JIM GRAY, MICROSOFT
MARK COMPTON, CONSULTANT

Long anticipated, the arrival of
radically restructured database
architectures is now finally at hand.

A Call To ARMS

We live in a time of extreme change, much of it precipitated by an avalanche of information that otherwise threatens to swallow us whole. Under the mounting onslaught, our traditional relational database constructs—always cumbersome at best—are now clearly at risk of collapsing altogether.

In fact, rarely do you find a DBMS anymore that doesn't make provisions for online analytic processing. Decision trees, Bayes nets, clustering, and time-series analysis have also become part of the standard package, with allowances for additional algorithms yet to come. Also, text, temporal, and spatial data access methods have been added—along with associated probabilistic logic, since a growing number of applications call for approximated results. Column stores, which store data column-wise rather than record-wise, have enjoyed a rebirth, mostly to accommodate sparse tables, as well as to optimize bandwidth.

Is it any wonder classic relational database architectures are slowly sagging to their knees?

But wait... there's more! A growing number of application developers believe XML and XQuery should be treated as our primary data structure and access pattern, respectively. At minimum, database systems will need to accommodate that perspective. Also, as external data increasingly arrives as streams to be compared with historical data, stream-processing operators are of necessity being added. Publish/subscribe systems contribute further to the challenge by inverting the traditional data/query ratios, requiring that incoming data be compared against

A Call To ARMS

millions of queries instead of queries being used to search through millions of records. Meanwhile, disk and memory capacities are growing significantly faster than corresponding capabilities for reducing latency and ensuring ample bandwidth. Accordingly, the modern database system increasingly depends on massive main memory and sequential disk access.

This all will require a new, much more dynamic query optimization strategy as we move forward. It will have to be a strategy that's readily adaptable to changing conditions and preferences. The option of cleaving to some static plan has simply become untenable. Also note that we'll need to account for intelligence increasingly migrating to the network periphery. Each disk and sensor will effectively be able to function as a competent database machine. As with all other database systems, each of these devices will also need to be self-managing, self-healing, and always-up.

No doubt, you're starting to get the drift here. As must be obvious by now, we truly have our work cut out for us to make all of this happen. That said, we don't have much of a choice—external forces are driving these changes.

WELCOME TO THE REVOLUTIONS

The rise, fall, and rise of object-relational databases.

We've enjoyed an extended period of relative stasis where our agenda has amounted to little more than "implement SQL better"—Michael Stonebraker, professor at UC Berkeley and developer of Ingres, calls this the "polishing-the-round-ball" era of database research and implementation. Well, friends, those days are over now because databases have become the vehicles of choice for delivering integrated application development environments.

That's to say that data and procedures are being joined at long last—albeit perhaps by a shotgun wedding. The Java or common language runtimes have been married to relational database engines so that the traditional EJB-SQL outside-inside split has been eliminated. Now Beans or business logic can run inside the database. Active databases are the result, and they offer tremendous

potential—both for good and for ill. (More about this as the discussion proceeds.) Our most immediate challenge, however, stems from traditional relational databases never being designed to allow for the commingling of data and algorithms.

The problem starts, of course, with Cobol, with its data division and procedure division—and the separate committees that were formed to define each. Conway's law was at work here: "Systems reflect the organizations that built them." So, the database community inherited the artificial data-program segregation from the Cobol DBTG (Database Task Group). In effect, databases were separated from their procedural twin at birth and have been struggling ever since to reunite—for nearly 40 years now.

The reunification process began in earnest back in the mid-1980s when stored procedures were added to SQL (with all humble gratitude here to those brave, hardy souls at Britton-Lee and Sybase). This was soon followed by a proliferation of object-relational databases. By the mid-1990s, even many of the most notable SQL vendors were adding objects to their own systems. Even though all of these efforts were fine in their own right—and despite a wave of over-exuberant claims of "object-oriented databases *uber alles*" by various industry wags—each one in turn proved to be doomed by the same fundamental flaw: the high risks inherently associated with all *de novo* language designs.

It also didn't help that most of the languages built into the early object-relational databases were absolutely dreadful. Happily, there now are several good object-oriented languages that are well implemented and offer excellent development environments. Java and C# are two good examples. Significantly, one of the signature characteristics of this most recent generation of OO environments is that they provide a common language runtime capable of supporting good performance for nearly all languages.

The really big news here is that these languages have also been fully integrated into the current crop of object-relational databases. The runtimes have actually been added to the database engines themselves such

that one can now write database *stored procedures* (modules), while at the same time defining database objects as *classes* within these languages. With data encapsulated in classes, you're suddenly able to actually program and debug SQL, using a persistent programming environment that looks reassuringly familiar since it's an extension of either Java or C#. And yes, pause just a moment to consider: SQL programming comes complete with version control, debugging, exception handling, and all the other functionality you would typically associate with a fully productive development environment. SQLJ, a nice integration of SQL and Java, is already available—and even better ideas are in the pipeline.

The beauty of this, of course, is that the whole inside-the-database/outside-the-database dichotomy that we've been wrestling with over these past 40 years is becoming a thing of the past. Now, fields are objects (values or references); records are vectors of objects (fields); and tables are sequences of record objects. Databases, in turn, are transforming into collections of tables. This objectified view of database systems gives tremendous leverage—enough to enable many of the other revolutions we're about to discuss. That's because, with this new perspective, we gain a powerful way to structure and modularize systems, especially the database systems themselves.

A clean, object-oriented programming model also increases the power and potential of database triggers, while at the same time making it much easier to construct and debug triggers. This can be a two-edged sword, however. As the database equivalent of rules-based programming, triggers are controversial—with plenty of detractors as well as proponents. Those who hate triggers and the active databases they enable will probably not be swayed by the argument that a better language foundation is now available. For those who are believers in active databases, however, it should be much easier to build systems.

None of this would even be possible were it not for the fact that database system architecture has been increasingly modularized and rationalized over the years. This inherent modularity now enables the integration of databases with language runtimes, along with all the various other ongoing revolutions that are set forth in the pages that follow—every last one of them implemented as extensions to the core data manager.

TPlite rides again. Databases are encapsulated by business logic, which—before the advent of stored procedures—always ran in the TP (transaction processing) monitor. It was situated right in the middle of classic triple-tier presentation/application/data architectures.

With the introduction of stored procedures, the TP

monitors themselves were disintermediated by two-tier client/server architectures. Then, the pendulum swung back as three-tier architectures returned to center stage with the emergence of Web servers and HTTP—in part to handle protocol conversion between HTTP and the database client/server protocols and to provide presentation services (HTML) right on the Web servers, but also to provide the execution environment for the EJB or COM business objects.

As e-commerce has continued to evolve, most Web clients have transformed such that today, in place of browsers blindly displaying whatever the server delivers, you tend to find client application programs—often in JavaScript—that have much of the presentation logic and that use XML to communicate with the server. Although most e-commerce clients continue to *screen-scrape* so as to extract data from Web pages, there is growing use of XML Web services as a way to deliver data to fat-client applications. Just so, even though most Web services today continue to be delivered by classic Web servers (Apache and Microsoft IIS, for example), database systems are starting to listen to port 80 and directly offer SOAP invocation interfaces. In this brave new world, you can take a class—or a stored procedure that has been implemented within the database system—and publish it on the Internet as a Web service (with the WSDL interface definition, DISCO discovery, UDDI registration, and SOAP call stubs all being generated automatically). So, the “TPlite” client/server model is making a comeback, if you want it.

Application developers still have the three-tier and *n*-tier design options available to them, but now two-tier is an option again. For many applications, the simplicity of the client/server approach is understandably attractive. Still, security concerns—bearing in mind that databases offer vast attack surfaces—will likely lead many designers to opt for three-tier server architectures that allow only Web servers in the demilitarized zone and database systems to be safely tucked away behind these Web servers on private networks.

Still, the mind spins with all the possibilities our suddenly broadened horizons seem to offer. For one thing, is it now possible—or even likely—that Web services will end up being the means by which we federate heterogeneous database systems? This is by no means certain, but it is intriguing enough to have spawned a considerable amount of research activity. Among the fundamental questions in play are: What is the right object model for a database? What is the right way to represent information on the wire? How do schemas work on the Internet? Accordingly, how might schemas be expected to evolve?

A Call To ARMS

How best to find data and/or databases over the Internet?

In all likelihood, you're starting to appreciate that the ride ahead is apt to get a bit bumpy. Strap on your seat-belt. You don't even know the half of it yet.

Making sense of workflows. Because the Internet is a loosely coupled federation of servers and clients, it's just a fact of life that clients will occasionally be disconnected. It's also a fact that they must be able to continue functioning throughout these interruptions. This suggests that, rather than tightly coupled, RPC-based applications, software for the Internet must be constructed as asynchronous tasks; they must be structured as workflows enabled by multiple autonomous agents. To get a better feel for these design issues, think e-mail, where users expect to be able to read and send mail even when they're not connected to the network.

All major database systems now incorporate queuing mechanisms that make it easy to define queues, to queue and de-queue messages, to attach triggers to queues, and to dispatch the tasks that the queues are responsible for driving. Also, with the addition of good programming environments to database systems, it's now much easier and more natural to make liberal use of queues. The ability to publish queues as Web services is just another fairly obvious advantage. But we find ourselves facing some more contentious matters because—with all these new capabilities—queues inevitably are being used for more than simple ACID (atomic, consistent, isolated, durable) transactions. Most particularly, the tendency is to implement publish/subscribe and workflow systems on top of the basic queuing system. Ideas about how best to handle workflows and notifications are still controversial—and the focus of ongoing experimentation.

The key question facing researchers is how to *structure* workflows. Frankly, a general solution to this problem has eluded us for several decades. Because of the current immediacy of the problem, however, we can expect to see plenty of solutions in the near future. Out of all that, some clear design patterns are sure to emerge, which should then lead us to the research challenge: characterizing those design patterns.

Building on the data cube abstraction. Early relational database systems used indices as table replicas to enable vertical partitioning, associative search, and convenient data ordering. Database optimizers and executors use semi-join operations on these structures to run common queries on covering indices, thus realizing huge performance improvements.

Over the years, these early ideas evolved into materialized views (often maintained by triggers) that extended well beyond simple covering indices to enable accelerated access to star and snowflake data organizations. In the 1990s, we took another step forward by identifying the "data cube" OLAP (online analytic processing) pattern whereby data is aggregated along many different dimensions at once. Researchers developed algorithms for automating cube design and implementation that have proven to be both elegant and efficient—so much so that cubes for multi-terabyte fact tables can now be represented in just a few gigabytes. Virtually all major database engines already rely upon these algorithms. But that hardly signals an end to innovation. In fact, a considerable amount of research is now being devoted to this area, so we can look forward to much better cube querying and visualizing tools.

One step closer to knowledge. To see where we stand overall on our evolutionary journey, it might be said that along the slow climb from data to knowledge to wisdom, we're only now making our way into the knowledge realm. The advent of data mining represented our first step into that domain.

Now the time has come to build on those earliest mining efforts. Already, we've discovered how to embrace and extend machine learning through clustering, Bayes nets, neural nets, time-series analysis, and the like. Our next step is to create a *learning table* (labeled *T* for this discussion). The system can be instructed to learn columns *x*, *y*, and *z* from attributes *a*, *b*, and *c*—or, alternatively, to cluster attributes *a*, *b*, and *c* or perhaps even treat *a* as a time stamp for *b*. Then, with the addition of training data into learning table *T*, some data-mining algorithm builds a decision tree or Bayes net or time-series model for our

dataset. The training data that we need can be obtained using the database systems' already well-understood create/insert metaphor and its extract-transform-load tools. On the output side, we can ask the system at any point to display our data model as an XML document so it can be rendered graphically for easier human comprehension—but the real power is that the model can be used as both a data generator (“Show me likely customers,”) and as a tester (“Is Mary a likely customer?”). That is, given a key a, b, c , the model can return the x, y, z values—along with the associated probabilities of each. Conversely, T can evaluate the probability of some given value being correct. The significance in all this is that this is just the start. It is now up to the machine-learning community to add even better machine-learning algorithms to this framework. We can expect great strides in this area in the coming decade.

The research challenges most immediately ahead have to do with the need for better mining algorithms, as well as for better techniques for determining probabilistic and approximate answers (we'll consider this further in a moment).

Born-again column stores. Increasingly, one runs across tables that incorporate thousands of columns, typically because some particular object in the table features thousands of measured attributes. Not infrequently, many of the values in these tables prove to be null. For example, an LDAP object requires only seven attributes, while defining another 1,000 optional attributes.

Although it can be quite convenient to think of each object as a row in a table, actually representing them that way would be highly inefficient—both in terms of space and bandwidth. Classic relational systems generally represent each row as a vector of values, even in those instances where the rows are null. Sparse tables created using this row-store approach tend to be quite large and only sparsely populated with information.

One approach to storing sparse data is to plot it according to three characteristics: *key*, *attribute*, and *value*. This allows for extraordinary compression, often as a bitmap, which can have the effect of reducing query times by orders of magnitude—thus enabling a wealth of new optimization possibilities. Although these ideas first emerged in Adabase and Model204 in the early 1970s, they're currently enjoying a rebirth.

The challenge for researchers now is to develop automatic algorithms for the physical design of column stores, as well as for efficiently updating and searching them.

Dealing with messy data types. Historically, the database community has insulated itself from the informa-

tion retrieval community, preferring to remain blissfully unaware of everything having to do with messy data types such as time and space. (Mind you, not *everyone* in the database community has stuck his or her head in the sand—just most of us.) As it turns out, of course, we did have our work cut out for us just dealing with the “simple stuff”—numbers, strings, and relational operators. Still, there's no getting around the fact that real applications today tend to contain massive amounts of textual data and frequently incorporate temporal and spatial properties.

Happily, the integration of persistent programming languages and environments as part of database architectures now gives us a relatively easy means for adding the data types and libraries required to provide textual, temporal, and spatial indexing and access. Indeed, the SQL standard has already been extended in each of these areas. Nonetheless, all of these data types—most particularly, for text retrieval—require that the database be able to deal with approximate answers and employ probabilistic reasoning. For most traditional relational database systems, this represents quite a stretch. It's also fair to say that, before we're able to integrate textual, temporal, and spatial data types seamlessly into our database

Several forces are pushing data management systems into accommodating approximate reasoning.

frameworks, we still have much to accomplish on the research front. Currently, we don't have a clear algebra for supporting approximate reasoning, which we'll need not only to support these complex data types, but also to enable more sophisticated data-mining techniques. This same issue came up earlier in our discussion of data mining—data mining algorithms return ranked and probability-weighted results. So there are several forces pushing data management systems into accommodating approximate reasoning.

A Call To ARMS

Handling the semi-structured data challenge. Inconvenient though it may be, not all data fits neatly into the relational model. Stanford University professor Jennifer Widom observes that we all start with the schema `<stuff/>`, and *then* figure out what structure and constraints to add. It's also true that even the best-designed database can't include every conceivable constraint and is bound to leave at least a few relationships unspecified.

How best to deal with that reality is a question that has sparked a huge controversy within the database community. On one side of the debate are the radicals who believe cyberspace should be treated as one big XML document that can be manipulated with XQuery++. The reactionaries, on the other hand, believe that structure is your friend—and that, by extension, semistructured data is nothing more than a pluperfect mess best avoided. Both camps are well represented—and often stratified by age. It's easy to observe that the truth almost certainly lies somewhere between these two polar views, but it's also quite difficult to see exactly how this movie will end.

One interesting development worth noting, however, has to do with the integration of database systems and file systems. Individuals who keep thousands of e-mail messages, documents, photos, and music files on their own personal systems are hard-pressed to find much of anything anymore. Scale up to the enterprise level, where the number of files is in the billions, and you've got the same problem on steroids. Traditional folder hierarchy schemes and filing practices are simply no match for the information tsunami we all face today. Thus, a fully indexed, semistructured object database is called for to enable search capabilities that offer us decent precision and recall. What does this all signify? Paradoxically enough, it seems that file systems are evolving into database systems—which, if nothing else, goes to show just how fundamental the semistructured data problem really is. Data management architects still have plenty of work ahead of them before they can claim to have wrestled this problem to the mat.

Historically aware stream processing. Ours is a world increasingly populated by streams of data generated by

instruments that monitor environments and the activities taking place in those environments. The instances are legion, but here are just a few examples: telescopes that scan the heavens; DNA sequencers that decode molecules; bar-code readers that identify and log passing freight cars; surgical unit monitors that track the life signs of patients in post-op recovery rooms; cellphone and credit-card scanning systems that watch for signs of potential fraud; RFID scanners that track products as they flow through supply-chain networks; and smart dust that has been programmed to sense its environment.

The challenge here has less to do with the handling of all that streaming data—although that certainly does represent a significant challenge—than with what is involved in comparing incoming data with historical information stored for each of the objects of interest. The data structures, query operators, and execution environments for such stream-processing systems are qualitatively different from what people have grown accustomed to in classic DBMS environments. In essence, each arriving data item represents a fairly complex query against the existing database. The encouraging news here is that researchers have been building stream-processing systems for quite some time now, with many of the ideas taken from this work already starting to appear in mainstream products.

Triggering updates to database subscribers. The emergence of enterprise data warehouses has spawned a wholesale/retail data model whereby subsets of vast corporate data archives are published to various data marts within the enterprise, each of which has been established to serve the needs of some particular special interest group. This bulk publish/distribute/subscribe model, which is already quite widespread, employs just about every replication scheme you can imagine.

Now the trend in application design is to install custom subscriptions at the warehouse—sometimes millions of them at a time. What's more, realtime notification is being requested as part of each of these subscriptions. Whenever any data pertaining to the subscription arrives, the system is asked to immediately propagate this information to the subscriber. Hospitals want to know if a

patient's life signs change, travelers want to know if their flight is delayed, finance applications ask to be informed of any price fluctuations, inventory applications want to be told of any changes in stock levels, just as information retrieval applications want to be notified whenever any new content has been posted.

It turns out that publish/subscribe systems and stream-processing systems are actually quite similar in structure. First, the millions of standing queries are compiled into a dataflow graph, which in turn is incrementally evaluated to determine which subscriptions are affected by a change and thus must be notified. In effect, updated data ends up triggering updates to each subscriber that has indicated an interest in that particular information. The technology behind all this draws heavily on the active database work of the 1990s, but you can be sure that work continues to evolve. In particular, researchers are still looking for better ways to support the most sophisticated standing queries, while also optimizing techniques for handling the ever-expanding volume of queries and data.

Keeping query costs in line. All of the changes discussed so far are certain to have a huge impact on the workings of database query optimizers. The inclusion of user-defined functions deep inside queries, for one thing, is sure to complicate cost estimation. In fact, real data with high skew has always posed problems. But we'll no longer be able to shrug these off because in the brave new world we've been exploring, relational operators make up nothing more than the outer loop of nonprocedural programs and so really must be executed in parallel and at the lowest possible cost.

Cost-based, static-plan optimizers continue to be the mainstay for those simple queries that can be run in just a few seconds. For more complex queries, however, the query optimizer must be capable of adapting to varying workloads and fluctuations in data skew and statistics, while also planning in a much more dynamic fashion—changing plans in keeping with variations in system load and data statistics. For petabyte-scale databases, the only solution may be to run continuous data scans, with queries piggybacked on top of the scans.

The arrival of main-memory databases. Part of the challenge before us results from the insane pace of growth of disk and memory capacities, substantially outstripping bandwidth capacity and the current capabilities for minimizing latency. It used to take less than a second to read all of RAM and less than 20 minutes to read everything stored on a disk. Now, a multi-gigabyte RAM scan takes minutes, and a terabyte disk scan can require hours. It's also becoming painfully obvious that random access

is 100 times slower than sequential access—and the gap is widening. Ratios such as these are different from those we grew up with. They demand new algorithms that let multiprocessor systems intelligently share some massive main memory, while optimizing the use of precious disk bandwidth. Database algorithms, meanwhile, need to be overhauled to account for truly massive main memories (as in, able to accommodate billions of pages and trillions of bytes). In short, the era of main-memory databases has finally arrived.

Smart devices: Databases everywhere. We should note that at the other end of the spectrum from shared memory, intelligence is moving outward to telephones, cameras, speakers, and every peripheral device. Each disk controller, each camera, and each cellphone now combines tens of megabytes of RAM storage with a very capable processor. Thus, it's quite feasible now to have intelligent disks and other intelligent peripherals that provide for either database access (via SQL or some other nonprocedural language) or Web service access. The evolution from a block-oriented interface to a file interface and then on to a set of service interfaces has been the defining goal of database machine advocates for three decades now. In the past, this required special-purpose hardware. But that's not true any longer because disks now are armed with fast general-purpose processors, thanks to Moore's law. Database machines will likely enjoy a rebirth as a consequence.

In a related development, people building sensor networks have discovered that if you view each sensor as a row in a table (where the sensor values make up the fields in that row), it becomes quite easy to write programs to query the sensors. What's more, current distributed query technology, when augmented by a few new algorithms, proves to be quite capable of supporting highly efficient programs that minimize bandwidth usage and are quite easy to code and debug. Evidence of this comes in the form of the tiny database systems that are beginning to appear in smart dust—a development that's sure to shock and awe anyone who has ever fooled around with databases.

Self-managing and always-up. Indeed, if every file system, every disk, every phone, every TV, every camera, and every piece of smart dust is to have a database inside, then those database systems will need to be self-managing, self-organizing, and self-healing. The database community is justly proud of the advances it has already realized in terms of automating system design and operation. The result is that database systems are now ubiquitous—your e-mail system is a simple database, as is your

A Call To ARMS

file system, and so too are many of the other familiar applications you use on a regular basis. As you can probably tell from the list of new and emerging features enumerated in this article, however, databases are getting to be much more sophisticated. All of which means we still have plenty of work ahead to create distributed data stores robust enough to ensure that information never gets lost and queries are always handled with some modicum of efficiency.

STAYING ON TOP OF THE INFORMATION AVALANCHE
People and organizations are being buried under an unrelenting onslaught of information. As a consequence, everything you thought was true about database architectures is being re-thought.

Most importantly, algorithms and data are being unified by integrating familiar, portable programming languages into database systems, such that all those design rules you were taught about separating code from data simply won't apply any longer. Instead, you'll work with extensible object-relational database systems where non-procedural relational operators can be used to manipulate object sets. Coupled with that, database systems are well on their way to becoming Web services—and this will have huge implications in terms of how we structure applications. Within this new mind-set, DBMSs become object containers, with queues being the first objects that need to be added. It's on the basis of these queues that future transaction processing and workflow applications will be built.

Clearly, there's plenty of work ahead for all of us. The research challenges are everywhere—and none is trivial. Yet, the greatest of these will have to do with the unification of approximate and exact reasoning. Most of us come from the exact-reasoning world—but most of our clients are now asking questions that require approximate or probabilistic answers.

In response, databases are evolving from SQL engines to data integrators and mediators that offer transactional and nonprocedural access to data in many different forms. This means database systems are effectively

becoming database operating systems, into which various subsystems and applications can be readily plugged.

Getting from here to there will involve many more challenges than those touched upon here. But I do believe that most of the low-hanging fruit is clustered around the topics outlined here, with advances realized in those areas soon touching virtually all areas of application design. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

ACKNOWLEDGMENTS

Talks given by David DeWitt, Michael Stonebraker, and Jennifer Widom at CIDR (Conference on Innovative Data Systems Research) inspired many of the ideas presented in this article.

JIM GRAY is a Distinguished Engineer in Microsoft's Scalable Servers Research Group and is also responsible for managing Microsoft's Bay Area Research Group. He has been honored as an ACM Turing Award recipient for his work on transaction processing. To this day, Gray's primary research interests continue to concern database architectures and transaction processing systems. Currently, he's working with the astronomy community, helping to build online databases, such as <http://terraservice.net> and <http://skyserver.sdss.org>. Once all of the world's astronomy data is on the Internet and accessible as a single distributed database, Gray expects the Internet to become the world's best telescope.

MARK COMPTON, who now runs a marketing communications consulting group called Hired Gun Communications, has been working in the technology market for nearly 20 years. Before going out on his own, he headed up marketing programs at Silicon Graphics, where he was the driving force behind the branding program for the company's Indigo family of desktop workstations. Over a four-year period in the mid-1980s, he served as editor-in-chief of *Unix Review*, at the time considered the leading publication for Unix software engineers.

© 2005 ACM 1542-7730/05/0400 \$5.00