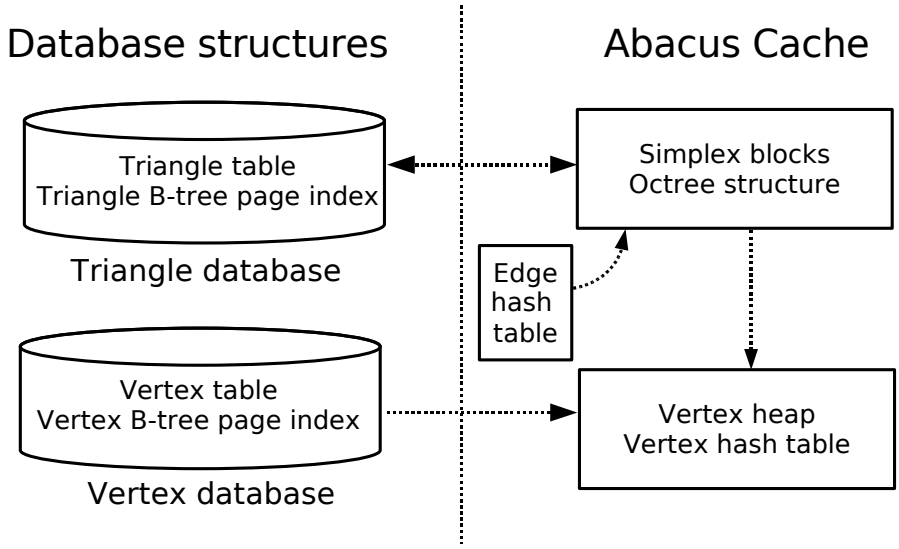


Figure 4.27: **Correlation of the data structures.**

Note the one-way arrow between the vertex database and the vertex heap and vertex hash table. In our current implementation, the vertex database is accessed read-only by the Abacus cache. The construction of the vertex table and the B-tree index is a separate process and does not make use of the Abacus cache. It usually involves staging the vertices in a certain way to optimize loading vertices into the Abacus cache. Section 4.12.2 explains how we stage a vertex set for constructing a Delaunay triangulation from scratch.

In the following sections, we describe (1) how to load vertices from the vertex database into the Abacus cache, (2) how to store triangles from the Abacus cache to the triangle database, and (3) how to load triangles from the triangle database into the Abacus cache.

For clarity, we add the term *database* before data structures that are associated with the databases. For example, *database vertex table*. Similarly, we add the term *Abacus* before data structures within the Abacus cache. For example, *Abacus vertex hash table*. Otherwise, confusion might arise since both are “tables”.

4.7 Loading Vertices into the Computational Cache

When we cannot find a vertex in the Abacus vertex hash table, a cache miss takes place and we must load the vertex into the Abacus cache. Instead of merely fetching the single missing vertex, our strategy is to prefetch all the vertices stored on the same vertex data page as the missing vertex.

We use the id of the missing vertex to search the database vertex B-tree page index. The result we get is the identifier (e.g., page number) of the database vertex data page that contains the missing vertex. We make a read request to the file system to fetch the page. If the page is already cached in the file buffer cache, it is returned immediately. Otherwise, a disk read operation is performed.

After we obtain a handle to the buffered page, we go through *all* the vertex records in the page one by one. For each record, we search the Abacus vertex hash table to check if it has been cached already. If so, we ignore the record. Otherwise, we allocate an entry on the Abacus vertex heap to cache the record and set its the reference count to 0. We also create a new Abacus vertex hash entry to keep track of the new incore vertex record.

The benefit of prefetching vertices is that we can amortize the cost of fetching a page, which include a B-tree search and, potentially, a disk read. The hope is that most of the vertices thus cached will soon be accessed.

However, some of the prefetched vertices might never be used. The reference counts of their incore vertex records will always *remain* to be 0. Therefore, these records will not be evicted *voluntarily* since their reference counts never *drops* to 0.

Our solution is to let them linger on the vertex heap. When all the heap memory is used up, we conduct a sweep and reclaim memory occupied by entries whose reference counts are 0 (See Section 4.5.1).

4.8 Linear Octree and Proximity Search

The key bridge between the triangle database and the Abacus triangle structure is a technique called *linear octree* [31, 71, 72]. Therefore, before describing how to exchange triangles between the Abacus cache and the database, we highlight the features of linear octrees and explain how to make use of them to conduct proximity searches in a triangle database.

The basic idea of the linear octree is to encode each leaf octant by a key called the *locational code* that uniquely identifies the octant. There are two different types of locational codes: variable-length and fixed-length. Conceptually, both schemes derive the locational code for an octant by concatenating the directional codes on the path from the root to some target octant. We refer to these concatenated bits as the *path information*. The difference between the two schemes lies in the number of bits used to encode the path information.

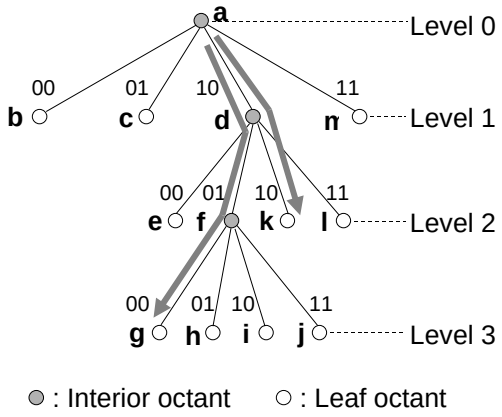
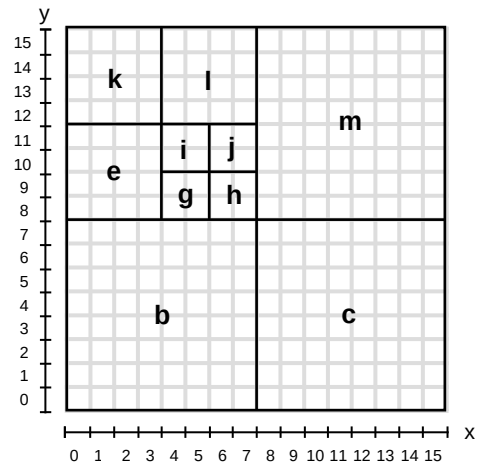
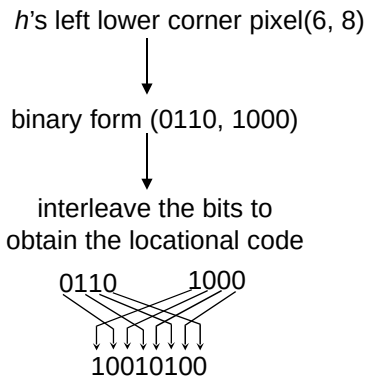


Figure 4.28: **Variable-length locational codes.** Concatenate the directional codes on the path from the root to the target octant.



(a) Domain representation.



(b) Computing the locational code for h.

Figure 4.29: **Computing a locational code by interleaving the bits of the coordinate.**

In the variable-length scheme, leaf octants at different levels use a different number of bits to encode their path information. For example, in Figure 4.28, the path information for octant k is 1010, while path information for octant g is 100100. Since the path information can uniquely identify an octant, the variable-length scheme actually uses the path information directly as the locational code for an octant.

In contrast, the fixed-length scheme uses the same number of bits to encode the path information for all leaves. Obviously, the number of bits should be sufficient to encode the leaf octant at the maximum allowable level. For octants at higher levels, zeroes are padded to extend the path information to the specified length. For example, the maximum allowable level of the octree in Figure 4.15 is four. Thus the fixed-length path information for octant k is 10100000, with the trailing four zeroes padded to its variable-length counterpart. Similarly, octant g has a fixed-length path information of 10010000.

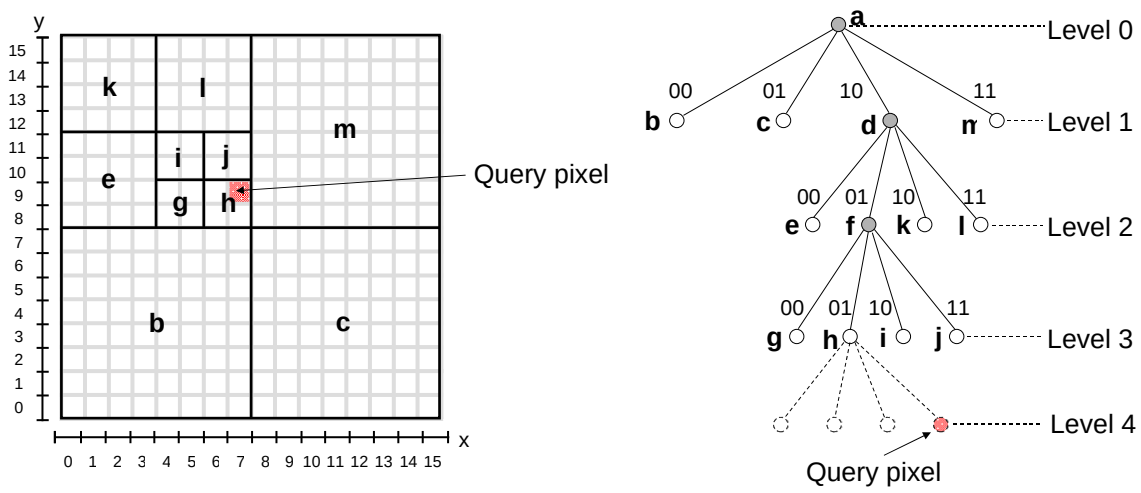
We adopt the fixed-length scheme because the locational codes thus constructed can be treated as fixed length byte strings, which is a requirement for indexing by a B-tree. For brevity, we will use the term *locational code* to refer to the fixed-length locational code.

4.8.1 Computing Locational Codes

The concatenation method just described is a conceptual model. In practice, we compute the locational codes by interleaving the bits of the coordinates of the left-lower corner pixel of an octant [93]. Figure 4.8.2(b) illustrates how to compute the locational code for octant h. (Figure 4.8.2(b) is a replication of Figure 4.15(a) for convenience of reference.)

4.8.2 Locating An Octant Without Knowing Its Locational Code

A useful property of the locational code is that the ordering of leaf octants based on their locational code is exactly the preorder traversal of the octree [93].



(a) Query pixel enclosed by leaf octant h (b) Hypothetical expansion of leaf octant h

Figure 4.30: **Finding a leaf octant without knowing its exact locational code.**

We make use of this property to locate an octant using the locational code of any pixels that fall inside the octant. For example, suppose we have a query pixel that is enclosed by octant h as shown in Figure 4.30(a). Let us hypothetically expand the enclosing leaf octant h to the maximum allowable level of the octree (which is 4 in the $2^4 \times 2^4$ domain) as shown in Figure 4.30(b). A preorder traversal guarantees that (1) The subtree root be visited before any descendants, and (2) The traversal of a subtree be completed before processing the next subtree.

Generally, if two leaf octants, say h and i, are next to each other in the ascending locational code order, the locational code of *any pixel within* h must also be strictly less than that of *any pixel within* i. This is because all the pixels within h belong to the subtree rooted at h and all the pixels within i belong to the subtree rooted at i. In a preorder traversal, all descendants of h come before those of i, given the assumption that h comes before i.

We can exploit this property to find an octant without knowing its *exact* locational code. It works as follows. We build a B-tree that indexes the locational codes of octants. Given an arbitrary query pixel in the domain, we can compute its locational code by interleaving its integer coordinates, as explained in the Figure . We use the locational code as a search key to query the B-tree. Instead of returning an exact match, we modify the search routine slightly and return the key that is the maximum among all the index keys that are less than the search key. The key returned is the locational code of the octant that encloses query pixel. In other words, we can use a B-tree to find an octant by specifying the locational codes of *any* pixels that fall inside the octant.

We make use of this capability to conduct proximity search into the triangle database.

4.8.3 Proximity Search

Now we explain how to compute the keys for the triangle B-tree index, an issue that we have deferred in Section 4.3.2.

Since the simplex block managed by a leaf octant maps to a triangle data page, we use the locational code of the leaf octant as a key to index the triangle data page in the triangle B-tree.

If a simplex block is associated with only one octant, for example leaf octant (00) in Figure 4.21, we use the locational code of the octant directly. If a simplex block is associated with a group of sibling octants, for example, leaf octants (01, 10, 11) in Figure 4.21, we use the locational code of the first sibling (01) as the key.

Figure 4.31 shows the distribution of a number of centroids within four octants. The circles, diamonds, four-stared points, and triangular tick marks are the centroids of the triangles. The triangles themselves are not shown explicitly. The centroids should be thought of as pixels within the octants. Because the locational code of an octant is computed by interleaving the pixel at its left lower corner, we mark them explicitly as `first_pixel_i` in the figure.

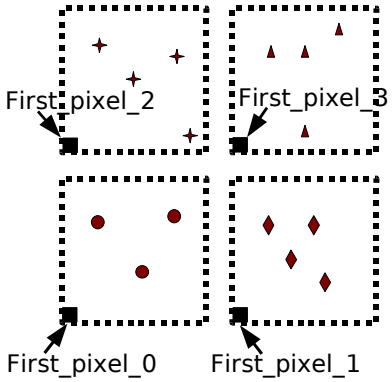


Figure 4.31: **Centroids of triangles scattered in 4 octants.** The black square box in the left lower corner of each octant represents the first pixel with that octant.

The result of indexing the locational codes of the octants (i.e., the `first_pixel_i`) is shown in Figure 4.32. It can be seen that triangles spatially close to one another (within the same octant) are placed in the same triangle data page. When we project the locational codes of the their centroids onto to the search key space (shown as an axis at the bottom of the figure), they are strictly in between two `first_pixels`.

Now consider an arbitrary query point within the four octants. Using the modified B-tree search technique described in the previous subsection, we find an index entry whose corresponding octant encloses the query point. The `page_id` field of the index entry points to a triangle data page that contains triangles whose centroids are enclosed by the same octant. The triangles we find are spatially close to the query point.

The beauty of this scheme is that we are able to conduct proximity search without the knowledge of how the triangles are distributed in space. We can query the domain using any coordinate. The triangle B-tree index will guide us to the region (a triangle data page) where we can find nearby triangles.

In summary, by using the locational codes to index triangle data pages, we are able to use a scalar B-tree to search the vector space of a triangulation that is stored in a database.

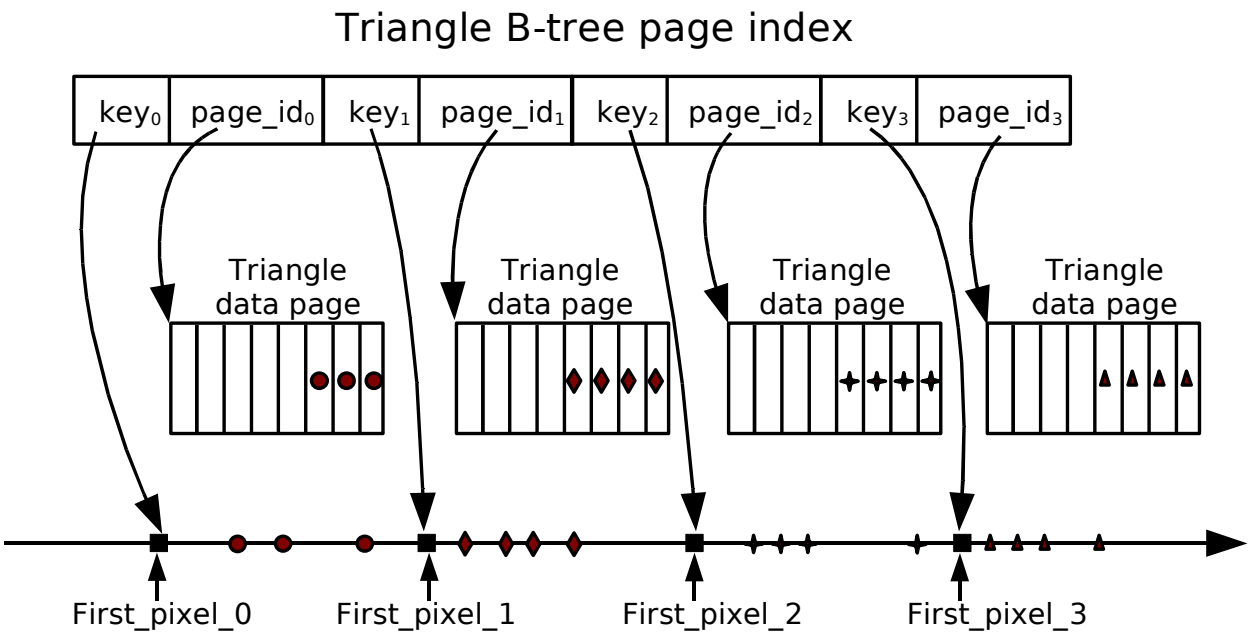


Figure 4.32: **The partition of the locational code space.**

4.9 Storing Triangles to the Database

As described in Section 4.5.2, we use a doubly-linked LRU list at the leaf level of the octree to keep track of the simplex block usage. When we run out free simplex blocks and need to allocate a new, we evict the simplex blocks associated with the leaf octants that are at the head of the LRU list. If a simplex block to be evicted has never been modified, that is, no triangles have been inserted into or deleted from the simplex block, we discard the content in simplex block and mark it as free immediately. Otherwise, we have to store the triangles in the simplex block to a database triangle data page. (Recall that each simplex block maps to a triangle data page.)

Mapping a (modified) simplex block to a triangle data page is a three-step procedure:

1. Obtaining a new triangle data page
2. Converting the triangles from the Abacus internal representation to the database format
3. Creating an index entry in the database triangle B-tree to keep track of the data page

The first step is straightforward in our current implementation. We use a linked list to keep track of triangle data pages that are freed (deallocated) and allocate new pages from the the list if there are pages available. Otherwise, we allocate a new data page at the end of the triangle table file.

The second step goes through the entries of the simplex block. For each entry (i.e., an incore triangle record with 6 pointers), we create a new item on the target slotted data page and then delete the entry. Various Abacus data structures are updated accordingly.

The last step adds an index entry into the database triangle B-tree page index to keep track of the new data page. The key we use to index a triangle data page is the locational code of the

octant that is associated with the simplex block that is being swapped out. If multiple octants are associated with the simplex block, the locational code of the first sibling is used.

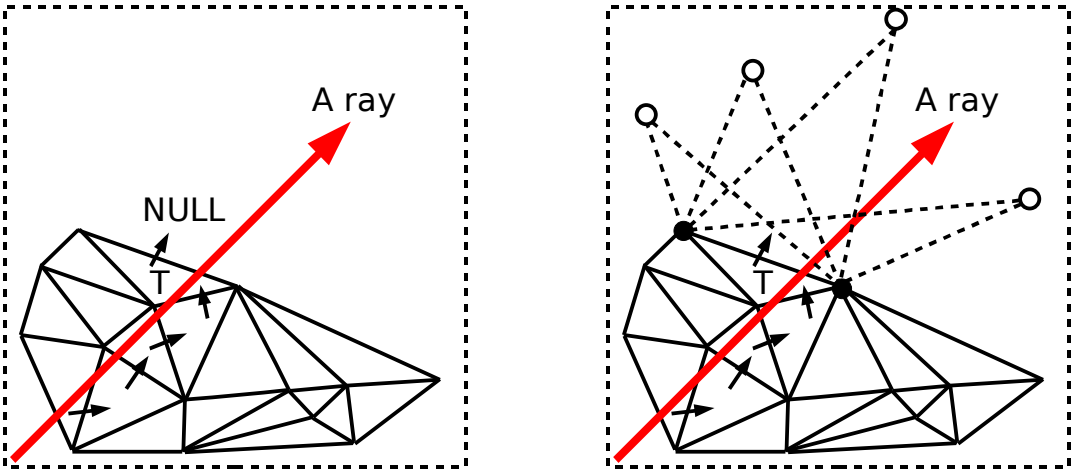
4.10 Loading Triangles into the Computational Cache

The inverse operation of storing triangles to the database is loading triangles from the database into the Abacus cache.

If we know *all* the vertex ids of a triangle, we retrieve the coordinates and compute the triangle's centroid. Then we can search the database triangle B-tree using the locational code of the centroid (as a pixel). The query result is an index entry pointing to a triangle data page that contains the target triangle we need.

However, in most cases we do not know *all* the vertex ids of a triangle and cannot use the simple procedure just described. In a typical application, for example, casting a ray through a triangulation, the basic operation is to move from a triangle to its neighbor across an edge. Figure 4.33(a) shows the part of a triangulation that has already cached in Abacus. The short arrows represent direct moves from one triangle to the next. When we reach triangle `T`, we find the pointer to the neighbor in the direction of the ray is NULL since the neighbor is not cached yet.

At this moment, we only know the vertex ids of the edge shared by the missing neighbor and `T`, marked as two filled circles in the in Figure 4.33(b). The third, unknown vertex, marked as an empty circle in the figure, could be anywhere in the domain. We cannot pinpoint where the centroid of the target triangle is and thus are unable to search for using the triangle B-tree.



(a) Moving within the Abacus triangle structure. (b) Unknown third vertex of an uncached triangle.

Figure 4.33: **Moving to adjacent triangles across edges.** When a triangle is not cache, we are unable to carry on the operation.

Taking advantage of the proximity search capability, we locate the missing triangle by *probing* the triangle B-tree. In order to reduce the number of probes and quickly find the data page

that encloses the missing triangle, we choose probing pixels wisely so that they are likely to lead us to a page hit.

Recognizing the spatial clustering property of a triangulation, we probe the triangle B-tree page index using three different methods. The first two methods, *quick probing* and *ray-casting probing* are speculative. They are more efficient but do not guarantee the target triangle could be found. The third method, *concentric probing*, works more conservatively and has higher overhead. But it ensures that we are able to finally find the triangle we need.

4.10.1 Quick Probing

The intuition of quick probing is that the middle point of an edge of a triangle *might* be spatially close to the centroid of the triangle. Hence, there is a good chance we could find the octant that encloses the centroid if we search the B-tree page index using the middle point pixel of the edge.

Operationally, it works as follows. We calculate the coordinate of the middle point of the edge that we have failed to cross, convert the coordinate to an integer pixel, and compute the locational code of the pixel.

We search the Abacus octree structure of check if any existing leaf octant encloses the query pixel. If so, we abort the quick probing operation because the triangle data page corresponding to the enclosing leaf octant already exists in the Abacus cache and it does not contain the triangle we seek. If not, we send a request to the triangle B-tree index and fetch the triangle data page pointed at by the hit index entry.

Similar to prefetching vertices, we prefetch all the triangle records stored in a triangle data page and install them into the Abacus cache. After the loading is completed, we check whether the pointer to the target triangle has been modified from NULL to a valid address. If so, the quick probing has been successful. Otherwise, we have to use the following two methods to continue probing.

4.10.2 Ray-Casting Probing

The limitation of the quick probing method is that we use only one probing pixel. In order to create more probing points, we borrow the idea of ray casting from computer graphics and visualization.

The basic idea of ray-casting probing is to cast a number of rays from the middle point of the edge that we have failed to cross. The directions of these rays are towards the opposite side of where we come from as shown in Figure 4.34. Each ray intersects a series of octants that are present either in the Abacus octree structure or in the triangle B-tree. We ignore those that already exist in the Abacus cache and fetch the ones that are not. It should be emphasized that we do not know about the location or the size of an uncached octant, nor do we know the distribution of the octants in space. We learn the information as we carry out the probing.

In order to conduct ray-casting probing, we have to solve three problems:

- Given a ray, how to compute a probing pixel for an unknown octant?
- How to define the set of rays to cast?
- How to control the order of ray-casting probing?

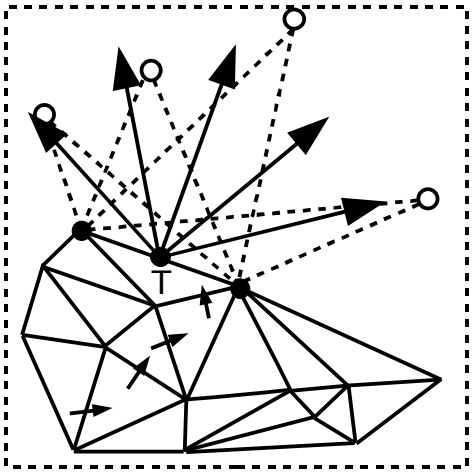


Figure 4.34: **Rays originated from the middle point of an edge.**

Ray-Box Intersection

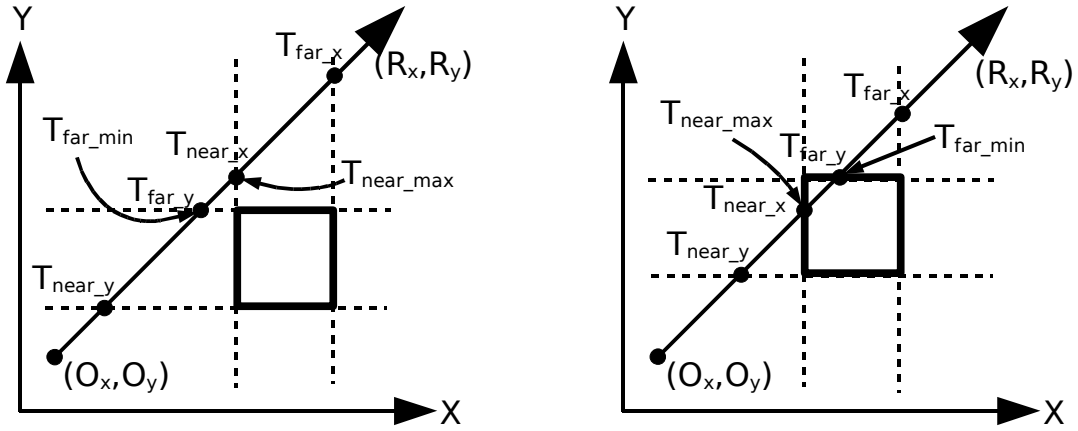
The main algorithm we use to compute probing pixels is based on a technique known as *ray-box intersection*, which solves the problem of whether a ray, specified by its origin and direction, intersects an axis-aligned rectangular box, and if so, what are the distances from the origin of the ray to the entry point and the exit point, respectively.

We use the particular method [48] developed by by Kay and Kayjia, which operates on "slabs". A slab is the space between two parallel planes. So the intersection of a set of slabs defines a box (an octant). The method looks at the intersection of each pair of slabs by the ray. It finds two values, T_{far_i} and T_{near_i} for each pair of slabs along axis i. If T_{near_max}, the largest T_{near} value among all axes is greater than T_{far_min}, the smallest T_{far} value among all axes, then the ray misses the box. Otherwise, it intersects the box. T_{near_max} and T_{far_min} are the distances from the origin of the ray to the entry point and the exit point, respectively. Figure 4.35(a) shows an example where a ray misses a box; (b) shows an example where a ray intersects a box. Note that in order to apply the slab method, we must know the size of the box (i.e., the widths of the slabs).

Equipped with the slab method, we compute the probing pixels along a ray iteratively. The origin is the middle point of the edge on which we have got stuck. We use the current cached octant (with which the ray intersects) to compute the probing pixel for the next octant. At the beginning, the octant that encloses the origin (i.e., edge middle point) must be present in the Abacus cache thanks to the quick probing step. Hence, the following procedure is well-founded.

For each cached octant, we compute the T_{far_min} value, which is the *distance* from the origin of the ray to the exit point on the octant. Next, we compute the *coordinate* of the exit point. Then, we shift the results towards the direction of the ray and convert them into integer values (assuming the floating-point numbers do not overflow the range of the integer representation). The nudged integer coordinate gives us a probing pixel to locate the next uncached octant. The various concept is illustrated in Figure 4.36.

After we find the next octant from the triangle B-tree index, we initialize the octant in the Abacus octree structure and assign a simplex block to the new octant. The triangle data page is



(a) A ray misses a box ($T_{near\_max} > T_{far\_min}$). (b) A ray intersects a box ($T_{near\_max} < T_{far\_min}$).

Figure 4.35: **Ray-box intersection.** (O_x, O_y) is the origin of the ray; (R_x, R_y) is the direction of the ray.

fetched and the records are converted and stored into the Abacus cache. If we find the missing triangle, then the probing is successful. Otherwise, we use the newly cached octant to probe for the next one.

Directions of Probing Rays

The next technical problem to solve is how to define a set of rays. From the middle point of the edge where we have failed to reach the adjacent neighbor, there are infinite number of directions to cast a ray. Which ones should we use? Without the knowledge of how the octants are distributed in space, there exists no optimal answer. We use a simple heuristic and cast a preset number of rays uniformly in all direction (in the halfspace opposite to where we come from). In the current implementation, the preset numbers are 9 for 2D and 17 for 3D.

The base ray direction is perpendicular to the edge we are crossing as shown in Figure 4.34. Since we know the coordinates of the end points of the edge, we can easily compute the normal of the edge V_{normal}. The other directions are computed by rotating the normal vector as follows.

$$V_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} V_{normal} \tag{4.3}$$

where V_θ is a ray that is θ aways from V_{normal} in the counterclockwise direction.

Order of Ray-Casting Probing

Our strategy for ray-casting probing is not to follow a single ray until it travels outside of the domain. Instead, we follow all the rays one step at a time. That is, after we retrieve an octant along a certain ray and do not find the triangle we need, we move to the next ray to probe forward for one step. Thus, we are able to probe nearby regions extensively before moving far away from the edge where we started the probing. In the current implementation, we bound the number of steps for speculative ray-casting probes to 32 for 2D and 4 for 3D.

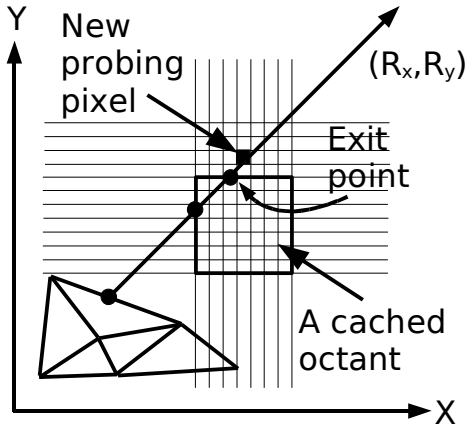


Figure 4.36: **Ray-casting probing.** An cached octant is used to compute the probing pixel for the next octant.

4.10.3 Concentric Probing

The ray-casting probing method is designed to exploit the locality of the triangulation structure. The rationale is that since the missing triangle shares an edge, its third vertex is *probably* not far away. By probing in the nearby region, there is a good chance that we can find it. However, there is no guarantee of finding the missing triangle unless we cast infinite number of rays.

To make sure that we can *always* find the missing triangle, we use a more conservative strategy called *concentric probing* to search the domain exhaustively.

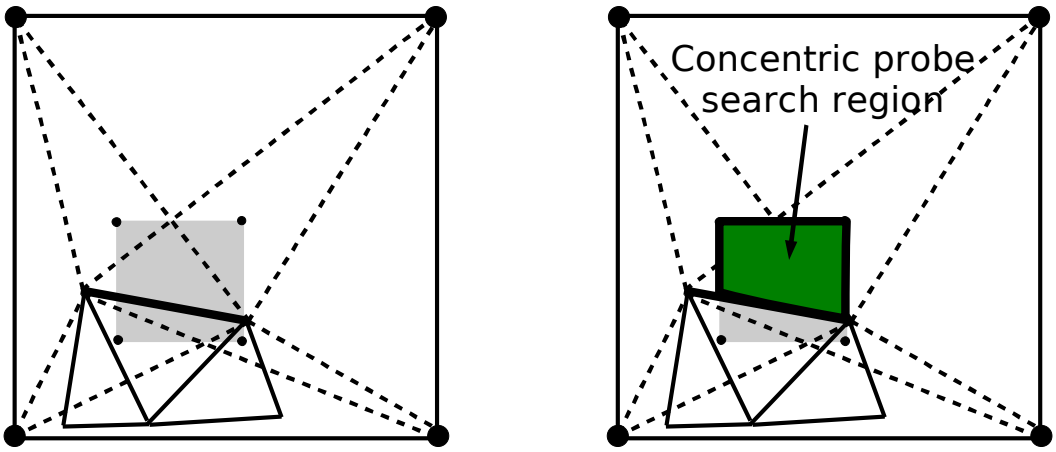
Probing Region

Probing the whole domain is expensive. In fact, it is unnecessary. We can limit the probing region effectively. Although we do not know where the third *vertex* of the missing triangle is, we can still accurately bound the region where its *centroid* might fall.

Recall that we can always know the domain specification (i.e., the origin and dimensions) in advance. The four corners of the domain bound where the third unknown vertex could fall. In the worse case, the unknown vertex falls on the domain boundary. If we connect the edge of interest to the four corners of the domain, we obtain four large (and probably skinny) triangles. When we connect the centroids of these four hypothetical triangles, we obtain the rectangular region where the centroid of *any* triangle that shares the edge must fall as shown in Figure 4.37(a). Because the centroid of the uncached neighbor triangle must be on the other side of the edge, we can further limit the search area to one side of the edge, as shown in Figure 4.37(b).

Order of Concentric Probing

Unlike ray-casting probing where we adapt to the size of the octants as we probe, we use a fixed-size grid overlay to conduct a concentric probing. The grid size is equal to the size of the smallest octant currently in the domain. It is not necessarily the size of a pixel, which is the smallest *possible* octant.



(a) Possible search region. (b) Reduced search region.

Figure 4.37: **Concentric probing search region.**

A concentric probing is not carried out within the search grid in an axis-aligned order. Instead, as the name suggests, we start the probing from the grid that encloses the middle point of the edge that we have failed to cross and expand the search radius gradually as shown in Figure 4.38.

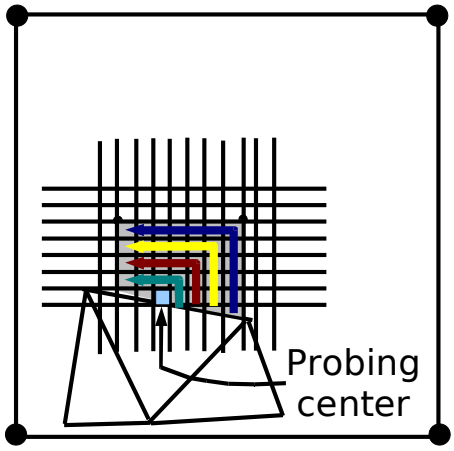


Figure 4.38: **Concentric probing order.** The thick arrows represent rounds of concentric probing, starting from the innermost circle and expanding outwards.

It must be emphasized that when we probe for a grid, we first search the Abacus octree structure to check if the enclosing octant is already cached. Useless requests are eliminated efficiently within the Abacus cache. Only when we cannot find an enclosing leaf octant is a search question sent to the database triangle B-tree.

4.11 Abacus API

The techniques described in the previous sections are used inside the Abacus system and are not visible to the outside world. A small API is exported to allow application programmers to interact with Abacus. This section presents the generic API functions for manipulating *any* triangulation datasets. The next section explains two specialized API functions that are implemented specifically to support Delaunay triangulation.

4.11.1 Opening and Closing An Abacus Database

```
#include <abacus.h>

abacus_t *abacus_open(const char *path, int flags, int dimension,
int payloadsize, double *near, double *far, int cachesize);
```

Returns: a handle to an open abacus database, NULL on error

The `abacus_open` function opens an existing Abacus database or create a new Abacus database depending on the options specified in the *flags* parameters. Internally, this function allocates memory for the vertex heap, the vertex hash table, the simplex blocks, the octree structure, and the edge hash table, as explained in Section 4.4. It also embeds the domain to a square region (see Section 4.2).

This function receives the following parameters:

- `path`: The path name of the Abacus database. Within an Abacus database, the file names of the vertex table, the triangle table, the vertex B-tree, and the triangle B-tree are assigned internally.
- `flags`: This parameter specifies the mode in which the file is to be opened. Flags can have one of the following values: `O_RDONLY` or `O_RDWR`. The flags may also be bitwise-or'd with `O_CREAT` or `O_TRUNC`. The semantics are the same as in UNIX.
- `dimension`: Specifies the dimension of the triangulation datasets. The current implementation supports 2D and 3D.
- `payloadsize`: The size of the payload of each triangle. This parameter is only used to created a new Abacus database (i.e., O_CREAT, O_TRUNC was specified), otherwise the payload size is obtained from an existing Abacus.
- `near`: This parameter is a 2-element array for 2D cases and 3-element array for 3D cases. It contains the coordinate of the left-lower corner of the problem domain.
- `far`: Similar to `near` in format, this parameter contains the right-upper corner of the problem domain.
- `cachesize`: Specifies the maximum amount of memory that can be used by the Abacus computational cache and all other internal data structures. In the current implementation, a heuristic is used to estimate the memory usage of different structures and the memory is allocated statically at initialization.

On success, this function returns an abacus handle (`abacus_t struct`) which is used in subsequent calls to the operate the Abacus database. If an error occurs, for example, running out of memory, the Abacus database is not opened and a NULL pointer is returned.

```
#include <abacus.h>

int_t abacus_close(abacus_t *ap);
```
Returns: 0 on success, -1 on error

The `abacus_close` function closes an Abacus database by completing the following tasks: (1) storing all the cached triangles that have been modified (or created) to the triangle database, (2) releasing the memory used by the Abacus cache, and (3) closing the database files.

This function receives the following parameter:

- `ap`: Handle to the Abacus database to be closed.

This function returns 0 on success, -1 on error. If an error occurs the application program should call `perror` to identify the nature of the problem.

4.11.2 Inserting and Deleting Simplices

```
#include <abacus.h>

int_t abacus_insert(abacus_t *ap, const simplex_t *simplex);
```
Returns: 0 on success, -1 on error

The `abacus_insert` function inserts a new simplex record (i.e., a triangle or a tetrahedron) in an Abacus database. If we have a pre-generated triangulation, we can use this function to load the data into an Abacus database.

This function receives the following parameters:

- `ap`: Handle to the Abacus database to which the simplex is to be inserted.
- `simplex`: This parameter is a generic wrapper, which is dereferenced internally by the Abacus runtime as either a `triangle_t` or a `tetrahedron_t` structure according to the dimension specified when the Abacus database is opened.

This function returns 0 on success and -1 on error. On a successful return, the inserted simplex is stored in the Abacus cache. Note that the records is not mapped to a triangle data page immediately. There is no persistence guarantee. If the system crashes immediately after the function call returns, the record may be lost.

The implementation of the function is straightforward. We fetch the coordinates of the vertices of the new triangle, either from the Abacus vertex heap if they have been cached or from the vertex database if they have not. Using the coordinates, we compute the centroid of the triangle and search the Abacus octree structure to find the enclosing leaf octant. If the leaf octant is a phantom octant (see Section 4.5.2), we request a new simplex block and assign it to the phantom octant. We then allocate an incore triangle record from the simplex block. If the simplex

block is full, we split it using the algorithms described in Section 4.5.2. The new incore triangle record is initialized properly to point to its neighbors and its vertices. The reference counts of its vertices stored in the Abacus vertex heap are incremented by 1. The edge hash table is updated accordingly to either expose new visible edges or remove shared interior edges.

In the current implementation, no data integrity check is enforced. If two identical simplices are inserted, no error occurs. But the second insertion becomes a dangling simplex that is not attached to the Abacus triangle structure. Worse, if two cross-over triangles are inserted, no immediate error is reported but further operations may fail.

For Abacus to be fully functional, we need to a implement semantic integrity check in the future.

```
#include <abacus.h>

int_t abacus_delete(abacus_t *ap, const simplex_t *simplex);
```

Returns: 0 on success, -1 on error

The `abacus_delete` function deletes a simplex record from an Abacus database. This function is provided for completeness of the API. [4]

This function receives the following parameters:

- `ap`: Handle to the Abacus database from which the simplex is to be deleted.
- `simplex`: Specifies the simplex to be deleted.

This function returns 0 on success and -1 on error. An error indicates that the target simplex does not exist in the Abacus database.

The implementation of the function is as follows. We use the centroid of the triangle to locate the leaf octant that encloses it. If we cannot find a leaf octant, we use the centroid as a probing pixel to search the database triangle B-tree index, fetch the triangle data page, and load the triangle records into a simplex block. If we do find a leaf octant, we retrieve the associated simplex block directly. Either way, we search the simplex block for the triangle to be deleted. If we cannot find it, return an error. If we find it, the entry in the simplex block is released. Relevant pointers in the neighboring triangles are modified to point to NULL since the triangle is being deleted. The reference counts of the vertices of the deleted triangle are decremented by 1, respectively. The edge hash table is updated accordingly to either expose new visible edges or remove edges that are no longer used by any triangles in the Abacus cache.

If the simplex block from which the deleted triangle is released is associated with a valid triangle data page id, it means the simplex block has never been modified. In this case, we carry out three operations: (1) delete the corresponding index entry in the triangle B-tree, (2) deallocate the triangle data page, and (3) mark the current simplex block as not associated with any triangle data page.

The aggressive deletion strategy is motivated by the lessons we learned in Chapter 2. Recall that the incremental insertion algorithm for constructing Delaunay triangulation deletes 66% of all the triangles that are ever generated. Therefore, the deletion of one triangle on a triangle data page often heralds more deletions on the same page. By deallocating a modified triangle data

[4] I do not know of an application that needs to delete individual triangles.

page at the first possible instant, we avoid the overhead of deleting individual triangles from the slotted data page. The Abacus cache will absorbs all the follow-up deletions, much in the same way as a processor cache absorbs repeated writes into the same cache line.

4.11.3 Searching for Simplices

Two API functions are provided to support querying triangles. They can be combined to implement more complicated queries such as general range queries, iso-contour extraction, and ray-casting volume rendering.

```
#include <abacus.h>

int_t abacus_reset(abacus_t *ap, double *location, simplex_t
*result);
```

Returns: 0 on success, -1 on error

The `abacus_reset` function searches an Abacus database and returns the simplex that encloses a query point.

This function receives the following parameters:

- `ap`: Handle to the Abacus database to be searched.
- `location`: Specifies either a 2D or 3D query point.
- `result`: An output parameter that points to either a `triangle_t` or a `tetrahedron_t` structure.

This function returns 0 on success and -1 on error. On success, the simplex enclosing the query point is stored in the structure pointed by `result`.

We implement this function using a combination of searching in an octree and stochastic walking in triangulation. Figure 4.39 shows a query point marked as a circle in our example triangulation.

Assuming all the triangles are already cache, we find the enclosing triangle T_0 in three steps, as shown in Figure 4.40. First, we start from the root of the Abacus octree and descend to the leaf octant (00), which encloses the query point. Second, we access the simplex block associated with the leaf octant. Third, starting from triangle T_1, we "walk" to T_0 across the edge (1,2). We will explain the "walk" operation shortly.

For a large triangulation, the *leaf octant* that encloses the query point might not be present in the Abacus octree structure, which means that the triangle data page covering that area is not cached within Abacus yet. In this case, we compute a search key using the coordinate of the query point and fetch the triangle page by querying the triangle B-tree.

Another subtle issue is illustrated in Figure 4.41. The query point might fall in an octant (10) that does not contains the centroid of the target triangle, that is, the target triangle is not allocated from the simplex block associated with the octant. In this case, we still pick an arbitrary triangle (e.g., T_4) in the simplex block associated with the "wrong" octant and start a "walk" from there to reach the target triangle T_0 across edge (2,3), as shown in Figure 4.42.

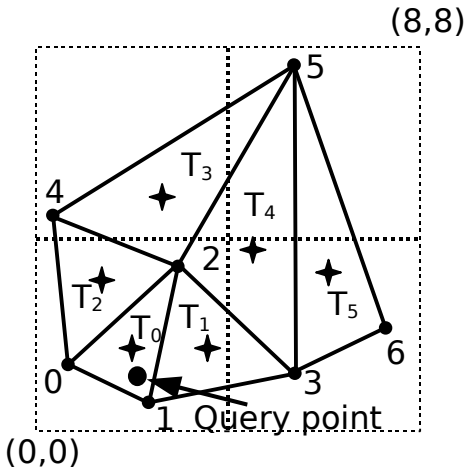


Figure 4.39: **A query point in a triangulation.** The circle represents the query point. The four-pointed stars represent the centroids of the triangles.

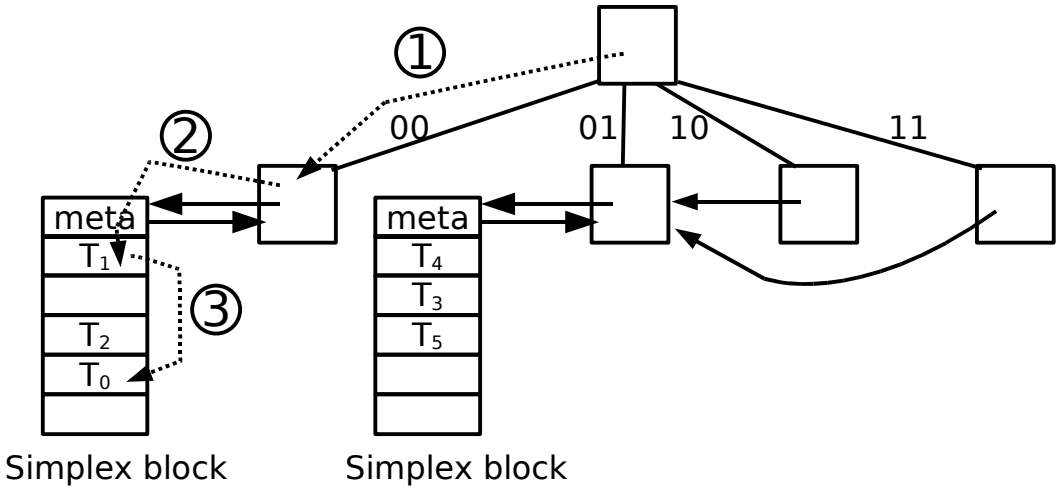


Figure 4.40: **Point location within the Abacus cache.**

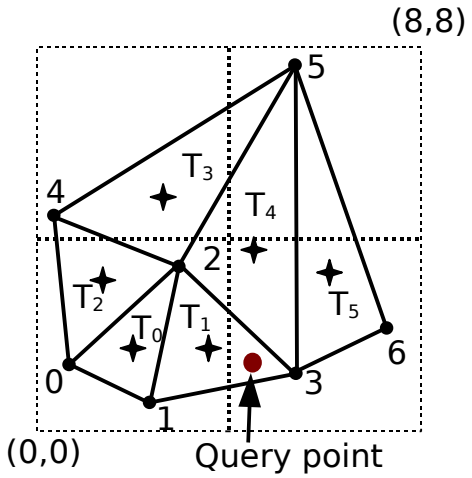


Figure 4.41: **The query point falls inside an octant that does not enclose the centroid of the target triangle.**

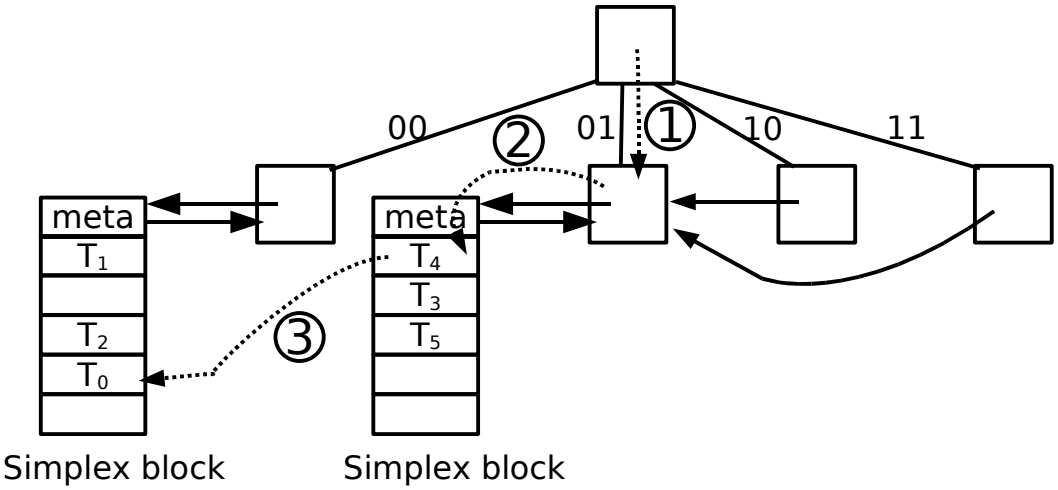


Figure 4.42: **Point location within the Abacus cache by walking across simplex blocks.**

Stochastic Walk

The "walk" operation we have implemented is based an algorithm known as *stochastic walk* [25], which is a variant of a simpler algorithm called the *visibility walk*.

A visibility walk starts from a triangle and walks towards a query point `p`. For each triangle `T` visited, the first edge (defined in certain way) e_0 is tested to check whether e_0 separates `T` from `p`. This is carried out by a robust orientation test. If so, the next visited triangle is the neighbor of `T` through e_0. Otherwise, the second edge is tested in the same way. In case the test for the second edge also fails, then the third edge is tested. The failure of the third test indicates that the target triangle has been reached. That is, `T` encloses `p`.

Figure 4.43 shows an example. Of the three edges of `T`, e_1 and e_2 separate `T` from the query point, while e_0 does not. According to the order of visiting the edges, we walk to triangle `S` across e_1 and continue the visibility walk from there. For clarity, other triangles are not shown in the figure.

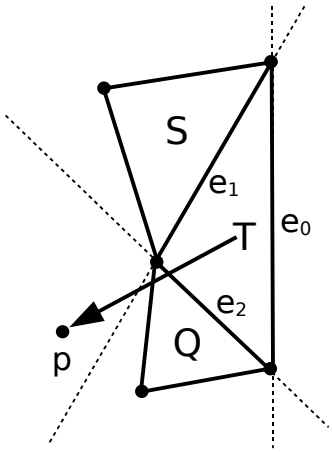


Figure 4.43: **Visibility walk.**

The visibility walk in a Delaunay triangulation always terminates [29]. However, for non Delaunay triangulations, the visibility walk may fall into a cycle. To avoid this problem, we use the stochastic walk algorithm. The only difference between a stochastic walk and a visibility walk is how the *first* edge of a triangle is chosen. Instead of using a prescribed (fixed) ordering of the edges of a triangle as is the case with a visibility walk, a stochastic walk choose the first edge randomly at runtime. In particular, when a triangle is visited for a second time (due to a loop), the first edge chosen *could* be different from the first edge chosen for the first time. Such a randomness ensures that even if a walk enters into a cycle within a triangulation, it cannot loop in the cycle forever. For a detailed proof, see the paper by Devillers and colleagues [25].

A variant of the stochastic walk algorithm called *Directed Local Search* (DLS) [64] was proposed by Papadomanolakis, myself, and other researchers recently (2006) to implement point queries on a tetrahedral mesh stored in a traditional database table. The major difference between DLS and the `abacus_reset` function is that the walking operations in DLS follow a kind of out-of-core pointers, while the walking operations within the Abacus cache follow memory

pointers to adjacent triangles in most cases. In terms of performance, Abacus is comparable to any efficient incore implementation of the stochastic walk algorithm.

In the process of walking, if we fail to reach a triangle across an edge if the triangle is not cached yet, we use the techniques described in Section 4.10 to load the missing triangle. Recall that we load a pageful of triangle records at a time. The cost of fetching a page is amortized among the prefetched triangles, which are likely to be "walked" through shortly.

```
#include <abacus.h>

int_t abacus_walk(abacus_t *ap, const simplex_t *from, int_t id,
simplex_t *to);
```

Returns: 0 on success, -1 on error

The `abacus_walk` function searches the Abacus cache and returns the target simplex as the result of walking from a source simplex across either an edge in 2D or a face in 3D.

This function receives the following parameters:

- `ap`: Handle to the Abacus database to be searched.
- `from`: Specifies current simplex where the walk is initiated.
- `id`: The edge id (2D) or the face id (3D) of the `from` simplex that is to be crossed.
- `to`: An output parameter that points to either a `triangle_t` or a `tetrahedron_t` structure.

This function returns 0 on success and -1 on error. On success, the target simplex record is stored in the `to` parameter.

The implementation of `abacus_walk` is trivial. We first call `abacus_reset` to find the starting triangle/tetrahedron of the walk and then either directly walk to the neighbor within the Abacus triangle structure or load the neighbor triangle from the triangle database if it is not cached yet. Either way, we retrieve the neighbor's record and store it in the output parameter.

4.12 Putting it All Together: Delaunay Triangulation Reimplemented

This section describes our second attempt to attack the problem of Delaunay triangulation after the structural mismatch has baffled our first attempt in Chapter 2. This time we build our solution on top of Abacus.

4.12.1 Balance Between Randomization and Locality of Reference

We have mentioned a number of times in the previous sections the importance of locality of reference to the performance of Abacus. Yet we have glossed over an important aspect of the performance: the algorithm complexity.

The *expected* running time of the incremental insertion algorithm for constructing Delaunay triangulations is $O(n \log n)$. In order to achieve the optimal *expected* running time, the insertion order of the vertices must be randomized. However, randomization eliminates the inherent spatial locality in a vertex set.

To achieve good performance, we must strike a balance between randomization and locality of reference. Ignoring randomization, we may turn incremental insertion from an $O(n \log n)$ algorithm into an $O(n^2)$ algorithm, for example, if the input vertices are sorted along the axes. On the other hand, ignoring locality, we may suffer from severe I/O thrashing, for example, if consecutive insertions are scattered far apart in the domain. Both would bring down the performance to a crawl.

Below we explain how we build on a prior research result to organize an input vertex database in such a way that we can achieve the optimal running time $O(n \log n)$ without suffering from I/O thrashing.

Biased Randomized Insertion Order (BRIO)

Amenta and colleagues developed an algorithm called *Biased Randomized Insertion Order* (BRIO) [6] that addresses the tension between randomization and locality. The BRIO algorithm assumes the input vertices have some fixed ordering that respects locality, for example, a space-filling curve ordering. Their main idea is to add enough randomness to the vertex set so that the incremental insertion algorithm remains theoretically optimal, and in the meantime retain enough locality so that the performance of the virtual memory system is improved.

A *biased randomized insertion order* for a set of n vertices (assuming n is a power of 2) is defined as follows. The vertices are inserted into a Delaunay triangulation in *rounds*, from round 0 through round $\log n$. To allocate vertices to rounds, they choose each vertex independently with probability 1/2 to be inserted in the final round. They choose each of the remaining vertices independently with probability of 1/2 to be inserted in the next-to-last round, and so on. When reaching round 0, they choose any remaining vertices with probability 1. The proof that such an insertion order is theoretical optimal can be found in [6].

Conceptual, we can think of the result of a BRIO order as a pyramid. At the top layer are the vertices that will be inserted in the first round; and at the bottom layer are the vertices that will be inserted in the last round. Intuitively, the effect of the BRIO order is to scatter vertices randomly in the entire domain in early rounds to obtain a well-shaped coarsen-grained Delaunay triangulation, and then insert spatially-clustered vertices in small regions in later rounds.

The BRIO order has been successfully used in practice, for example, in streaming computation of large-scale Delaunay triangulations [45].

Redundant Insertion

However, we cannot use the BRIO order directly for two reasons. First, in order to keep track of the BRIO insertion order, we have to either create a flat file to record the vertex ids in the BRIO order or build another vertex table to record the BRIO order. Not only does it clutter the database structure, but it also introduce confusion because there are two primary keys associated with vertices now, one is the original vertex id and the other is the sequence number of the vertex

in the BRIO order. Second, the BRIO insertion order can result in a large number of intermediate triangles whose vertices are widely spread across vertex id space. For example, a triangle may consist of vertices (100, 2000000, 50000). Since we have to query the database to retrieve vertices, the large disparity in the vertex id values may cost us three separate I/O operations to retrieve the vertices of a single triangle.

We have adapted the BRIO algorithm slightly to suit our need. Our algorithm also works from the last round backwards to the first round. Initially, we include all the input vertices in the the last round of insertion. Then we carry out *promotion* of vertices starting from the last round. For each vertex in the current round, we toss a coin. With probability p, we *promote* the vertex to the previous round. Unlike the BRIO algorithm, we still *keep* the vertex in the current round. After the vertices in the current round are processed, we move to the previous round, which has just been created, and repeat the promotion process. The algorithm terminates when the number of vertices in the current round is less than the capacity of a vertex data page.

The difference between this algorithm and the BRIO algorithm is that a vertex may exist in a number of consecutive rounds instead of just one round. That is, if a vertex exists in round k, it must also exists in round $(k+1)$, $(k+2)$, and the all the way to last round. Otherwise, it would not have been promoted through the hierarchy. As a result, the vertex will appear multiple times in the insertion order.

For the occurrence of a vertex in round k, we assign the vertex an *artificial* new id based on its assigned id in round $(k+1)$. All the vertices have their *original* ids in the last round.

Artificial Vertex IDs

We assume that the input vertices have vertex ids starting from 0 and we can bound the maximum number of vertices that can be possibly added into the vertex set. We define the range from 0 to the maximum bound as an *epoch*.

For example, Figure 4.44 shows a vertex set with 30 vertices, represented as tick marks between 0 and 29. For illustration purpose, we define the epoch to be 100. The start of the current epoch is at 0.

When we start processing vertices in the current round, in our example, the last round, we create a new epoch to the left of the current epoch. The two epochs are adjacent to each other but do not overlap. In our example, the start of the new epoch is at -100.

For each vertex being promoted into the previous round, we take its current vertex id and add it to the origin of the new epoch. For example, the numbers enclosed by parentheses in the figure represent the vertex ids of three vertices that have been promoted (8, 18, 25). The artificial vertex ids for the three vertices in the previous round are -92 (i.e., 8 + (-100)), -82 (i.e., 18 + (-100)), and -75 (i.e., 25 + (-100)), respectively.

We repeatedly shift the epoch to the left when more rounds are needed. The smaller the round number, the smaller the artificial vertex ids in that round.

Effectiveness of the New Algorithm

If we ignore the re-occurrence of vertices in later rounds, our algorithm is identical to the BRIO algorithm. Randomness has been added into the insertion order.

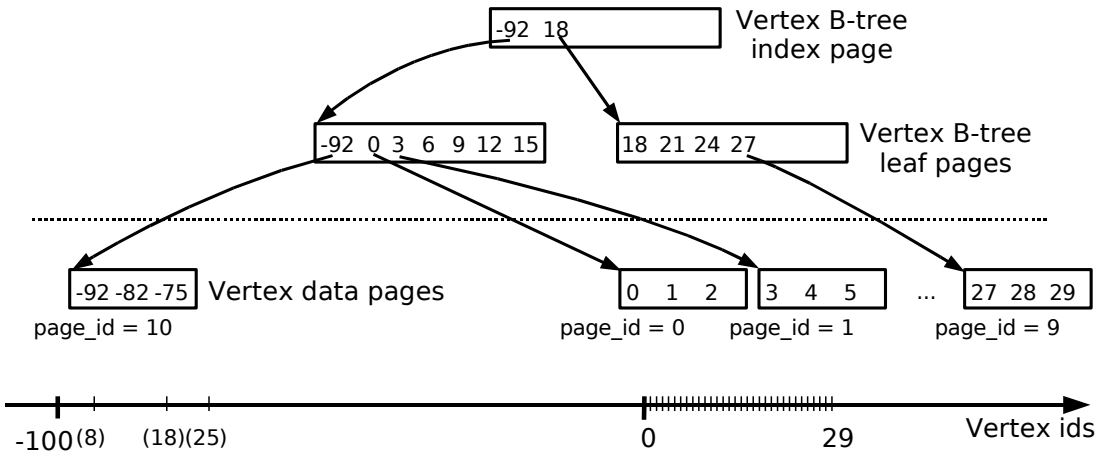


Figure 4.44: **Staging vertices into the vertex database.** The coordinates of the vertices are not shown in the figure. We assume a promote probability of 10%.

Now consider the redundancy in the insertion order. At the end of round k, all the vertices inserted between round 0 and round $(k-1)$ must have been replaced with the vertices in round k because the vertices in round k represent a proper superset of all the vertices in the previous rounds. The triangles in the triangulation at the end of round k consist of only artificial vertex ids drawn from the epoch associated with round k. If we look at the vertices within an epoch as shown in Figure 4.44, their artificial ids are clustered in the vertex id space. Locality of reference has been achieved.

The tunable parameter for striking a balance between randomization and locality is the promotion probability p. If we set p too low, we fail to introduce enough randomness into the insertion order. If we set p too high, we pay the extra cost of redundant insertions. In theory, p should be set to 1/2, as proved by the BRIO paper. Doing so would result in the total number of vertices to be inserted to be doubled (i.e., $1 + (1/2) + (1/2) \times (1/2) + \ldots$)). In practice, we have found a p value of 0.02 works more efficiently.

4.12.2 Staging the Vertices

Given a vertex set, we load the vertices into an Abacus vertex database by calling the following function.

```
#include <abacus.h>

int32_t abacus_stage_vertices(abacus_t *ap, const char *vertexset);
```

Returns: 0 on success, -1 on error

The `abacus_stage_vertices` function implements the modified BRIO algorithm just described and bulk-loads the results into an Abacus vertex database.

This function receives the following parameters:

- `ap`: Handle to the Abacus database.
- `vertexset`: The file name of the input vertex set.

This function returns 0 on success, -1 on error. Figure 4.44 shows the result of calling the function to stage 30 vertices.

We implement `abacus_stage_vertices` in a number of iterations. In the first iteration (i.e., the last round of the insertion order), we open the input file and process all the vertices in order. We pack them into vertex data pages, assign vertex page ids starting from 0, store the vertex data pages to disk, and insert keys (i.e., the first vertex id on each page) into the vertex B-tree. While going through the vertices, we also promote vertices for the next iteration (i.e., the previous round), compute their artificial vertex ids, and store them one after another in a temporary flat file.

After the first iteration is finished, we move to the next iteration using the temporary flat file just created as the input file. Note that at the second iteration, the free vertex data page id has increased to the number of vertex pages that have already been stored in the first iteration.

We repeat the iterations until the remaining vertices in a temporary file is less than the capacity of the vertex data page. The last iteration simply store and index the vertex data page without further promotion of vertices.

The output of the `abacus_stage_vertices` function is a vertex table and a vertex B-tree that store and index not only the original input vertices but also the redundant vertices that have been promoted to the earlier rounds, as shown in Figure 4.44.

From the Abacus cache's point of view, there is no difference between an artificial vertex ids and an original vertex id. All vertices are treated equally. If two vertices have the same coordinate, the second one replaces the first one in the triangulation. Since all the original input vertices are inserted in the last round, the output triangle database does not contain any artificial vertex ids.

4.12.3 Triangulating the Vertices

```
#include <abacus.h>

int32_t abacus_delaunay_insert(abacus_t *ap, int64_t vid, double
*coord);
```

Returns: 0 on success, -1 on error

The `abacus_delaunay_insert` function implements the incremental insertion algorithm as explained in Chapter 2.

This function receives the following parameters:

- `ap`: Handle to the Abacus database to be searched.
- `vid`: Specifies the id of the new vertex to be inserted.

- `coord`: Specifies the coordinate of the vertex.

This function returns 0 on success and -1 on error. On success, the function adds the vertex into an existing triangulation managed by the Abacus system and restores the empty circle property within the triangulation. If the triangulation is Delaunay before the function is called, the triangulation remains Delaunay after the function returns

We implement `abacus_delaunay_insert` using the incremental insertion algorithm presented in Chapter 2 almost verbatim. This is the beauty and the power of the computational database approach. The data representation within the Abacus cache emulates how a triangulation is represented if the incremental insertion algorithm is implemented incore. There is no need to devise a work-around as we did with the Goose implementation.

The two steps of inserting a new vertex into an existing triangulation map naturally to operations in the Abacus cache.

Cavity creation

The cavity creation step finds all triangles whose circumcircles enclose the new vertex and forms a cavity. The internal data structure used to carry out the cavity creation step is a queue that keeps track of the triangles that fall inside the cavity. Since these triangles are to be deleted, we refer to the queue as the *delete queue*. Initially the delete queue is empty. We append triangles to the queue by conducting a breadth-first search within the Abacus triangle structure.

The starting triangle for the breadth-first search is located by executing an *internal* version of the `abacus_reset` function. Since we do not need to return the result to an application, the internal version returns a pointer to the incore triangle record that encloses the new insertion vertex rather than copying the results into an output parameter. We then add the pointer to the starting triangle into the delete queue.

For each triangle we read from the head of the queue, we check its three neighbors in turn. For each neighbor, if its circumcircle encloses the new insertion vertex, it is appended to the end of the queue. Otherwise, it is ignored. If we cannot reach a neighbor, we use the techniques described in Section 4.10 to load the missing triangle. The breadth-first search terminates when all the triangles inserted in the queue are processed, that is, when we cannot expand the cavity further.

It should be clarified that the delete queue contains pointers to the incore triangle records that are stored in simplex blocks instead of the records themselves. The effect of creating the cavity is to load all *encroached* triangles into the Abacus cache.

Re-triangulation

The re-triangulation step deletes the triangles stored in the delete queue by calling the `abacus_delete` function, and inserts new triangles by connecting the new vertex with the edges of the cavity and calling the `abacus_insert` function.

The deletion operations are grouped together and executed first before any new triangle is inserted. Otherwise, cross-over triangles would appear and destroy the coherence of the Abacus triangle structure.

The `abacus_delaunay_insert` is implemented as a standalone function for two reasons. First, the algorithm needs to keep track of pointers to the incore triangle structure, which is not visible to an application. Second, the re-triangulation step has a stringent requirement of deletions before insertions. It should be implemented within the Abacus system to ensure correctness.

4.12.4 Application Cache

The cavity creation step just described follows the standard procedure of locating a point within the Abacus cache as illustrated in Figure 4.40 and Figure 4.42. It requires traversing down the octree to locate the enclosing leaf octant first, and then starting from an arbitrary triangle in the associated simplex block to walk to the target triangle.

When a triangulation is small, the octree is shallow. The standard point location procedure works well. However, when a triangulation is large, the octree is deep. We pay the overhead of traversing down a deep octree every time we insert a new vertex.

In order the alleviate the problem, we have devised a technique called an *application cache* to accelerate the process of finding the first triangle in the cavity. The structure of the application cache is an array of pointers, each pointing at a newly created incore triangle record. The size of the cache is 4 entries for 2D and 16 entries for 3D.

Because real-world vertex sets exhibit inherent *spatial coherence* [45], that is, there is strong correlation between the spatial proximity of the vertices and the proximity of their positions in a stream (i.e., the vertex id sequence), the vertex to be inserted next is highly likely to be enclosed by one of the newly created triangles incident to the current insertion vertex. Hence, instead of following the standard point location procedure, we first check the application cache to see if there is a hit.

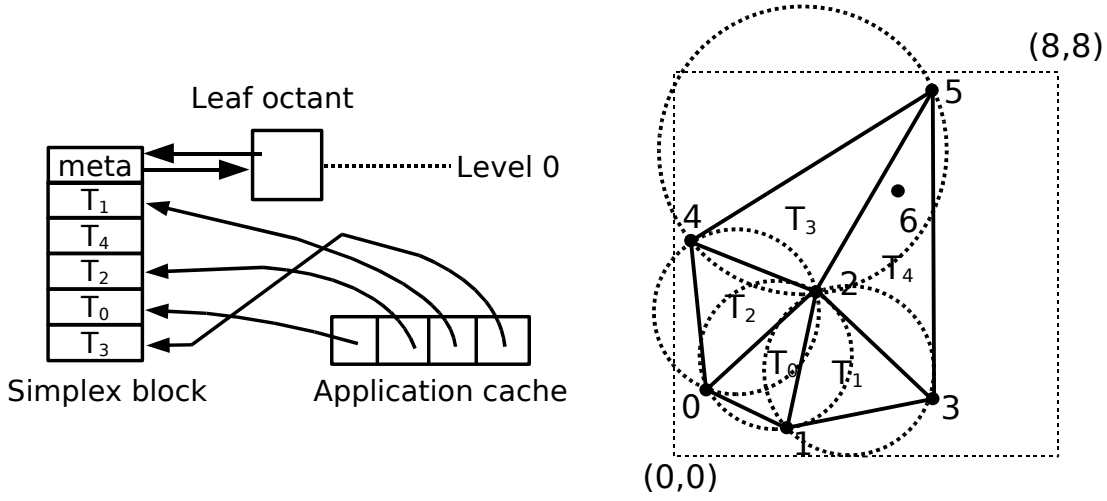
The criterion of a hit is also relaxed. We do not require a strict enclosure. As long as the circumcircle of a triangle encloses the new insertion vertex, it is a hit, which is referred to as an *incircle cache hit*. The correctness of the cavity creation step is not compromised. Any triangle that falls inside the cavity can be used as the base for a breadth-first expansion. Because an incircle cache hit returns a triangle whose circumcircle encloses a new vertex, the triangle must be part of the cavity. Hence, we can safely push it into the delete queue and start the breadth-first search from there.

For example, Figure 4.45(a) shows the status of the application cache after vertex 2 is inserted into a Delaunay triangulation. We assume triangles $\mathtt{T}_0, \mathtt{T}_1, \mathtt{T}_2$ and $\mathtt{T}_3$ are tracked by the application cache. Figure 4.45(b) shows the insertion of a new vertex 6. Although none of the triangles in the application cache encloses vertex 6, we are able to get an incircle cache hit since the circumcircle of $\mathtt{T}_3$ encloses vertex 6.

4.12.5 A New Delaunay Triangulation Program

The pseudo-code of our new Delaunay triangulation program is shown in Algorithm 1.

We first open an Abacus database and stage the vertices into the vertex database using the modified BRIO algorithm implemented in the `abacus_stage_vertices` function. Then we insert two initialization Delaunay triangles into the Abacus triangle database. All input vertices



(a) Status of the application cache. (b) Vertex 6 enclosed by T_3.

Figure 4.45: **Incircle cache hit.**

Algorithm 1: Constructing a Delaunay triangulation using Abacus

Input: A flat file containing a set of input vertices
Output: An Abacus triangulation database

```
1  ap = abacus_open(dbname, O_CREAT|O_RDWR, dim, 0, near, far, cachesize);
2  abacus_stage_vertices(ap, vertexset);
3  abacus_insert(ap, bounding_triangle_0);
4  abacus_insert(ap, bounding_triangle_1);
5  create a vertex record cursor;
6  set the cursor to the first record in the vertex table;
7  while (not at the end the vertex table) do
8  |   read the current vertex id and its coordinate;
9  |   abacus_delaunay_insert(ap, vertex_id, coordinate);
10 |   advance the cursor;
11 endw
12 abacus_close(ap);
```

must fall inside the two initial triangles. Next, we create a cursor to iterate through the vertex table in an ascending vertex id order. We call the `abacus_delaunay_insert` function repeatedly to insert the vertices into the Abacus database until the cursor reaches the end of the vertex table. The output of the algorithm is an Abacus triangulation database that contains a vertex table, a vertex B-tree index, a triangle table, and a triangle B-tree index.

A caveat: We insert 2 initial bounding triangles (or 5 initial bounding tetrahedra for 3D) at the beginning of Delaunay triangulation process. This is a makeshift measure we use to avoid the problem of convex hull expansion so that we can focus on the Delaunay triangulation proper. In the long term, we need to develop new computational database algorithms for constructing convex hulls dynamically.

The algorithm just outlined can be used in a number of flexible ways. For example, we can suspend an ongoing triangulation process (though not in the middle of inserting a particular vertex), close the Abacus database, and flush the data to disk. Later, we can reopen the Abacus database and continue the insertions. For long running jobs, which take days or weeks to finish, the ability to checkpoint is a huge plus. Or, we can gradually expand a Delaunay triangulation as data become available. For example, survey data often become available in small pieces. It may take months or even years to collect a complete dataset. Using Abacus, we can triangulate a partial dataset and add more data later.

The translation mechanism of exchanging data between the Abacus cache and the database presented in this chapter may have appeared to be too heavy-weight to be efficient. In fact, they are not. The next chapter demonstrates the performance and scalability of our techniques for Delaunay-triangulating massive 2D and 3D datasets.

Chapter 5

Performance Evaluation

One of the driving motivations for designing the computational database framework in general and implementing the Abacus system in particular is to solve the problem of large-scale Delaunay triangulation. Although specific to a certain class of applications, Delaunay triangulations are among the most challenging data sets in terms of complexity and scale. Whether Abacus can effectively support Delaunay triangulation provides a litmus test of the feasibility of the proposed computational cache.

This chapter presents the performance evaluation of constructing massive 2D and 3D Delaunay triangulations using Abacus. Our main focus is on the speed and the scalability of our new Delaunay triangulation solution. To put the performance in perspective, we compare our results with those of the state-of-the-art incore Delaunay triangulators, *Triangle* [1] [79] and *Pyramid* [81].

The main findings of this chapter are: (1) Abacus matches the performance of Triangle and Pyramid when triangulating data sets that fit in memory, and (2) Abacus *significantly* outperforms Triangle and Pyramid when triangulating data sets larger than the memory size.

[1] 2003 James Hardy Wilkinson Prize in Numerical Software, the highest honor for numerical software awarded once every 4 years.

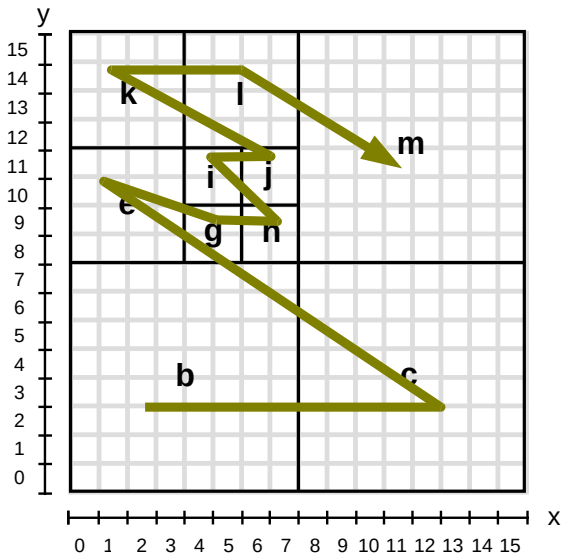


Figure 5.1: **Z-order traversal in a domain.**

5.1 Data Set Characteristics

We used both 2D and 3D data sets to evaluate the effectiveness of our solution. The 2D data sets were generated randomly and the 3D data sets came from a grand challenge application—earthquake ground motion modeling. Below we describe the characteristics of the data sets.

5.1.1 The 2D Data Sets

For the 2D cases, we generated a series of random data sets, each consisted of a fixed number of vertices distributed randomly within a 1 by 1 square. However, a sequence of random vertices are not sufficient to approximate the characteristics of real-world data sets. Adjacent vertices (in the vertex id order) in a random data set are often scattered far apart in the domain, while adjacent vertices in a realistic data set tend to be clustered in space. For example, the terrain data set of the Neuse River Basin of North Carolina, consisted of 500 million 2D vertices (double-precision x, y coordinates and a height value), were captured by an airborne laser scanner [68]. Given the continuous flying path of an aircraft, a sequence of consecutive vertices can hardly be scattered far apart.

To introduce spatial locality into the randomly generated data sets, we sorted the vertices into Z-order [93, 98], a space-filling curve ordering that tends to cluster spatially close vertices in the linearized ordering, as shown in Figure 5.1.

The choice of the Z-order is of no particular significance to our evaluation. Other space-filling curve orderings such as the Hilbert-Peano curve [27] can also be used and might result in better performance. What is important is that we need to introduce *some* form of spatial locality into our data sets. Sorting them in Z-order is merely a convenient solution.

Figure 5.2 summarizes the the sizes of the Z-ordered random data sets. The first row lists the numbers of vertices within the data sets. “M” stands for million and “B” for billion. (We will

this convention throughout this chapter.) The second row shows the sizes of the files storing the data sets. Vertices are stored in files in binary format. Each vertex contains an 8-byte vertex id and a 16-byte coordinate (i.e., 8-byte double-precision floating point number $\times$ 2).

Number of vertices	1 M	10 M	40 M	100 M	200 M	1 B
File size	24 MB	240 MB	960 MB	2.4 GB	4.8 GB	24 GB

Figure 5.2: **Summary of the 2D vertex sets.**

5.1.2 The 3D Data Sets

The 3D data sets we used for our experiments came from a real-world application—earthquake ground motion modeling. The models were were built by a parallel octree mesh generator [97, 99].

Vertices in these models were distributed irregularly in the domain as shown in Figure 5.3. The variation in the density of distribution of the vertices was due to the variation of soil material properties in the domain. Since the vertices in these data sets were organized in a semi–Z-order by the parallel octree mesh generator [99], we did not sort them or conduct any other pre-processing. In practice, we expect most real-world data sets have similar inherent spatial locality.

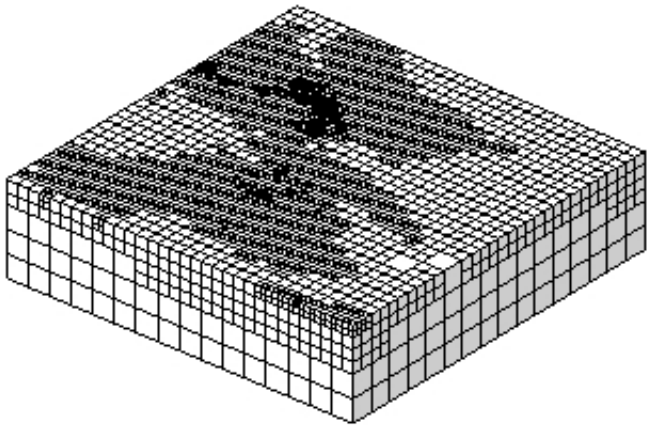


Figure 5.3: **A 3D earthquake model data set.** Only the exterior of the vertex set is shown.

For our experiments, we used the models generated for the Greater Los Angeles Basin, a 3D volume of 100 km x 100 km x 37.5 km. The sizes of the data sets are listed in Figure 5.4. The "Frequency" row shows the seismic frequencies for which the models were built to resolve. The higher the frequency, the higher the resolution (of a simulation) and the larger the data set.

Frequency	0.25 Hz	0.5 Hz	0.75 Hz	1 Hz	1.5 Hz	1.85 Hz
Number of Vertices	1.3 M	11.3 M	35.7 M	134.0 M	534.5 M	1.0 B
File size	40 MB	362 MB	1.14 GB	4.29 GB	17.10 GB	32.75 GB

Figure 5.4: **Summary of the 3D vertex sets.**

5.2 System Configurations

All our experiments were conducted on a server with two Intel 3.6 GHz Xeon processors running Linux 2.6.17. (Only one processor was used in the experiments.) The memory subsystem consisted of 8 GB physical memory and 18 GB swap space. The same machine was used to conduct the experiments for the Goose implementation described in Chapter 2.

All the programs were compiled with `gcc` using the `-O2` optimization flag. The database page size used in vertex tables, vertex B-trees, simplex (triangle and tetrahedron) tables and simplex B-trees was set to be 16 KB.

5.3 Execution Time

The total running time of constructing Delaunay triangulations using the Abacus system consists of two parts: (1) staging the vertices, and (2) triangulating the vertices. Below, we present the running time of the two parts, respectively, and compare the overall running time with the state-of-the-art incore Delaunay triangulators.

5.3.1 Staging the Vertices

The promotion probability we used to stage the vertices was 0.02, which was an engineering choice rather than a theoretical must. Figure 5.5 shows the running time and throughput of staging the 2D vertex sets. The first two rows list the numbers of vertices and the corresponding file sizes. The "Vertex database" row shows the aggregate sizes of the vertex databases, which are the sums of the sizes of vertex tables and vertex B-trees listed below. The "Staging time" row lists the wall clock times of staging the vertex sets. The "CPU utilization" row shows the percentages of the staging time when the CPU was busy. The "user time" and "system time" rows report the times spent by the staging operation executing the user code and the system code, respectively. The "Throughput" row shows the throughput of the staging operation in terms of how many megabytes of input vertex data are processed every second.

The table reveals three interesting phenomena. First, the staging operation is I/O bound. The CPU utilization rate is consistently around 22%. We waited for I/O to complete for most of the time. The breakdown of the (busy) CPU time shows that the ratio between the user time and the system time is about 1:5, an indication that the staging operation spent significantly more time in executing system calls than generating random numbers to promote vertices.

Second, the throughput of staging, though fluctuated, scales reasonably well. It reflects the effectiveness of the bulk-loading procedure described in Section 4.12.2. If we need to stage a data set with 10 billion vertices, we can make an educated guess of how long it may take. For

Number of vertices	1 M	10 M	40 M	100 M	200 M	1 B
Flat file size	24 MB	240 MB	960 MB	2.4 GB	4.8 GB	24 GB
Vertex database	25 MB	246 MB	981 MB	2.5 GB	4.90 G	24.5 GB
Vertex table	24.5 MB	245 MB	980 MB	2.5 GB	4.9 GB	24.5 GB
Vertex B-tree	65 KB	279 KB	1 MB	2.4 MB	5 MB	24 MB
Staging time	00:00.5	00:03.6	00:27	01:08	02:25	12:29
CPU utilization	24%	38%	22%	23%	22%	21%
User time (s)	0.02	0.24	1.2	3.1	5.6	28
System time (s)	0.1	1.15	4.9	13	27.3	133
Throughput (MB / s)	91	134	72	71	67	65

Figure 5.5: **The running time of staging the 2D vertex sets.** The "Staging time" is shown in (mm:ss) format. The "s" shown in the parentheses stands for seconds.

large-scale applications, the ability to provide some indication of the running time is often helpful by itself. For example, if an estimated running time far exceeds a deadline, the problem size can be scaled down properly.

Third, the sizes of the vertex B-trees are about three orders of magnitude smaller than the vertex tables. This is because each 16 KB vertex data page is tracked by one index entry, which is 16 bytes in size (i.e., an 8-byte vertex id plus an 8 byte page id). The ratio between the two happens to be 1000. Given the small sizes of the vertex B-tree indices, they are likely to be completely cached in the file system buffer cache at runtime. Also note that the sizes of vertex tables are almost identical to those of the input flat files. This is because we have fully packed the vertex pages when we stage the vertex database. As such, we make full use of the I/O bandwidth when we fetch data pages of the vertex table.

Figure 5.6 shows the running time and throughput of staging the 3D vertex sets.

Number of vertices	1.3 M	11.3 M	35.7 M	134.0 M	534.5 M	1.0 B
Flat file size	40 MB	362 MB	1.14 GB	4.29 GB	17.1 GB	32.8 GB
Vertex database	41 MB	370 MB	1.17 GB	4.39 GB	17.5 GB	33.5 GB
Vertex table	41 MB	370 MB	1.17 GB	4.38 GB	17.5 GB	33.5 GB
Vertex B-tree	80 KB	400 KB	1.2 MB	4 MB	32 MB	33 MB
Staging time	00:01	00:16	00:48	02:54	11:21	22:19
CPU utilization	21%	13%	13%	14%	15%	15%
User time (s)	0.04	0.3	1	4	15	30
System time (s)	0.2	1.8	6	21	88	171
Throughput (MB / s)	71	46	48	50	51	49

Figure 5.6: **The running time of staging the 3D vertex sets.**

The overall characteristics of staging 3D data sets are similar to those of the 2D cases. There are two minor differences, though. The CPU utilization rate and the throughput rate both drop for about 33%. This is due to the additional coordinate we have to process. For each vertex in 3D, we have to read in and write out 32 bytes, that is, an 8-byte vertex id plus a 24 byte 3D coordinate. In two 2D, each vertex is 24 bytes. Hence, there is a difference of about 33% in the

size of the records, which is reflected in the proportional drop in the CPU utilization rate and the throughput rate.

5.3.2 Triangulating the Vertices

The more time consuming part of the our solution is Delaunay triangulating the vertices. Figure 5.7 shows the running time of triangulating the 2D data sets. The Abacus computational cache size was set to be 8 MB for these experiments.

The first row lists the number of vertices in the data sets. The second row shows the number of triangles stored in the final triangulation. The "Triangle database" row shows the aggregated file size of the triangle databases, which are the sums of sizes of the triangle tables and triangle B-trees listed below, respectively. The "Triangulating time" row shows the running times of triangulating the data sets. The "Amortized time" row shows how long it takes to triangulate 1 million points on average for each data sets.

Number of vertices	1 M	10 M	40 M	100 M	200 M	1 B
Number of triangles	2M	20 M	80 M	200 M	400 M	2 B
Triangle database	67 MB	817 MB	3.1 GB	8.3 GB	16.4 GB	69.0 GB
Triangle table	67 MB	814 MB	3.1 GB	8.3 GB	16.3 GB	68.7 GB
Triangle B-tree	280 KB	3 MB	13 MB	30 MB	53 MB	269 MB
Triangulating time	00:00:18	00:03:30	00:15:19	00:43:10	01:18:16	07:17:12
CPU utilization	99%	98%	94%	94%	93%	92%
User time (s)	18	204	860	2416	4300	23901
System time (s)	0.17	3	10	41	71	369
Amortized time (s)	18	18	23	25	24	26

Figure 5.7: **The running time of Delaunay triangulating the 2D vertex sets (8 GB physical memory, 8 MB Abacus computational cache).** The triangulating time is represented in (hh:mm:ss) format. The amortized time represents the average time (in seconds) to triangulate 1 million vertices, that is, the total triangulation time divided by the number of millions in a vertex set.

The table illustrates a number of interesting results. First of all, our new solution takes only 18 seconds to triangulate 1 million vertices. Compared to the running time (25 hours 43 minutes and 35 seconds, or 92615 seconds) of the R-tree based Goose implementation presented in Chapter 2, we have achieved a speedup of more than 5000 times.

Second, the performance of Abacus scales exceptionally well. The amortized time for triangulating 1 million vertices increase moderately as expected. Recall that the complexity of the incremental insertion algorithm is $O(n \log n)$. By dividing the total triangulation time by the number of millions, we *should* expect $O(\log n)$ as the amortized running time. As the problem size n increase, the amortized running time should increase moderately.

Note how the performance holds up even when triangulating the 1 billion vertex data set. The total size of the data set is 93 GB (i.e., 24 GB vertex database plus 69 GB triangle database), which is an order of magnitude larger than the physical memory size (8 GB). There is no performance degradation of any kind.

The third observation is that the triangulation process is CPU-bound. The overall CPU utilization rates are always above 90%. Of the busy CPU time, about 98% is spent in executing the triangulation (user) code. This is a strong proof that constructing large-scale Delaunay triangulation on Abacus is compute-intensive. Most of the wall clock time is spent in executing the triangulation code rather than making system calls.

We believe that running the experiments on a dual-CPU machine has contributed to the overlapping of I/O with the normal execution of our program. Since we have used only one processor, the OS daemons must have used the other processor to swap in and out disk pages in the background. However, we have not yet quantitatively analyzed the effect of the dual-processor setup. It is an interesting line of research to pursue in the future.

Besides setting the Abacus computational cache to 8 MB, we also experimented with larger cache sizes, ranging from 16 MB to 4 GB. The results were all within 5 percent of the running time reported in Figure 5.7, which suggests that the working set of 2D Delaunay triangulation is quite small. As long as there is some amount of memory allocated for the Abacus computational cache, the performance of Delaunay triangulation is boosted.

Figure 5.8 shows the running time of triangulating the 3D data sets. We set the Abacus computational cache size to 6 GB for these experiments. The rows of the table have the same meaning as those in Figure 5.7.

Frequency	0.25 Hz	0.5 Hz	0.75 Hz	1 Hz	1.5 Hz	1.85 Hz
Number of vertices	1.3 M	11.3 M	35.7 M	134 M	534.5 M	1.0 B
Number of tetrahedra	7 M	63.4 M	200.1 M	738 M	3.0 B	5.7 B
Tetrahedron database	330 MB	3 GB	9.6 GB	36.8 GB	137.2 GB	256.0 GB
Tetrahedron table	329 MB	3 GB	9.5 GB	36.6 GB	136.6 GB	255.0 GB
Tetrahedron B-tree	1 MB	12 MB	38 MB	146 MB	546 MB	1 GB
Triangulating time	00:05:47	00:54:23	03:09:08	12:23:56	51:07:00	102:36:54
CPU utilization	98%	99%	97%	97%	97%	93%
User time (s)	334	3227	11083	43278	178394	346468
System time (s)	8	14	32	111	405	757
Amortized time (s)	275	289	318	333	344	361

Figure 5.8: **The running time of Delaunay triangulating the 3D vertex sets (8 GB physical memory, 8 MB Abacus computational cache).** The triangulating time is represented in (hh:mm:ss) format. The amortized time represents the average time (in seconds) to triangulate 1 million vertices, that is, the total triangulation time divided by the number of millions in a vertex set.

The performance characteristics of the 3D cases are almost identical to those of the 2D cases. The difference in the amortized times between the 2D and 3D cases are due to the extra work for dealing with 3D data sets. For the 2D random data sets, the average number of deletions caused by a new insertion is 4.7 and the average number of of new insertions is 6.6. In contrast, for the 3D earthquake data sets, the average number of deletions caused by a new insertion is 36.1 and the average number of new insertions is 41.6. Adding other overhead such as the breadth-first search, the overall cost of triangulating 1 million 3D vertices is about 12 times that of triangulating 1 million 2D vertices.

The most striking result shown in Figure 5.8 is that Abacus produced 3 billion and 5.7 billion tetrahedra for the 1.5 Hz and 1.85 Hz data sets, respectively. The sizes of the output tetrahedra databases alone are 137 GB and 256 GB, respectively, far exceeding the size of the 8 GB physical memory.

That we are able to generate billion-element Delaunay tetrahedral meshes on commodity servers represents a breakthrough new capability. Previously, meshes of such scale and complexity *had to* be generated by parallel programs running on supercomputers with hundreds of gigabytes to terabytes physical memory.

5.3.3 Comparison with Incore Delaunay Triangulators

In order to assess the performance of Abacus in a meaningful way, we conducted experiments to triangulate the 2D and 3D data sets using the state-of-the-art incore Delaunay triangulators, Triangle and Pyramid.

The inputs to Triangle and Pyramid did not include the redundant vertices we added into the vertex databases. Both program took the flat vertex files as input directly. All the experiments were conducted on the same server as used in previous sections.

The 2D cases

An interesting observation we made was that Triangle ran about 3 times faster triangulating the Z-ordered vertex sets than the corresponding random data sets. For example, it took Triangle 21 seconds to triangulate 1 million Z-ordered random data set, while it took Triangle 1 minutes to triangulate the original random data sets. To ensure a fair comparison, we used the Z-ordered random data sets as inputs to Triangle.

Besides the incremental insertion algorithm, Triangle also implements the divide-and-conquer algorithm for constructing Delaunay triangulations, which is the fastest 2D Delaunay triangulation algorithm in practice. We compare our results to both implementations of Triangle.

Figure 5.9 shows the running time of triangulating the 2D data sets using Abacus, Triangle executing the incremental insertion algorithm, and Triangle executing the divide-and-conquer algorithm, respectively.

The “Abacus time” shows the total time of executing the Delaunay triangulation program shown in Section 4.12.5, which include the time of staging the vertices,“Staging time”, and the time of triangulating the vertices, “Triangulating time”. The “Abacus amortized time” shows how many seconds Abacus takes to process 1 million vertices on average. “Triangle (ii)” represents the execution time of Triangle running the incremental insertion algorithm. “Memory usage (ii)” lists the amount of memory used by Triangle running the incremental insertion algorithm. For the 1 billion vertex set, Triangle ran out of the virtual memory (26 GB). The 114 GB shown in the table is a linearly extrapolation estimate of the memory usage based on the memory usage by the smaller data sets. “Triangle (ii) amortized time” shows how many seconds Triangle, executing the incremental insertion algorithm, takes to process 1 million vertices on average. The three rows below have the same meaning as the previous three rows except that they represent the case of Triangle running the divide-and-conquer algorithm.

Number of vertices	1 M	10 M	40 M	100 M	200 M	1 B
Number of triangles	2M	20 M	80 M	200 M	400 M	2 B
Abacus time	00:00:19	00:03:34	00:15:46	00:44:18	01:20:41	07:29:41
Staging time	00:00:01	00:00:04	00:00:27	00:01:08	00:02:25	00:12:29
Triangulating time	00:00:18	00:03:30	00:15:19	00:43:10	01:18:16	07:17:12
Abacus amortized time (s)	19	21	24	27	27	27
Triangle (ii)	00:00:21	00:06:22	00:38:06	02:39:51	09:15:49	N/A
Memory usage (ii)	114 MB	1.1 GB	4.6 GB	11.4 GB	22.8 GB	114 GB (est.)
Triangle (ii) amortized time (s)	21	38	57	96	167	N/A
Triangle (dc)	00:00:08	00:01:19	00:06:01	01:07:51	09:55:47	N/A
Memory usage (dc)	122 MB	1.2 GB	4.9 GB	12.2 GB	24.4 GB	122 GB (est.)
Triangle (dc) amortized time (s)	8	8	9	41	179	N/A

Figure 5.9: **Running time comparison between Abacus and Triangle.** "Triangle (ii)" represents Triangle executing the incremental insertion algorithm. "Triangle (dc)" represents Triangle executing the divide-and-conquer algorithm.

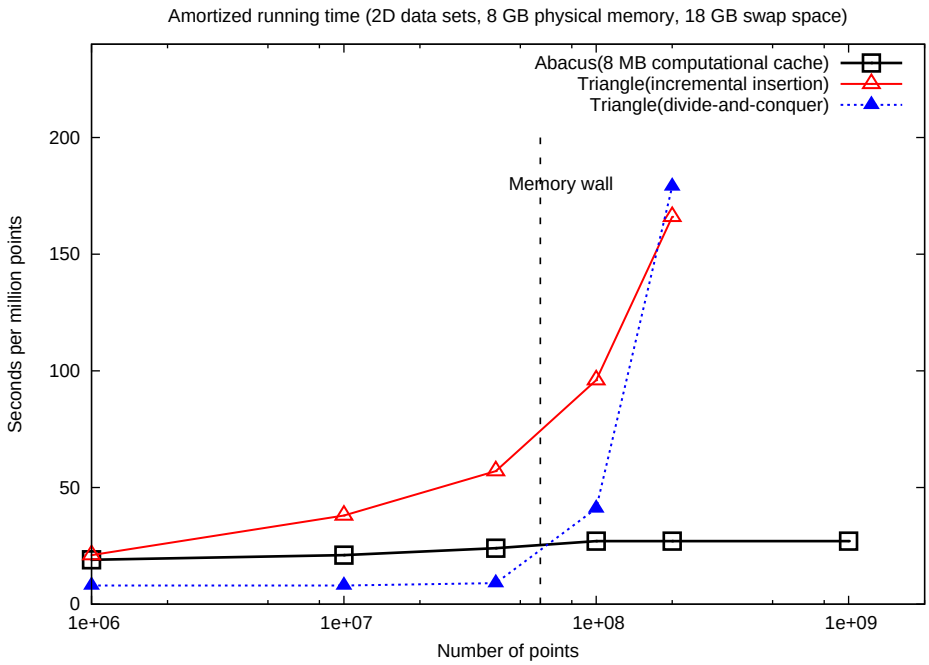


Figure 5.10: **The amortized running time for triangulating the 2D vertex sets.** The x-axis (in logarithmic scale) represents the number of vertices. The y-axis represents the average execution time in seconds for processing every 1 million vertices. The three curves represent the results of triangulating the same data sets using Abacus, Triangle executing the incremental insertion algorithm, and Triangle executing the divide-and-conquer algorithm, respectively.

Number of vertices	1.3 M	11.3 M	35.7 M	134 M	534.5 M	1.0 B
Number of tetrahedra	7 M	63.4 M	200 M	738 M	3.0 B	5.7 B
Abacus time	00:05:48	00:54:39	03:09:56	12:26:50	51:18:21	102:59:13
Staging time	00:00:01	00:00:16	00:00:48	00:02:54	00:11:21	00:22:19
Triangulation time	00:05:47	00:54:23	03:09:08	12:23:56	51:07:00	102:36:54
Abacus amortized time (s)	268	290	319	334	346	371
Pyramid	00:03:16	00:43:28	inf	N/A	N/A	N/A
Memory usage	493 MB	4.4 GB	14 GB(est.)	52 GB(est.)	0.2 TB(est.)	0.4 TB(est.)
Pyramid amortized time (s)	155	231	inf	N/A	N/A	N/A

Figure 5.11: **Running time comparison between Abacus and Pyramid.** "TB" stands for terabyte.

Figure 5.10 plots the amortized running times for triangulating 1 million vertices on average by the three programs. The x-axis is in logarithmic scale and represents the number of vertices. The y-axis is in normal scale and shows the amortized running time for triangulating every 1 million points. The dashed vertical bar in the figure marks the size of the data set that would use us all the 8 GB physical memory.

The 3D cases

Pyramid, the 3D cousin of Triangle, only implemented the incremental insertion algorithm. Hence, the performance comparison shown below does not include the divide-and-conquer case.

Figure 5.11 shows the running time of triangulating 3D data sets using Abacus and Pyramid, respectively. The meaning of the rows is identical to that in Figure 5.9.

Figure 5.12 shows the comparison of the amortized time for processing 1 million vertices on average between Abacus and Pyramid. The vertical bar in the figure shows the memory wall (8 GB physical memory) which Pyramid failed to cross.

Pyramid started to thrash severely when triangulating the 35.7 million vertex set. The CPU utilization of the program dropped below 0.1% (observed using the `top` program). Since the program did not finish in a month, we set the running time as infinity in the table.

The estimated memory usage shown in the table is based on linear extrapolation of the memory usage by smaller data sets. Since pyramid used about 390 MB for every one million vertices, the 1 1 billion vertex set requires about 400 GB memory. Even assuming the memory size keeps growing according to Moore's Law, that is, doubling every 18 months, it will take about 4 years for the memory of our server (currently 8 GB) to reach the size that can accommodate the data set. In comparison, it took Abacus 4 days and 6 hours.

Main Conclusions

The first main conclusion we draw from the comparison study is that Abacus is a competitive solution when triangulating data sets that fit in memory. Its performance matches that of the best incore Delaunay triangulators.

The second main conclusion is that Abacus is a superior solution than Triangle and Pyramid

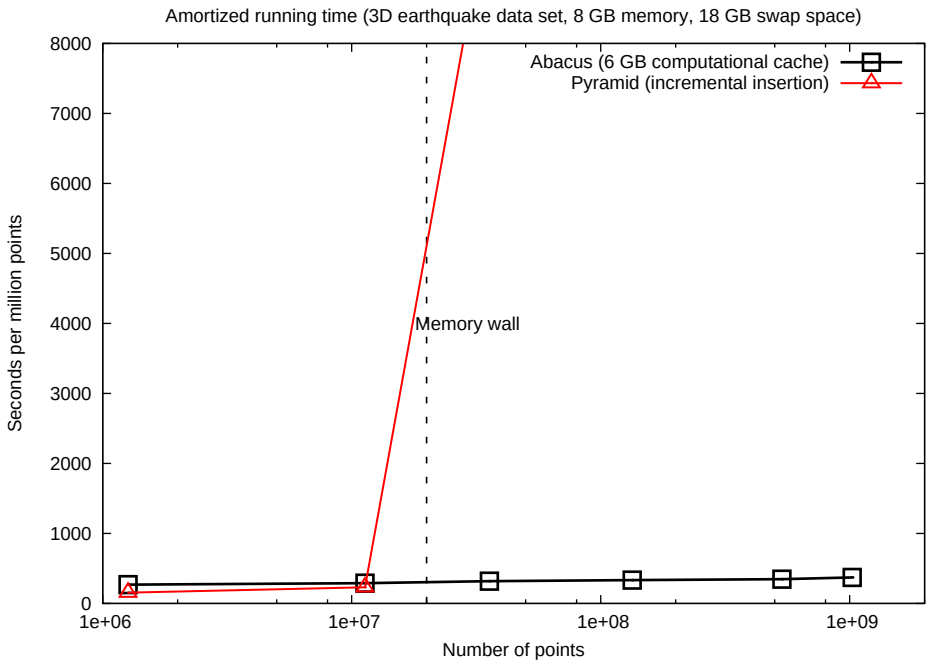


Figure 5.12: **The amortized running time for triangulating the 3D vertex sets.** The x-axis (in logarithmic scale) represents the number of vertices. The y-axis represents the average execution time in seconds for processing every 1 million vertices. The two curves represent the results of triangulating the same data sets using Abacus and Pyramid, respectively.

when triangulating data sets larger than the memory size. When Triangle and Pyramid start thrashing (out of physical memory) and stop working (out of virtual memory), Abacus continues to deliver scalable performance.

As a final note of this section, we must clarify that Triangle and Pyramid were designed and implemented as highly efficient *incore* quality-guaranteed Delaunay mesh generators. They are capable of powerful functions such as Delaunay refinement mesh generation, constrained Delaunay triangulation, and convex hull construction, which we do not support in Abacus. It is an interesting research topic to investigate how to incorporate these algorithms into Abacus in the future.

5.4 The Effect of the Physical Memory Size

The server we used to conduct the previous experiments has 8 GB physical memory, which is the norm today. In order to understand the effect of the physical memory size on the performance of Abacus, we configured the kernel to boot with only 64 MB physical memory (and a 2 GB swap space) and re-conducted the experiments for the 2D cases.

After the machine was rebooted, there were only about 20 MB free physical memory. We set the size of the Abacus computational cache to 8 MB, as we did for the previous 2D experiments. The operating system did not make aggressive use of the remaining 12 MB memory and left them unused all the time. Hence, in this configuration, we completely skipped the file system buffer cache and practically used only 8 MB physical memory.

Figure 5.13 shows the running time of triangulating the 2D data sets in the setup just described. The database sizes are the same as listed in previous sections. They are listed as a quick reference.

Number of vertices	1 M	10 M	40 M	100 M	200 M	1 B
Number of triangles	2M	20 M	80 M	200 M	400 M	2 B
Total database size	92 MB	1 GB	4 GB	11 GB	21 GB	94 GB
Triangle database	67 MB	817 MB	3.1 GB	8.3 GB	16.4 GB	69.0 GB
Vertex database	25 MB	246 MB	981 MB	2.5 GB	4.90 G	24.5 GB
Abacus time	00:00:29	00:06:11	00:32:01	01:33:01	02:37:50	14:49:05
Staging time	00:00:02	00:00:23	00:01:40	00:05:09	00:10:26	00:52:14
Triangulating time	00:00:27	00:05:48	00:30:22	01:27:52	02:27:24	13:56:51
Abacus amortized time (s)	27	35	46	53	47	53
Slowdown (vs. 8 GB memory)	1.4	1.9	2.0	2.1	2.0	2.0

Figure 5.13: **The running time of Delaunay triangulating the 2D vertex sets (64 MB physical memory, 8 MB Abacus computational cache).**

As the table shows, we have successfully constructed a Delaunay triangulation for a data set (1 billion vertices, 94 GB database size) that is 4 orders of magnitude larger than the memory size (the 8 MB Abacus cache).

What's more surprising is that there is only about a factor of 2 slowdown when we compare

the performance to the cases when 8 GB physical memory was used. The side-by-side comparison of the amortized running times are shown Figure 5.14.

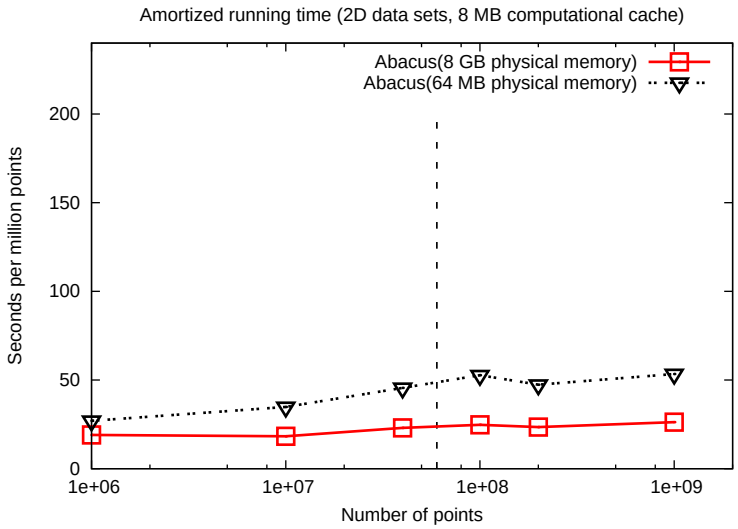


Figure 5.14: **Comparison between the Abacus amortized running times between the 8 GB memory setup and the 64 MB memory setup.** Both cases use 8 MB computational cache.

5.5 I/O Characteristics

In order to understand why the experiments with only 64 MB memory footprint did not suffer from I/O thrashing, we instrumented our code to collect the I/O traces for accessing the vertex table and the triangle table, and re-ran the experiment for triangulating 100 million vertices.

5.5.1 I/O Requests to the Vertex Table

Figure 5.15 shows the read I/O trace for accessing the vertex table. Since the vertex database is read-only during the construction of a triangulation, there is no write I/O trace. The x-axis in the figure shows the execution time in microsecond and the y-axis shows the vertex data page ids. A dot at `(x, y)` represents a read request at time `x` for page `y`.

We can see from the I/O trace plot that there are random reads are scarce. Most of the read requests are clustered along two major threads, one at the top of the plot (the top thread) and the other diagonally across the plot (the diagonal thread).

The top thread represents read accesses to the redundant vertices inserted in earlier rounds. Recall from Chapter 4 that the redundant vertices in earlier rounds are stored in vertex data pages with larger page ids because we bulk-load the vertex database in a backward order. Therefore,

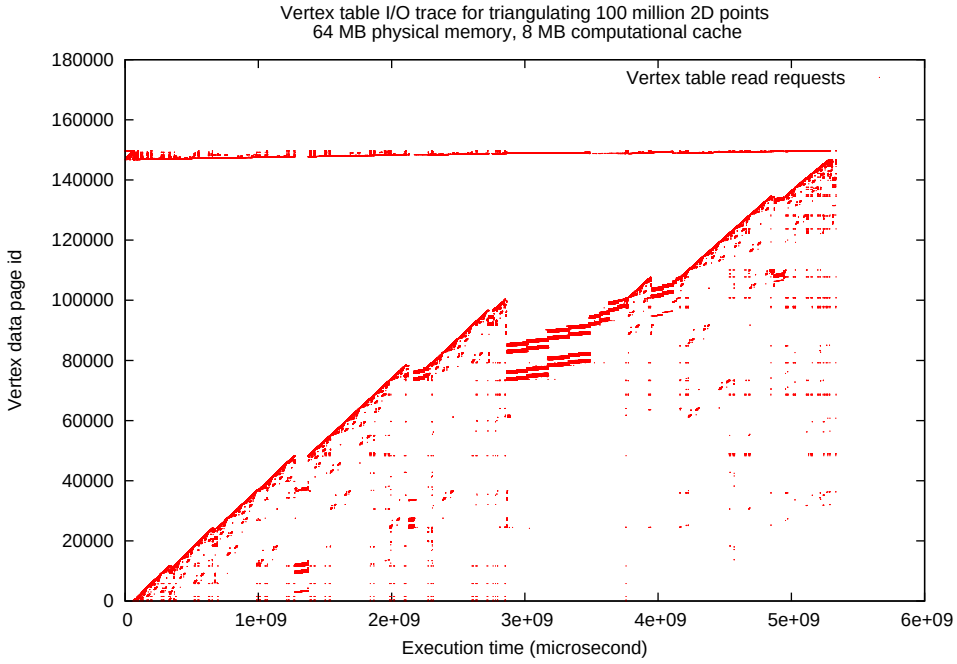


Figure 5.15: **I/O trace for reading the vertex table.** The vertex table is read-only when we triangulate the vertices.

the top thread starts at a large page id on the y-axis. Meanwhile, the redundant vertices all together account for a little more than 2% of the total data pages (since we have set the promotion probability to 0.02.) Thus, the top thread is almost flat.

The reason the redundant vertices are accessed throughout the execution time is that they are gradually being replaced by the (original) vertices inserted in the last round. When a vertex in the last round has a redundant counterpart in an earlier round, the triangles incident to the redundant counterpart will be deleted. While doing so, we need to load the triangles and their vertices into the Abacus cache. One of the vertices must be the redundant counterpart. In most cases, the redundant vertex is already flushed out of the computational cache since the access time between the initial insertion of the redundant vertex and the time it is replaced is usually quite long. This is the reason why there are read requests to access redundant vertices late in the execution time.

The diagonal thread represents read accesses to the original vertices. The clustering shows the effect of the spatial locality in the data sets. When a vertex is inserted into the Delaunay triangulation, the triangles that need to be deleted due to its occurrence usually consist of vertices that have close-by vertex ids. If these vertices have been flushed out of the computational cache, read requests are issued to fetch them back in. As a result, the read requests for vertex page ids tend to be clustered.

5.5.2 I/O Requests to the Triangle Table

Figure 5.16 shows both the read and write I/O traces for accessing the simplex table. The read trace is shown in the upper half of the figure. A dot at `(x, y)`, where $y > 0$, represents a read

request at time x for page y. The write trace is shown in the lower half of the figure. A dot at (x, y), where $y < 0$, represents a write request at time x for page (-y).

Figure 5.16: **I/O trace for accessing the simplex table.** The upper half of the figure shows the read requests; the lower half shows the write requests.

The figure is almost symmetric for dual reasons. On the one hand, a new triangle page that is written out to disk is very likely to be fetched into the Abacus cache shortly since the triangles on the page needs to be removed due to the spatially locality among the insertion vertices. On the other hand, if a triangle page recently read is modified, the page becomes deallocated immediately. The Abacus system reuses the page, stores other triangles on the page and writes it to disk. Therefore, the symmetry is a natural outcome by design.

Both the read and write traces show two main I/O threads, one close to the x-axis and the other diagonal. They correspond to the top I/O thread and the diagonal thread shown in Figure 5.15, respectively. Triangles created due to vertices inserted in the earlier rounds are allocated from the beginning of the triangle page id space. There are only about 2% of these triangles since there are only about 2% of redundnat vertices. Hence, the triangle page id range they occupy is very narrow. Since these triangles are gradually deleted as the original vertices are inserted, the read requests to access these triangles are spread across the entire execution time in a narrow range. Therefore, the read and write I/O threads associated with these triangles are very close to the x-axis.

The diagonal threads associated with the triangles created and deleted due to the insertions of the original vertices. For the same reason as explained for the vertex I/O trace, the triangles recently created are likely to be deleted due to a new insertion. Hence, the read and write accesses in the triangle page id space are also clustered.

The basic conclusion we draw from the I/O trace study is that the design schemes of exploiting locality at both the application level (the modified BRIO order) and the system level (the Abacus cache) have resulted in good I/O behavior. Instead of thrashing, the storage system seems to have identified the small number of I/O threads and have been able to service the I/O requests satisfactorily. The technical challenge of how to tune the storage system to improve the performance is beyond the scope of this thesis and is left as future work.

5.6 Abacus Cache Performance

The running time presented in the previous sections suggest that most of the operations on the Delaunay triangulations have taken place with the Abacus cache.

Figure 5.17 shows the performance of the Abacus cache. performance when triangulating the 2D data sets. The size of the physical memory for these experiments was reset to 8 GB and the size of the Abacus cache was still 8 MB. The performance of the Abacus cache is characterized by "Abacus cache hit ratio". An *Abacus cache hit* means that when we walk from a triangle to one of its neighbors in a certain direction (i.e., across a particular edge), the neighbor is already stored in the Abacus cache. The Abacus cache hit ratio is the ratio between the total number of Abacus cache hits and the total number of walk attempts. To service an Abacus cache miss, we need to fetch a triangle data page from disk using the various probing techniques described in Section 4.10. Hence, the higher the Abacus cache hit ratio, the better the overall performance.

The table also shows the performance of the application cache, a secondary cache implemented to speed up the cavity creation step of the incremental insertion algorithm as described in Section 4.12.4. An incircle cache miss causes a traversal down the Abacus octree structure to find the starting triangle to expand a cavity. Since the octree structure is in the main memory, the cost of servicing an incircle cache miss is orders of magnitude less than that of servicing an Abacus cache miss.

Number of vertices	1 M	10 M	40 M	100 M	200 M	1 B
Total walk attempts	11 M	114 M	514 M	1.1 B	2.6 B	11.2 B
Abacus cache hit ratio	99.9948%	99.9911%	99.9937%	99.9893%	99.9935%	99.9938%
Incircle cache hit ratio	81.88%	81.95%	82.18%	82.68%	83.53%	89.82%

Figure 5.17: **Abacus cache performance for triangulating the 2D data sets (8 GB physical memory, 8 MB Abacus cache).**

Figure 5.18 shows the performance of the Abacus cache performance when triangulating the 3D data sets. The meaning of the rows is the same as that in Figure 5.17. Since the data set with 1.3 million vertices fit completely in the Abacus cache, the Abacus cache hit ratio is 100%.

These statistics explain why Abacus has been able to compete with the best incore algorithms: Abacus computes as if it *were* an incore algorithm. The Abacus cache hit ratios are above 99.99% and 99.999% for the 2D and 3D cases, respectively. Almost all data are accessed directly from the main memory. Further, unlike a traditional database where main memory cached data are retrieved by searching, Abacus retrieves a cached triangle via one memory reference. Therefore, Abacus can compute as fast as an incore algorithm.

Number of vertices	1.3 M	11.3 M	35.7 M	134 M	534.5 M	1.0 B
Total walk attempts	118 M	1.1 B	3.3 B	12.4 B	50.0 B	97.1 B
Abacus cache hit ratio	100%	99.99958%	99.99937 %	99.99913%	99.99920%	99.99915%
Incircle cache hit ratio	92.28%	92.29%	92.34%	92.18%	92.32%	92.41%

Figure 5.18: **Abacus cache performance for triangulating the 3D data sets (8 GB physical memory, 6 GB Abacus cache).**

In addition, the application cache also turns out to be an effective secondary optimization technique. In 2D, more than 4 out of 5 search of the application cache result in an incircle cache hit; and in 3D, more than 9 out of 10.

5.7 The Translation Mechanism

The translation mechanism is triggered when an Abacus cache miss occurs. The three probing techniques described in Section 4.10 are invoked in combination to load the missing triangles.

Figure 5.19 summarizes the characteristics of the probing operations for triangulating the 2D data sets. "Abacus cache misses" represents the number of times we cannot walk directly to a neighbor. Each row below represents the percentage of Abacus cache misses that is serviced by a particular probing technique. For example, the quick probe successfully retrieves 67.79% of the 596 missing triangles for triangulating the 1 M vertex data sets.

Number of vertices	1 M	10 M	40 M	100 M	200 M	1 B
Abacus cache misses	596	10118	32647	120520	168262	694189
Quick probe	67.79%	69.16 %	68.06%	69.32%	69.23%	62.83%
Ray-casting probe	31.54%	29.74%	30.81%	29.80%	29.80%	36.01%
Concentric probe	0.67%	1.10%	1.13%	0.88%	0.97%	1.16%

Figure 5.19: **Characteristics of the probing operations for triangulating the 2D data sets (8 GB physical memory, 8 MB Abacus cache).**

The statistics in the table illustrate the efficiency of our probing strategy. About 2/3 of the Abacus cache misses are serviced by quick probes, which involve only one disk access per probe. The speculative ray-casting probe retrieves about 1/3 of the missing triangles. The most expensive concentric probe accounts for about 1% of the misses.

Figure 5.20 summarizes the characteristics of the probing operations for triangulating the 3D data sets. The meanings of the rows are the same as in Figure 5.19.

The 3D cases have similar characteristics as the 2D cases. Together, the speculative quick probe and ray-casting probe resolve more than 92% of the misses. The concentric probe serves the remaining 7%. Considering that the Abacus cache miss in 3D is below 0.001%, the slightly higher number of concentric probes has no material effect on the overall performance. After all, it only accounts for less than 0.00007% of the total walk attempts.

Besides the I/O probing into the databases, we are also interested in understanding what is the CPU overhead of conducting data translation. However, since different components work in-

Number of vertices	1.3 M	11.3 M	35.7 M	134 M	534.5 M	1.0 B
Abacus cache misses	0	4459	21512	108424	398532	821445
Quick probe	N/A	50.93%	52.50%	53.40%	51.91%	52.13%
Ray-casting probe	N/A	42.52%	40.75%	39.36%	40.79%	40.50%
Concentric probe	N/A	6.55%	6.75%	7.24%	7.29%	7.37%

Figure 5.20: **Characteristics of the probing operations for triangulating the 3D data sets (8 GB physical memory, 6 GB Abacus cache).**

timately within the Abacus system to carry out data translation, it is difficult to precisely quantify the CPU cost.

Recall that there are two hash tables within the Abacus cache: the vertex hash table and the edge hash table. Whenever vertices or triangles are loaded into or flushed out of the Abacus cache, these two hash tables are looked up and operated on. Hence, as a first order approximation, we use the time spent in hash table operations as an indicator of the cost of data translation.

In the next experiment, we re-compiled the Abacus system and the Delaunay triangulation program with the `-g -pg` flags and re-ran the program triangulating the 2D vertex data set with 40 million vertices. The physical memory used for the experiment was 8 GB, and the Abacus cache size was set to 16 MB. Figure 5.21 lists the top 10 time-consuming functions as reported by `gprof`.

1	`hash_uint32key`:	24.63%
2	`hash_insert`:	11.95%
3	`hash_search`:	10.10%
4	`incircle`:	8.77%
5	`delaunay2d_insert`:	5.70%
6	`release_triangle`:	4.93%
7	`direct_insert_triangle`:	4.84%
8	`octree_search`:	4.51%
9	`get_leaf`:	2.40%
10	`moma_newobj`:	2.10%

Figure 5.21: **The top 10 time-consuming functions for triangulating 40 million 2D vertices (8 GB physical memory, 16 MB Abacus cache).**

The `hash_uint32key` function is permutes (scrambles) an input key to ensure that all the bits are used in a hash key. The `hash_insert` and `hash_search` functions are standard hash table operations. The `incircle` function is a robust geometry predicate that tests whether a vertex falls within the circumcircle of a triangle. The remaining functions are various internal functions used by Abacus to carry out the basic operations.

The aggregated cost of hash related functions accounts for about 47% of the total running time. In order words, we spend about half of the CPU time conducting data translation. In return, we achieve sustainable high performance for dealing with massive Delaunay triangulations.

In summary, the performance evaluation presented in this chapter should disperse any doubt

about the viability of a computational cache. However, we do not claim that the performance can be readily replicated for other scientific data sets and applications. But at the least, our results have unambiguously demonstrated the feasibility of the proposed database approach to computing large-scale Delaunay triangulations.

Chapter 6

Conclusion

The research presented in this dissertation is—according to our knowledge—the first of its kind. We have demonstrated how to integrate a set of new techniques with existing database technologies to compute large-scale Delaunay triangulations.

Nevertheless, how to extend and use existing database techniques to process other massive unstructured scientific data sets remains a difficult task in general. In order to *qualitatively* transform our overall ability to manipulate and interpret large-scale scientific data sets, significant research challenges must be overcome; and opportunities abound.

This concluding chapter summarizes the main contributions of the dissertation, outlines the road map for future work, and presents a final remark on the thesis.

6.1 Contributions

The development of the proposed database approach and the Abacus system has not taken place in a vacuum. From the conception of the idea to the design and implementation of the software system, we have been driven by the needs of real-world scientific data sets and applications every step of the way. Hence, our research results directly benefit scientific applications that operate on massive data sets.

Intellectual Contributions

Compared with previous research works, which targeted efficient query processing of *static* scientific data sets at the SQL level [62, 63, 64] or at the index level [88], this dissertation focuses on how to extend existing database techniques with new methods and algorithms to create opportunities for *dynamic* scientific applications to run directly on databases.

The intellectual contribution of the research consists of two parts:

- The identification of the structural mismatch between the unstructured representation of scientific data sets and tabular data layout in traditional database systems
- The proposition of an application-specific computational cache as a generic solution

Using Delaunay triangulation as a running example, we have demonstrated that *structural mismatch*, the discrepancy between how data are stored and indexed in databases and how data are accessed and manipulated by scientific applications, is the root cause for the difficulty of mapping core scientific applications to database systems.

We solve the problem by introducing an additional layer of indirection by adding an application-specific computational cache on top of the standard database page buffer pool, and carrying out runtime data translation between the unstructured representation of a scientific data set and its tabular data storage format in database pages.

Practical Contributions

From a practitioner's perspective, our research makes two contributions:

- The illustration of a systemic methodology of designing and implementing the Abacus computational cache and the associated translation mechanism
- The creation of a breakthrough new capability for constructing massive Delaunay triangulations on commodity servers

We have illustrated the complete procedure of how to organize triangulation data in a database, how to build and maintain a pointer-based topological structure in the Abacus cache, and how to map data between the database and the Abacus cache. Scientific data sets and applications of different natures may require different data organizations throughout the memory hierarchy. But the basic design and implementation methodology should the same as what we have described in Chapter 4.

The first application of the Abacus system is the new Delaunay triangulation program presented in Section 4.12.5. As shown in Chapter 5, the performance and scalability of the Abacus-enabled Delaunay triangulator are unparalleled. It creates a new capability for scientists to

construct massive 2D/3D Delaunay triangulations on their servers or even desktops that have sufficient disk storage space, deferring the time when scientists have to submit batch jobs to supercomputers that have hundreds of gigabytes to terabyte physical memories.

6.2 Future Work

Even though the Abacus prototype has shed light on the promise of the proposed database approach, there is still a long way to go to turn our vision into reality. The research challenges lying ahead spread across almost the entire spectrum of computer science, ranging from scientific computing to discrete algorithms to programming language to systems design to parallel architectures. Below we outline the plan for extending our current work to ultimately support the data-centric framework envisioned in Section 1.3.

Short Term Plan

The Neuse River Basin data set mentioned in in Chapter 5 was a data set collected by the North Carolina Floodplain Mapping Project. The project, started after Hurricane Floyd in 1999, was the first in the country to use the LIDAR technology (Light Detection and Ranging, an airborne laser scanning technology) to capture terrain data (i.e., a 2D vertex set) of the entire state. The goal was to analyze the elevation distribution of the terrain data to access flood risks, set insurance premiums, and create disaster plans. However, the sheer volume enormity of the 2D vertex set (500 million vertices for the Neuse River Basin alone) has rendered incore Delaunay triangulation programs useless, delaying the project's completion time from the projected 2002 to 2007 [68].

Isenburg and colleagues have devised a highly sophisticated streaming Delaunay triangulation algorithm to solve the problem [45]. The idea is to pre-process the input vertices by sequential-scanning the data set multiple times. In the final pass, a Delaunay triangulation is constructed and output on the fly to either disk storage or another streaming algorithm, for example, a streaming iso-contour extraction program.

The streaming-algorithm–based solution, though efficient, have two major limitations. First, when more vertices in an already triangulated region become available, the Delaunay triangulation streaming algorithm have to be re-run. Second, if a user such as a county emergency management officer needs to examine a specific set of iso-contours within her county, she has no choice but to re-run the the streaming visualization program on the entire Delaunay triangulation data set, even though she may be only concerned about 2% of the data.

We plan to extend Abacus to solve both problems and make it useful to a large number of other applications. In particular, our short-term research agenda is to develop Abacus into a full-fledged high-performance database system for dealing with Delaunay triangulation data sets.

- *Developing specialized read-only Abacus API functions.* The specialized API functions `abacus_stage_vertices` and `abacus_delauanay_insert` are both designed for support dynamic construction of Delaunay triangulations. A useful extension of the Abacus API is to develop a set of commonly used read-only functions such as range query,

iso-contour extraction, and ray-casting visualization. The facility to support these functions is already built within the Abacus system. In essence, all these operations can be efficiently supported by the walk operation described in Section 4.11.3. The only difference lies in which direction to walk to from within a particular triangle. For example, a range query walks to all directions (i.e., breadth-first expansion); a ray-casting algorithm walks along the direction of a ray; and an iso-contour extraction algorithm walks across the edges where the iso-contour values lie.

- *Porting the Abacus prototype to full-fledged database systems.* We have chosen to build Abacus as a standalone system to simplify software development and performance evaluation. On the downside, we cannot capitalize on the existing capabilities of a full-fledged database system. In particular, there is no support for associative search. For example, if a user queries for all the vertices with a certain attribute value (e.g., the x component of the velocity falls within a specified range), we must scan the vertex table to retrieve the qualified vertex records.

  Porting the Abacus prototype to an open-source database system such as PostgreSQL will create a full-fledged database system for dealing with Delaunay triangulations data sets. We will be able to provide the efficient data manipulation capabilities of the Abacus system as well as the associative query capabilities of a standard database system.

  Since our current implementation does not rely on any special features of the index structures, the mapping of software modules will be relatively straightforward. However, coupling Abacus data page operations with the transactional aspect of a standard database system will be a major problem. In our current design and implementation, we do not take the ACID property [33, 44] of a database system into consideration and focus only on the performance. Although we believe the ACID property is far less important to scientific applications than to business application, we must nevertheless thoroughly investigate the issue in order to carry out the porting.

- *Leveraging multi-core processors to speed up data translation.* As shown in Chapter 4, Delaunay triangulation on Abacus is a compute-intensive task, with a CPU utilization rate above 90%. In the meantime, roughly 47% of the CPU time (in 2D) is spent in data translation (i.e., hash table operations). An interesting research topic is to make use of the extra cores on a CPU to hide the translation overhead and further improve the performance of Abacus.

  Since data translation take place when triangles are loaded from or stored into a database page, we can decouple data translation from the normal execution of the triangulation code by prefetching the page or deferring the write-out. Thus, we will be able to make full use of the additional cycles provided by the extra cores. The technical challenge is how to share the Abacus computational cache among multiple cores.

Long Term Plan

- *Building a generic computational database framework.* The Abacus system is tailored to deal with one particular data set type: 2D/3D triangulations. There are many other commonly used scientific data set types. Building one special-purpose database system

for each data set type is certainly a possible solution. But we plan to develop a generic database framework to allow domain experts to define and implement their own computational cache. As mentioned in Section 3.1.2, a user-defined computational cache (UDC) needs to interact more intimately with the inner working of a database system. The challenge lies in how to export the right level of abstraction to support UDC extensions.

A related technical problem is how to provide a high-level declarative language (not necessarily an extension of the SQL) for user applications to interact with scientific data sets stored in a database system. In our current implementation, we have largely ignored this problem. But in a fully-functional database system, there should be one simple, coherent data manipulation language. Application programmers should not call the internal API functions as those exported by Abacus. Instead, all the interaction with the database should be expressed in the same high-level language construct.

- *Coupling with parallel scientific simulations.* Beyond all new the capabilities brought about by the proposed database approach, our ultimate goal is to develop a parallel database infrastructure to support terascale/petascale scientific simulations. All the simulation components will fetch data from or store data into a parallel distributed, computational database system. The physical storage space of the computational database will provided by a parallel file system such as Lustre.

  Going parallel adds a whole new dimension to the design space of the proposed database approach. How to enforce cache-coherence among the computational cache on different processors resembles the classic cache-coherence problem on shared-memory parallel computers. But the the granularity of sharing and the mechanism of enforcing coherence are different. Instead of relying on hardware instrument to enforce coherence at the cache-line level, we will need to use software protocol to enforce coherence at the application-logic level. Besides the cache-coherence problem, there are many other technical challenges. For example, how should we aggregate and schedule I/O to maximize the performance of the parallel file system? What kind of parallel database transaction semantic should be supported to facilitate application development?

All in all, these are exciting new research territories for us to explore. Only after we carry out the research can we materialize the full potential of a computational database system.

6.3 A Final Remark

The analysis, design, implementation, and evaluation presented in the previous chapters, along with the prospect of future work discussed in this chapter, jointly corroborate the following thesis statement:

> Extending existing database techniques with an application-specific computational cache is a scalable solution to computing large-scale Delaunay triangulations.

As we move towards the age of petascale computing, the data produced by scientific simulations and instruments will become more massive and unstructured. The data sets of tomorrow will dwarf the scale and complexity of the most unmanageable ones of today. In order to manage the extraordinary new data sets, we need extraordinary new capabilities.

Bibliography

[1] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H. G. Molina, D. Gawlick, J. Gray, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, M. Kersten, M. Pazzani, M. Lesk, D. Maier, J. Naughton, H. Schek, T. Sellis, A. Silberschatz, M. Stonebraker, R. Snodgrass, J. Ullman, G. Weikum, J. Widom, and S. Zdonik. The Lowell database research self-assessment. *Communications of the ACM*, 48(5):111–118, May 2005. 1

[2] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3), 2002. 3.4.1, 3.4.1

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of International Conference on Very Large Data Bases*, pages 169–180, 2001. 3.4.1, 3.4.1

[4] A. Ailamaki, Y. E. Ioannidis, and M. Livny. Scientific workflow management by database management. In *10th International Conference on Scientific and Statistical Database Management*, pages 190–199, Capri, Italy, 1998. 1.2.3

[5] V. Akcelik, J. Bielak, G. Biros, I. Ipanomeritakis, Antonio Fernandez, O. Ghattas, E. Kim, J. López, D. R. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terasacale computers. In *SC2003*, Phoenix, AZ, November 2003. 1.1

[6] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, pages 211–219, 2003. 4.12.1

[7] F. Bancilhon and D. Maier. Multilanguage object oriented systems: New answer to old database problems. In K. Fuchi and L. Kott, editors, *PROGRAMMING OF FUTURE GENERATION COMPUTERS II (Proceedings of the Second Franco-Japanese Symposium, Cannes, France, 9-11 November 1987)*. North-Holland, 1988. 2, 2.3

[8] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. O'Hallaron, J. Shewchuk, and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 152:85–102, January 1998. 1.1

[9] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. R. O'Hallaron, J. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Supercomputing '96*, Pittsburgh, PA, November 1996. 1.1

[10] N. Beckmann, H-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and

robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322 – 331, Atlantic City, NJ. 2.3.1, 3.1.1

[11] J. Belca and D. L. Wang. Lessons learned from managing a petabyte. In *Proceedings of the 2005 Conference on Innovative Data Systems*, pages 70–83, Asilomar, CA, January 2005. 1.1, 1.2.3

[12] J. Bielak, L. Kallivokas, J. Xu, and R. Monopoli. Finite element absorbing boundary for the wave equation in a halfplane with an application to engineering seismology. In *Proceedings of the Third International Conference on Mathematical and Numerical Aspects of Wave Propogation*, pages 489–498, Mandelieu-la-Napule, France, April 1995. INRIA-SIAM. 1.1

[13] A. Bowyer. Computing dirichlet tessellations. *Computer Journal*, 24(2):162–166, 1981. 2.2

[14] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proceedings of the fourteenth annual symposium on computational geometry*, pages 165–174, 1998. 2.3.1

[15] R. E. Bryant. Data-intensive supercomputing: The case for DISC. Technical Report CMU-CS-07-128, Carnegie Mellon School of Computer Science, 2007. 1.3

[16] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*, chapter 6: The Memory Hierarchy. Prentice Hall, 2003. 3.2.1

[17] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proceedings of International Conference on Data Engineering*, 2004. 3.4.1

[18] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 2001. 3.4.1

[19] H. Childs, M. Duchaineau, and K.-L. Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, Lisbon, Portugal, May 2006. 1.2.1

[20] H-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of International Conference on Very Large Data Bases*, pages 127–141, 1985. 3.1.1

[21] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, Jun 1979. 3.1.1

[22] C. J. Date. An architecturee for high-level language database extensions. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data*, pages 101–122, 1976. 3.4.2

[23] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*, chapter 9. Springer, 2 edition, 2000. 2.2

[24] B. N. Delaunay. Sur la Sphére Vide. *Izvestia Akademia Nauk SSSR, VII Seira, Otdelenie*

Matematicheskii i Estestvennyka Nauk, 7:793–800, 1934. 2.2

[25] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. In *Proceedings of the 7th Annual Symposium on Computational Geometry*, pages 106–114, 2001. 4.11.3, 4.11.3

[26] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2(2):137–151, 1987. 2.2

[27] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMID-SIGART Symposium on Principles of Database Systems (PODS)*, 1989. 1.2.2, 5.1.1

[28] Flexible Image Transport System. `http://fits.gsfc.nasa.gov`. 1.2.2

[29] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. On sorting triangles in a delaunay tesselation. *Algorithmica*, 6:522–532, 1991. 4.11.3

[30] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(2):153–174, 1987. 2.2

[31] I. Gargantini. An effecive way to represent quadtrees. *Communicatoins of the ACM*, 25 (12):905–910, Dec 1982. 1.2.2, 4.8

[32] J. Gray, D. T. Liu, M. Nieto-Santistenban, A. S. Szalay, D. DeWitt, and G. Heber. Scientific data management in the coming decade. Technical Report MSR-TR-2005-10, Microsoft Research, One Microsoft Way, Redmond, WA 98052, January 2005. 1, 1.2.3, 1.3, 2, 2.3

[33] J. Gray and A. Reuter. *Transcation Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, 1993. 2.3.2, 6.2

[34] J. Gray and A. Szalay. Where the rubber meets the sky: Bridging the gap between databases and science. Technical Report MSR-TR-2004-110, Microsoft Research, One Microsoft Way, Redmond, WA 98052, October 2004. 1.1, 1.1

[35] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985. 2.2, 4.4

[36] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984. 2.3.1, 3.1.1

[37] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *Proceedings of the First International Conference on Innovative Data Systems Research (CIDR)*, 2003. 3.4.1

[38] S. Harizopoulos and A. Ailamaki. Steps towards cache-resident transaction processing. In *Proceedings of the 30th Internatoinal Conference on Very Large Data Bases*, 2004. 3.4.1

[39] T. D. R. Hartley, U. Catalyurek, F. Özgüner, A. Yoo, S. Kohn, and K. Henderson. MSSG: A framework for massive scale semantic graphs. In *Proceedings of 2006 IEEE International Conference on Cluster Computing*, pages 1–10, 2006. 2.3.2

[40] HDF5. `http://hdf.ncsa.uiuc.edu/HDF5`. 1.2.2

[41] G. Heber and J. Gray. Supporting finite element analysis with a relational database back-end part I: There is life beyond files. Technical Report MSR-TR-2005-49, Microsoft Research, One Microsoft Way, Redmond, WA 98052, May 2005. 1.2.3, 2, 2.3

[42] G. Heber and J. Gray. Supporting finite element analysis with a relational database back-end part II: Database design and access. Technical Report MSR-TR-2006-21, Microsoft Research, One Microsoft Way, Redmond, WA 98052, March 2006. 1.2.3

[43] G. Heber, C. Pelkie, A. Dolgert, J. Gray, and D. Thompson. Supporting finite element analysis with a relational database backend; part III: Opendx—where the numbers come alive. Technical Report MSR-TR-2005-151, Microsoft Research, One Microsoft Way, Redmond, WA 98052, December 2005. 1.2.3

[44] J. M. Hellerstein and M. Stonebraker, editors. *Readings in Database Systems*, chapter 1. The MIT Press, Cambridge, MA, 4 edition, 2005. 1.2.3, 3, 3.4.2, 2, 6.2

[45] M. Isenburg, Y. Liu, J. R. Shewchuk, and J. Snoeyink. Streaming computation of delaunay triangulations. In *Proceedings of the 2006 SIGGRAPH*. ACM, August 2006. 4.12.1, 4.12.4, 6.2

[46] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 500–509, 1994. 2.3.1

[47] G. Karypis and V. Kumar. A course-grain parallel formulation of multi-level k-way graph partitioning algorithm. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997. 1.2.2

[48] T. L. Kay and J. T. Kayjiya. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer graphics and Interactive Techniques*. 4.10.2

[49] E. Kim, J. Bielak, and O. Ghattas. Large-scale northridge earthquake simluation using octree-based multiresolution mesh method. In *Proceedings of the 16th ASCE Engineering Mechanics Conference*, Seattle, Washington, July 2003. 1.2.2

[50] C. L. Lawson. *Software for C^1 Surface Interpolation. Mathematical Software III*, pages 161–194. Academic Press, New York, 1977. 2.2

[51] H. Yu (Technial lead), T. Tu (Team lead), J. Bielak, O. Ghattas, J. C. López, K-L Ma, D. R. O'Hallaron, L. Ramirez-Guzman, N. Stone, R. Taborda-Rios, and J. Urbanic. Remote runtime steering of integrated terascale simulation and visualization. In *SC2006*, Tampa, FL, November 2006. Analytics Challenge Winner. 1.2.1

[52] D-T. Lee and B. J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer and Information Sciences*, 9(3):219–242, 1980. 2.2

[53] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of International Conference on Very Large Data Bases*, pages 212–223, 1980. 3.1.1

[54] K-L. Ma and T. Crockett. Parallel visualization of large-scale aerodynamics calculations: A case study on the Cray T3E. In *Proceedings of 1999 IEEE Parallel Visualization and Graphics Symposium*, pages 15–20, San Francisco, CA, October 1999. 1.2.1

[55] K-L. Ma, A. Stompel, J. Bielak, O. Ghattas, and E. Kim. Visualizing large-scale earthquake simulations. In *SC2003*, Phoenix, AZ, November 2003. 1.2.1

[56] National Center for Biotechnology Information. `http://ncbi.nih.gov`. 1.1

[57] NetCDF. `http://www.unidata.ucar.edu/software/netcdf`. 1.2.2

[58] NIH Genbank. `http://www.ncbi.nlm.nih.gov/Genbank/index.html`. 1.2.3

[59] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 297–306, 1993. 3.1.1

[60] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of ACM SIGMOD*, pages 326–336, Washington D.C, 1986. 1.2.2

[61] J. A. Orenstein and T. H. Merrett. A class of data structure for associative searching. In *Proceedings of ACM SIGACT-SIGMOD*, pages 181–190, Waterloo,Ontario,Canada, 1984. 1.2.2

[62] S. Papadomanolakis and A. Ailamaki. AutoPart: Automated schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, Santorini Island, Greece, June 2004. 1.2.3, 6.1

[63] S. Papadomanolakis and A. Ailamaki. Workload-driven schema design for large scientific databases. *Bulletin of the Technical Committee on Data Engineering*, 27(4):21, 2004. 1.2.3, 6.1

[64] S. Papadomanolakis, A. Ailamaki, J. C. López, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of Data*, pages 551–562, 2006. 4.11.3, 6.1

[65] Petaflops. 1997 Petaflops algorithms workshop summary report. `http://www.hpcc.gov/pubs/pal97.html`, 1997. 2

[66] postgresql. `http://www.postgresql.org`. 2.4

[67] Qhull. `http://www.qhull.org`. 2.2

[68] M. Quillin. Flood plain maps better, but late—years late. Raleigh News & Observers, March 11 2002. 5.1.1, 6.2

[69] A. L. Rosenburg and L. Snyder. Time- and space-optimality in B-trees. *Transactions on Database Systems*, 6(1):174–183, March 1981. 3.1.1

[70] J. Salmon and M. S. Warren. Parallel out-of-core methods for N-body simulation. In *Proceedings of the 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997. 3.4.3

[71] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison-Wesley Publishing Company, 1990. 4.8

[72] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley

Publishing Company, 1990. 4.8

[73] J. W. Schmidt. Some high level language constructs for data of type relation. *Trasactions on Database Systems*, pages 247–261, 1977. 3.4.2

[74] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of SIGMOD Conference*, pages 23–34, 1979. 3.1.1

[75] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, 1987. 2.3.1, 3.1.1

[76] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling page layout from storage organization. In *Proceedings of the 30th VLDB Conference*, 2004. 3.4.1, 3.4.1

[77] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transcations on Database Systems*, 11(3):239–264, 1986. 3.1.1

[78] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 141–150, May 1996. 2.3.1

[79] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry. 4.4, 5

[80] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, 1997. 2.3.1

[81] J. R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1997. 2.1, 2.2, 5

[82] Sloan Digital Sky Survey. `http://www.sdss.org`. 1.2.3

[83] Southern California Earthquake Center Community Modeling Environment Data Products. `http://epicenter.usc.edu/cmeportal/products.html`. 1.2.2

[84] Stanford Linear Accelerator Center BaBar Detector. `http://www-public.slac.stanford.edu/babar`. 1.2.3

[85] N. T. B. Stone, D. Balog, B. Gill, B. Johanson, J. Marsteller, P. Nowoczynski, D. Porter, R. Reddy, J. R. Scott, D. Simmel, J. Sommerfield, K. Vargo, and C. Vizino. PDIO: High-performance remote file I/O for Portals enabled compute nodes. In *Proceedings of the 2006 Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2006. 1.2.1

[86] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981. 4.1

[87] M. Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of*

International Conference on Data Engineering, pages 262–269, 1986. 3.1.1

[88] A. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh. Technical Report MSR-TR-2005-123, Microsoft Research, One Microsoft Way, Redmond, WA 98052, August 2005. 1.2.3, 6.1

[89] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 451–462, Dallas, TX, May 2000. 1.1, 1.2.3

[90] Tetgen. `http://tetgen.berlios.de`. 2.2

[91] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 161–179. American Mathematical Society, 1999. 3.4.3

[92] Triangle. `http://www.cs.cmu.edu/~quake/triangle.html`. 2.2

[93] T. Tu, D. O'Hallaron, and J. C. López. The Etree library: A system for manipulating large octrees on disk. Technical Report CMU-CS-03-174, Carnegie Mellon School of Computer Science, July 2003. 1.2.2, 1.2.2, 4.8.1, 4.8.2, 5.1.1

[94] T. Tu and D. R. O'Hallaron. Balanced refinement of massive linear octrees. Technical Report CMU-CS-04-129, Carnegie Mellon School of Computer Science, April 2004. 1.2.2

[95] T. Tu and D. R. O'Hallaron. A computational database system for generating unstructured hexahedral meshes with billions of elements. In *SC2004*, Pittsburgh, PA, November 2004. 1.2.2

[96] T. Tu and D. R. O'Hallaron. Extracting hexahedral mesh structures from balanced linear octrees. In *Proceedings of the Thirteenth International Meshing Roundtable*, Williamsburgh, VA, September 2004. 1.2.2

[97] T. Tu, D. R. O'Hallaron, and O. Ghattas. Scalable parallel octree meshing for terascale applications. In *SC2005*, Seattle, WA, November 2005. 1.2.1, 5.1.2

[98] T. Tu, D. R. O'Hallaron, and J. C. López. Etree: A database-oriented method for generating large octree meshes. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 127–138, Ithaca, NY, Sep 2002. 1.2.2, 5.1.1

[99] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K-L Ma, and D. R. O'Hallaron. From mesh generation to scientific visualization—an end-to-end approach to parallel supercomputing. In *SC2006*, Tampa, FL, November 2006. 1.1, 1.2.1, 5.1.2

[100] S-K. Ueng, C. Sikorski, and K-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4): 370–380, 1997. 3.4.3

[101] United States Geological Survey San Francisco 1906 Earthquake Simulation Project Velocity Model. `http://www.sf06simulation.org/geology/velocitymodel`. 1.2.2

[102] J. Vitter. External memory algorithms and data structures: dealing with massive data.

ACM Computing Surveys, 33(2):209–271, 2001. 3.4.3

[103] D. F. Watson. Computing the n-dimensional Delaunay tesselation with applications to Voronoi polytopes. *Computer Journal*, 24(2):167–172, 1981. 2.2

[104] H. F. Yu, K-L. Ma, and J. Welling. A parallel visualization pipeline for terascale earthquake simulations. In *SC 2004*, Pittsburgh, PA, November 2004. 1.2.1

[105] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 1990. 2, 2.3, 3.4.2