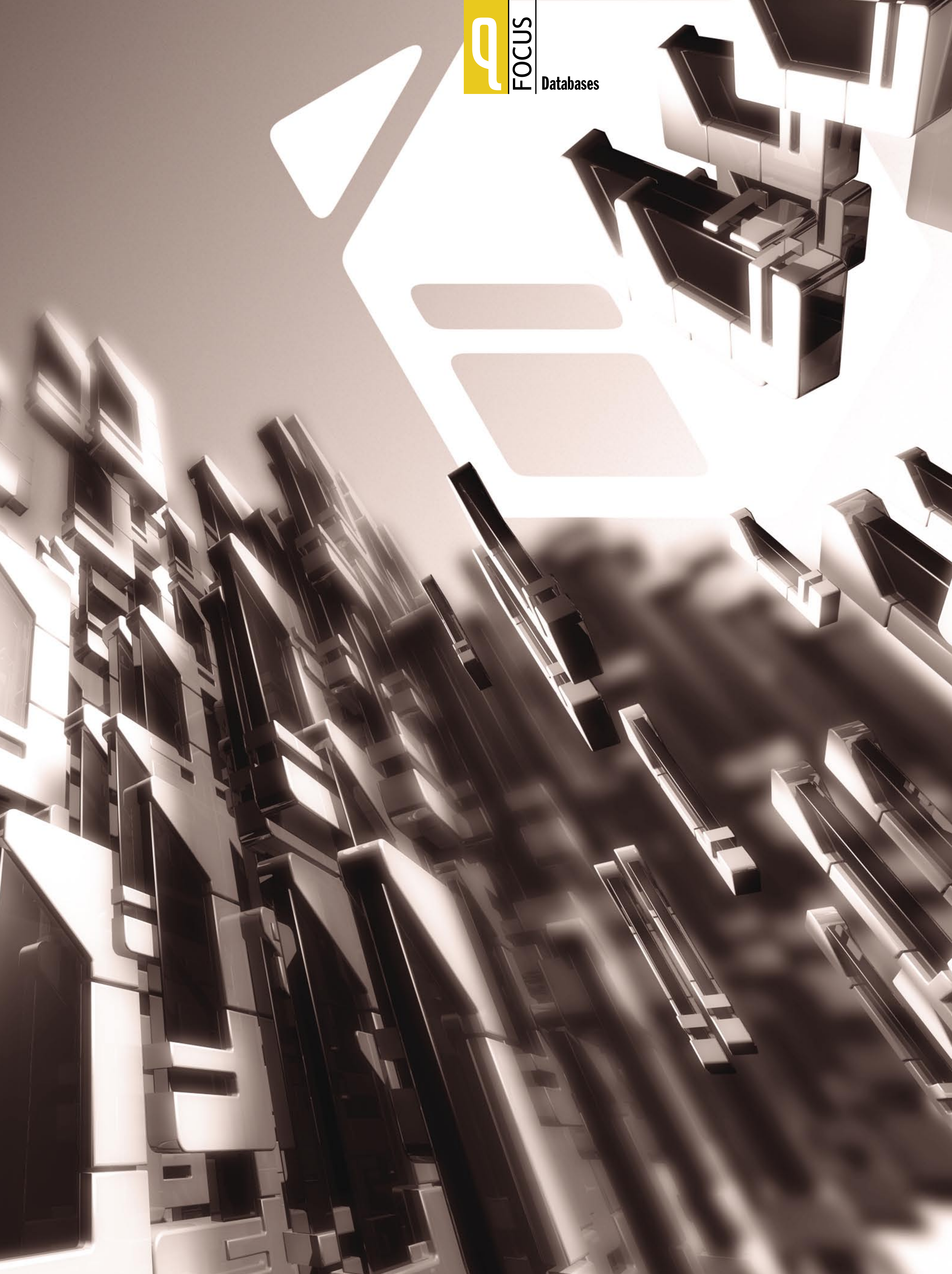
Beyond Relational Databases

There is more to data access than SQL.

The number and variety of computing devices in the environment are increasing rapidly. Real computers are no longer tethered to desktops or locked in server rooms. PDAs, highly mobile tablet and laptop devices, palmtop computers, and mobile telephony handsets now offer powerful platforms for the delivery of new applications and services. These devices are, however, only the tip of the iceberg. Hidden from sight are the many computing and network elements required to support the infrastructure that makes ubiquitous computing possible.

With so much computing power traveling around in briefcases and pockets, developers are building applications that would have been impossible just a few years ago. Among the interesting services available today are text and multimedia messaging, location-based search

MARGO SELTZER,
SLEEPYCAT



Beyond Relational Databases

and information services (for example, on-demand reviews of nearby restaurants), and ad hoc multiplayer games. Over the next several years, new classes of mobile and personalized services, impossible to predict today, will certainly be developed.

While these services differ from one another in major ways, they also share some important attributes. One—the focus of this article—is the need for data storage and retrieval functions built into the application. Messaging applications need to move messages around the network reliably and without loss. Location-based services need to map physical location to logical location (for example, GPS or cell-tower coordinates to postal code) and then look up location-based information. Gaming applications must record and share the current state of the game on distributed devices and manage content retrieval and delivery to each of the devices in realtime. In all these cases, fast, reliable data storage and retrieval are critical.

As soon as the discussion turns to data storage and retrieval, relational databases come to mind. Relational databases have been tremendously successful over the past three decades, and SQL has become the lingua franca for data access. While *data management* has become almost synonymous with RDBMS, however, there are an increasing number of applications for which lighter-weight alternatives are more appropriate.

In this article, we begin with a brief review of how relational systems came to dominate the data management landscape, discuss how the relational technologies have evolved, present a data-centric overview of today's emergent applications, and delve into data management needs for today's and tomorrow's applications.

RELATIONAL PREHISTORY

Relational databases came out of research at IBM[1,2] and the University of California at Berkeley[3] in the 1970s. Relational databases were fundamentally a reaction to the escalating costs required for deploying and maintaining complex systems.

The key observation was that programmers, who were very expensive, had to rewrite large amounts of application software manually whenever the content or physical organization of a database changed. Because the application generally knew in detail how its data was stored, including its on-disk layout, reorganizing databases or adding new information to existing databases forced wholesale changes to the code accessing those databases.

Relational databases solved this problem in two ways. First, they hid the physical organization of the database from the application and provided only a logical view of the data. Second, they used a declarative language to describe the data of interest in a particular query, rather than forcing the programmer to write a collection of function calls to fetch the data. These two changes allowed programmers to describe the information they wanted and to leave the details of optimization and access to the database management system. This transformation relieved programmers of the burden of rewriting application code whenever the database layout or organization changed.

Relational databases enjoyed tremendous success in the IT shops and data centers of the world. Businesses with large quantities of data to manage and sophisticated applications using that data adopted the new technology quickly. Demand for relational products created a market worth billions of dollars in licensing revenue per year. Several RDBMS vendors arose in the 1980s to compete for this lucrative business.

In the 20 years that followed, two related trends emerged. First, the RDBMS vendors increased functionality to provide market differentiators and to address each new market niche as it arose. Second, few applications need all the features available in today's RDBMSs, so as the feature set size increased, each application used a decreasing fraction of that feature set.

This drive toward increasing DBMS functionality has been accompanied by increasing complexity, and most deployments now require a specialist, trained in database administration, to keep the systems and applications running. Since these systems are developed and sold as monolithic entities, even though applications may require only a small subset of the system's functionality, each installation pays the price of the total overall complexity. Surely, there must be a better way.

THE NEW FRONTIER

We are not the first to notice these tides of change. In 1998, the leading database researchers concluded that database management systems were becoming too complex and that automated configuration and management were becoming essential.[4] Two years later, Surajit Chaudhuri and Gerhard Weikum proposed radically rethinking database management system architecture.[5] They suggested that database management systems be made more modular and that we broaden our thoughts about data management to include rather simple, component-based building blocks. Most recently, Michael Stonebraker joined the chorus, arguing that "one size no longer fits all" and citing particular application examples where the conventional RDBMS architecture is inappropriate.[6]

As argued by Stonebraker, the relational vendors have been providing the illusion that an RDBMS is the answer to any data management need. For example, as data warehousing and decision support have emerged as important application domains, the vendors have adapted products to address the specialized needs that arise in these new domains. They do this by hiding fairly different data management implementations behind the familiar SQL front end. This model breaks down, however, as one begins to examine emerging data needs in more depth.

The following sections introduce some of the new problems arising in data management. Our goal in examining these examples is to derive some emergent application classes for which conventional data management approaches may be suboptimal.

Data warehousing. Retail organizations now have the ability to record every customer transaction, producing an enormous data source that can be mined for information about customers' purchasing patterns, trends in product popularity, geographical preferences, and countless other phenomena that can be exploited to increase sales or decrease the cost of doing business. This database is *read-mostly*: it is updated in bulk by periodically adding new transactions to the collection, but it is read frequently as analysts cull the data extracting useful tidbits. This application domain is characterized by enormous tables (tens or hundreds of terabytes), queries that access only a few of the many columns in a table, and a need to scan tables sorted in a number of different ways.

Directory services. As organizations become increasingly dependent upon distributed resources and personnel, the demand for directory services has exploded.[7] Directory servers provide fast lookup of entities arranged in a hierarchical structure that frequently matches the hierarchical structure of an organization. The LDAP standard emerged in the 1990s in response to the heavyweight ISO X.400/X.500 directory services. LDAP is now at the core of authentication and identity management systems from a number of vendors (e.g., IBM Tivoli's Directory Server, Microsoft's Active Directory Server, and the Sun ONE Directory Server). Like data warehousing, LDAP is characterized by read-mostly access. Queries are either single row retrieval (find the record that corresponds to this user) or lookups based on attribute values (find all users in the engineering department). The prevalence of multivalued attributes makes relational representation quite inefficient.

Web search. Internet search engines lie at the intersection of database management and information retrieval. The objects upon which they operate are typically semi-structured (i.e., HTML instead of raw text), but the queries posed are most often keyword lookups where the desired response is a sorted list of possible answers. Practically all the successful search engines today have developed their own data management solutions to this problem, constructing efficient inverted indices and highly parallelized implementations of index and lookup. This application is read-mostly with bulk updates and nontraditional indexing.

Relational vendors have been providing the illusion that an RDBMS is the answer to any data management need.

Mobile device caching. The prevalence of small, mobile devices introduces yet another category of application: caching relevant portions of a larger dataset on a smaller, low-functionality device. While today's users think of their cellphone's directory as their own data collection, another view might be to think of it as a cache of a global phone and address directory. This model has attractive properties—in particular, the ability to augment the local dataset with entries as they are used or needed. Mobile telephony infrastructure requires similar caching

Beyond Relational Databases

capabilities to maintain communication channels to the devices. The access pattern observed in these caches is also read-mostly, and the data itself is completely transitory; it can be lost and regenerated if necessary.

XML management. Online transactions are increasingly being conducted by exchanging XML-encoded documents. The standard solution today involves converting these documents into a canonical relational organization, storing them in an RDBMS, and then converting again when one wishes to use them. As more documents are created, transmitted, and operated in XML, these translations become unnecessary, inefficient, and tedious. Surely there must be a better way. Native XML data stores with XQuery and XPath access patterns represent the next wave of storage evolution. While new items are constantly added to and removed from an XML repository, the documents themselves are largely read-only.

Stream processing. Stream processing is a bit of an outcast in this laundry list of data-intensive applications. Strictly speaking, stream processing is not a data management task; it is a data-filtering task. That is, data is produced at some source and sent streaming to recipients, which filter the stream for "interesting" events. For example, financial institutions watch stock tickers looking for hotly traded items and/or stocks that aren't being traded as heavily as expected.

The reason that these stream-processing applications are included here is a linguistic one: the filters that are typically desired in these environments *look* like SQL; however, while SQL was designed to operate on persistently stored tables, these queries act upon a realtime stream of data values. Stonebraker explains in some depth how poorly equipped databases are for this task. Perhaps the bigger surprise is not that database systems are poorly equipped to address this task, but that because SQL appears to be the "right" query language, developers use relational database systems for applications that have no persistent storage!

Stream processing represents a class of applications that could benefit from a SQL-like query language atop a data management system with properties that are radically different from an RDBMS. Since streaming queries frequently operate on data observed during a time window, some transient local storage is necessary, but this storage needn't be persistent, transactional, or support complex query processing. Instead, it needs to be blindingly fast. Although relational databases are well equipped to handle dynamic queries over relatively static or slowly changing data, this application class is characterized by a fairly static query set over highly dynamic data.

FLEXIBLE SOLUTIONS

Relational systems have been designed to satisfy OLTP (online transaction processing), workloads characterized by ad hoc queries, significant write traffic, and the need for strong transactional and integrity guarantees. In contrast, the applications described here are almost all read-dominated, and streaming applications don't even take advantage of persistent data, just an SQL-like query language. Few of these applications require transactional guarantees, and there is little inherently relational about the data being accessed. Thus, the data management question becomes how best to satisfy the needs of these different types of applications. As Stonebraker claims, there really is no single right answer. Instead, we must focus on flexible solutions that can be tailored to the needs of a particular application.

There are several ways to deliver flexibility in today's changing data environment. The back-to-basics approach is to require that every single application build its own data storage service. This option, while seemingly simple, is impractical in all but the simplest of applications. Some data-intensive applications running today, however, are built upon simple, homegrown solutions.

The second way to address the need for flexibility is to provide a smorgasbord of data management options, each of which addresses a particular application class. We see this approach emerging in the traditional relational market, where the SQL veneer is used to hide the different capabilities required for OLTP and data warehousing.

The third approach to flexibility is to produce a storage engine that is more configurable so that it can be tuned to the requirements of individual applications. This solution allows concentrated investment in a single storage system, improving quality. Configurability, however, makes new demands of developers who use the database, since they must understand the configuration options and then integrate the data management component properly into their product designs.

In fact, the solution emerging in the marketplace is to have a handful of reasonably configurable storage systems, each of which is useful across a broad application class.

There are fundamentally two properties that a solution must possess to address the wide range of application needs emerging today: modularity and configurability. Few applications require all the functionality possible in a data management system. If an application doesn't need functionality, it should not have to "pay" for that functionality in size (footprint, memory consumption, disk utilization, etc.), complexity, or cost. Therefore, a flexible engine must allow the developer to use or exclude major subsystems depending on whether the application needs them. Once a system is sufficiently modular to permit a truly small footprint, we will find that system deployed on an array of hardware platforms with staggeringly large differences in capabilities. In these cases, the system must be configurable to its operating environment: the specific hardware, operating system, and application using it. In the rest of this article, we discuss these two properties in more detail.

MODULARITY

Some argue that database architecture is in need of a revolution akin to the RISC revolution in computer hardware. The conventional monolithic DBMS architecture is not facile enough to adapt to today's data demands, so we need to build data management capabilities out of a collection of small, simple, reusable components. For example, instead of viewing SQL as a simple binary decision, Chaudhuri and Weikum argue that query capabilities should be provided at different levels of sophistication. You might begin with a single-table selection processor that has a B+ tree index that supports simple indexing, updating, and selection. To this, you might add transactions. Continuing up the complexity hierarchy, consider a select-project-join processor. Next, add aggregates. In this manner, you transform SQL from a monolithic language into a family of successively richer languages, each of which is provided as a component and satisfies a significant number of application domains. Any particular application selects the components it needs. This idea of a component-based architecture can be extended to include several other aspects of database design: concurrency control, transactions, logging, and high availability.

Concurrency control lends itself to a hierarchy similar to that presented in the language example. Some applications are completely single-threaded and require no locking; others have low levels of concurrency and would be well served by table-level locks or API-level locks (i.e., allow only one writer or multiple readers into the database system simultaneously); finally, highly concurrent applications need fine-grain locking and multiple degrees of isolation (potentially allowing applications to see values that have been written by incomplete transactions).[8] In a conventional database management system, locking is assumed; in the brave new world discussed here, locking is optional and different components can be used to provide different levels of concurrency.

Transactions provide the illusion that a collection of operations are applied to a database in an atomic unit and that once applied, the operations will persist, even in the face of application or system failure. Transaction management is at the heart of most database management systems, yet many applications do not require transactions. In a component-based world, transactions, too, should be optional. When they are present, a system might still have a number of different components providing basic transactional mechanisms, savepoints (the ability to identify a point in time to which the database may be rolled back), two-phase commit to support transactions that span multiple databases, nested transactions to decompose a large operation into a number of smaller ones, and compensating transactions to undo high-level, logical operations.

Many transaction systems use some form of logging to provide rollback and recovery capabilities. In that context, it hardly seems necessary to treat logging as a separable component, but it should be. A transactional component might be designed to work with multiple implementations, some of which do not use logging (e.g., now-overwrite schemes such as shadow-pages). Perhaps even more interesting, a logging system might be useful outside the context of transactions; it might be used for auditing or providing some sort of backup mechanism. In either case, it should be an application designer's decision whether logging is necessary rather than having it imposed by the database vendor.

Finally, data is sometimes so critical that downtime is unacceptable. Many database systems provide replicated or highly available systems to address this need. Although this functionality is often available as an add-

on in today's systems, they have not gone far enough. A developer may wish to use a database's HA (high-availability) configuration, but may use it in conjunction with some other company's HA substrate. If the application already has a substrate that performs heartbeat protocols (or any other mechanism that notifies the application or system when a component fails), fail-over, and redundant communication channels, then you will want to exclude those components from the database management system and hook into the existing functionality. Monolithic systems do not allow this, whereas a component-based, modular architecture does.

In addition to providing smaller, simpler applications, components with well-defined, clean, exposed interfaces provide for a degree of extensibility that is simply not possible in a monolithic system. For example, consider the basic set of components needed to construct a transactional system: a transaction manager, a lock manager, and a log manager. If these modules are open and extensible, then the developer can build into transactions various systems that incorporate items that are not managed by the database system. Consider, for example, a network switch: the state of the configuration database depends on the state of hardware inside the device, and vice versa. If the electrical control over chips and boards can be incorporated into transactions, by allowing the programmer to extend the locking and logging system to communicate with them, then operations such as "power up the backup network interface card" can be made transactional.

Modularity is a powerful tool for managing the size and complexity of applications and systems while also enabling the application and data management capabilities to seamlessly interact. This type of architecture enables developers to exclude functionality they do not need and include functionality they do need that is not provided by the database vendor.

CONFIGURABILITY

The second property of a flexible data management system is configurability. Whereas modularity is an architectural mechanism, configuration is mostly a runtime mechanism. With a component-based architecture, the build-time configuration is involved in selecting appropriate components. A single collection of components may still run on a range of systems with wildly different capabilities. For example, just because two applications both want transactions and B-trees, this does not mean that both can support a multi-gigabyte in-memory cache. The ability to adapt to radically different circumstances is critical. Configurability refers to how well a system can be matched to its environment and application needs. In this article we discuss configurability with respect to the hardware, the environment in which the application runs (e.g., the operating system), the application's software architecture, and the "natural" data format of the application.

Hardware environments introduce variability in CPU speed, memory size, and persistent storage capabilities. Variability in CPU speed and persistent storage introduces the possibility of trading computation for disk bandwidth. On a fast processor, it may be beneficial to compress data, consuming CPU cycles, in order to save I/O; on a PDA, where CPU cycles are sparse and persistent I/O is fast, compression might not be the right trade-off.

In a world where resource-constrained devices require potentially sophisticated data management, developers must have control over the memory and disk consumption policies of the database. In different environments, applications may need control over the maximum size of in-memory data structures, the maximum size of persistent data, and the space consumed by transactional logs. Policies for consumption of these resources must be set by the application developer, not the end user, since the developer is more likely to have the technical savvy necessary to make the right decisions.

Variability in persistent storage technologies places new demands on the database engine as well. Not only must it work well in the presence of spinning, magnetic storage, but it should also run well on other media (e.g., flash) with constraints on behaviors (such as the number of writes to a particular memory location), and it may need to run in the absence of any persistent storage. Some applications want to manage data entirely in main memory, with no persistence; some want to manage data with full synchronous transactional guarantees on updates; and some need something in the middle. Each of these policies should be implemented by the same transactional component, but the database should allow the programmer to control whether or not data persists across power-down events and the strictness of any transactional assurances that the system makes to the end user.

Although many embedded systems are now able to use COTS hardware platforms, many proprietary devices still exist. The ubiquitous data management solution will be portable to these special-purpose hardware devices. It will also be portable to a variety of operating systems; the services available from the operating system on a mobile telephone handset are different from those available on a 64-way multiprocessor with gigabytes of RAM, even if both are running Linux. If the data management system is to run everywhere, then it must rely only on the services common to most operating systems, and it must provide explicit mechanisms to allow portability, through simple interposition libraries or source-code availability.

Even on a single platform, the developer makes architectural choices that affect the database system. For example, a system may be built using: a single thread of control; a collection of cooperating processes, each of which is single-threaded; multiple threads of control in a single process; multiple multithreaded processes; or a strictly event-based architecture. These choices are driven by a combination of the application's requirements, the developer's preferences, the operating system, and the hardware. The database system must accommodate them.

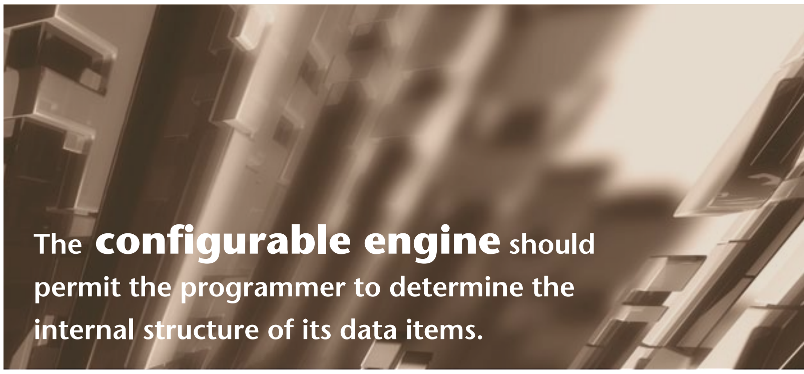
The database must also avoid making decisions about network protocols. Since the database will run in environments where communication takes place over backplanes, as well as environments where it takes place over WANs, the developer should select the appropriate communication infrastructure. A special-purpose telephone switch chassis may include a custom backplane and protocol for fast communication among redundant boards; the database must not prevent the developer from using it.

Up to this point, configurability has revolved around adapting to the hardware and software environment of the application. The last area of configuration that we address revolves around the application's data. Data layout, indexing, and access are critical performance considerations. There are three main design points with respect to data: the physical clustering, the indexing mechanism, and the internal structure of items in the database. Some of these, like the indexing mechanism, really are runtime configuration decisions, whereas others are more about giving the application the ability to make design decisions, rather than having designers forced into decisions because of the database management system.

Database management systems designed for spinning, magnetic media expend considerable effort clustering related data together on disk so that seek and rotation times can be amortized by transferring a large amount of data per repositioning event. In general, this clustering is good, as long as the data is clustered according to the "right" criteria. In the case of a configurable database system, this means that the developer needs to retain control over primary key selection (as is done in most relational database management systems) and must be able to ignore clustering issues if the persistent medium either doesn't exist or doesn't show performance benefits from accessing locations that are "close" to the last access.

On a related note, the developer must be left the flexibility to select an indexing structure for the primary keys that is appropriate for the workload. Workloads with locality of reference are probably well served by B+ trees; those with huge datasets and truly random access might be better off with hash tables. Perhaps the data is highly dimensional and requires a completely different indexing structure; the extensibility discussed in the previous section should allow a developer to provide an application-specific indexing mechanism and use it with all of the system's other features (e.g., locking, transactions). At a minimum, the configurable database should provide a range of alternative indexing structures that support iteration, fast equality searches, and range searches, including searches on partial keys.

Unlike relational engines, the configurable engine should permit the programmer to determine the internal structure of its data items. If the application has a dynamic or evolving schema or must support ad hoc


The configurable engine should permit the programmer to determine the internal structure of its data items.

queries, then the internal structure should be one that enables high-level query access such as SQL, XPath, XQuery, LDAP, etc. If, however, the schema is static and the query set is known, selecting an internal structure that maps more directly to the application's internal data structures provides significant performance improvements. For example, if an application's data is inherently nonrelational (e.g., containing multivalued attributes or large chunks of unstructured data), then forcing it into a relational organization simply to facilitate SQL access will cost performance in the translation and is unlikely to reap the benefits of the relational store. Similarly, if the application's data were relational, forcing it into a different format (e.g., XML, object-oriented) would add overhead for no benefit. The configurable engine must support storing data in the format that is most natural for the application. It is then the programmer's responsibility to select the format that meets the "most natural" criteria.

NEW-STYLE DATABASES FOR NEW-STYLE PROBLEMS

Old-style database systems solve old-style problems; we need new-style databases to solve new-style problems. While the need for conventional database management systems isn't going away, many of today's problems require a configurable database system. Even without a crystal ball, it seems clear that tomorrow's systems will also require a significant degree of configurability. As programmers and engineers, we learn to select the right tool to do a job; selecting a database is no exception. We need to operate in a mode where we recognize that there *are* options in data management, and we should select the right tool to get the job done as efficiently, robustly, and simply as possible. Q

REFERENCES

1. Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM* 13(6): 377-387.
2. Astrahan, M. M., et al. 1976. System R: Relational approach to database management. *ACM Transactions on Database Systems* 1(2): 97-137.
3. Stonebraker, M. 1976. The design and implementation of Ingres. *ACM Transactions on Database Systems* 1(3): 189-222.
4. Bernstein, P., et al. 1998. The Asilomar report on database research. *ACM SIGMOD Record* 27(4). http://www.sigmod.org/record/issues/9812/asilomar.html.
5. Chaudhuri, S., and Weikum, G. 2000. Rethinking database system architecture: Towards a self-tuning RISC-style database system. *The VLDB Journal:* 1-10. http://www.vldb.org/conf/2000/P001.pdf.
6. Stonebraker, M., and Cetintemel, U. 2005. One size fits all: An idea whose time has come and gone. Proceedings of the 2005 International Conference on Data Engineering (April). http://www.cs.brown.edu/~ugur/fits_all.pdf.
7. Broussard, F. 2004. Worldwide IT asset management software forecast and analysis, 2002-2007. IDC Doc. #30277. http://www.idc.com/getdoc.jsp?containerId=30277&pid=35178981.
8. Gray, J., and Reuter, A. 1993. *Transaction Processing: Concepts and Technologies*, 397-402. San Mateo, CA: Morgan Kaufman Publishers.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

MARGO I. SELTZER, Ph.D., is Herchel Smith professor of computer science and associate dean in the Division of Engineering and Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. Seltzer is also a founder and CTO of Sleepycat Software, the makers of Berkeley DB. She is a Sloan Foundation Fellow in computer science, a Bunting Fellow, and was the recipient of the 1996 Radcliffe Junior Faculty Fellowship and the University of California Microelectronics Scholarship. She won the Phi Beta Kappa teaching award in 1996 and the Abrahmson Teaching Award in 1999. She received an A.B. degree in applied mathematics from Harvard/Radcliffe College in 1983 and a Ph.D. in computer science from the University of California, Berkeley, in 1992.