DOI:10.1145/1953122.1953144

Partition data and operations, keep administration simple, do not assume one size fits all.

BY MICHAEL STONEBRAKER AND RICK CATTELL

10 Rules for Scalable Performance in 'Simple Operation' Datastores

THE RELATIONAL MODEL of data was proposed in 1970 by Ted Codd[5] as the best solution for the DBMS problems of the day—business data processing. Early relational systems included System R [2] and Ingres,[9] and almost all commercial relational DBMS (RDBMS) implementations today trace their roots to these two systems.

As such, unless you squint, the dominant commercial vendors—Oracle, IBM, and Microsoft—as well as the major open source systems—MySQL and PostgreSQL—all look about the same today; we term these systems general-purpose traditional row stores, or GPTRS, sharing the following features:

- Disk-oriented storage;
- Tables stored row-by-row on disk, hence, a row store;
- B-trees as the indexing mechanism;
- Dynamic locking as the concurrency-control mechanism;
- A write-ahead log, or WAL, for crash recovery;
- SQL as the access language; and
- A "row-oriented" query optimizer and executor, pioneered in System R.[7]

The 1970s and 1980s were characterized by a single major DBMS market—business data processing—today called online transaction processing, or OLTP. Since then, DBMSs have come to be used in a variety of new markets, including data warehouses, scientific databases, social-networking sites, and gaming sites; the modern-day DBMS market is characterized in the figure here.

The figure includes two axes: horizontal, indicating whether an application is read-focused or write-focused, and vertical, indicating whether an application performs simple operations (read or write a few items) or complex operations (read or write thousands of items); for example, the traditional OLTP market is write-focused with simple operations, while the data warehouse market is read-focused with complex operations. Many appli-

» key insights

- **Many scalable SQL and NoSQL datastores have been introduced over the past five years, designed for Web 2.0 and other applications that exceed the capacity of single-server RDBMSs.**
- **Major differences characterize these new datastores as to their consistency guarantees, per-server performance, scalability for read versus write loads, automatic recovery from failure of a server, programming convenience, and administrative simplicity.**
- **Applications must be designed for scalability, partitioning application data into "shards," avoiding operations that span partitions, designing for parallelism, and weighing requirements for consistency guarantees.**

1 Look for shared-nothing scalability.

2 High-level languages are GOOD and need not hurt performance.

3 Plan to carefully LEVERAGE main memory databases.

4 High availability and automatic recovery are essential for SO scalability.

5 ONLINE EVERYTHING.

6 Avoid MULTI-NODE operations.

7 Don't try to build ACID consistency yourself.

8 Look for administrative SIMPLICITY.

9 Pay attention to NODE performance.

10 OPEN SOURCE gives you more CONTROL over your future.

cations are, of course, in between; for example, social-networking applications involve mostly simple operations but also a balance of reads and writes. Hence, the figure should be viewed as a continuum in both directions, with any given application somewhere in between.

The major commercial engines and open source implementations of the relational model are positioned as "one-size-fits-all" systems; that is, their implementations are claimed to be appropriate for all locations in the figure.

However, there is also some dissatisfaction with one-size-fits-all. Witness, for example, the commercial success of the so-called column stores in the data-warehouse market. With these products, only those columns needed in the query are retrieved from disk, eliminating overhead for unused data. In addition, superior compression and indexing is obtained, since only one kind of object exists on each storage block, rather than several-to-many. Finally, main-memory bandwidth is economized through a query executor that operates on compressed data. For these reasons, column stores are remarkably faster than row stores on typical data-warehouse workloads, and we expect them to dominate the data-warehouse market over time.

Our focus here is on simple-operation (SO) applications, the lower portion of the figure. Quite a few new, non-GPTRS systems have been designed to provide scalability for this market. Loosely speaking, we classify them into the following four categories:

Key-value stores. Includes Dynamo, Voldemort, Membase, Membrain, Scalaris, and Riak. These systems have the simplest data model: a collection of objects, each with a key and a payload, providing little or no ability to interpret the payload as a multi-attribute object, with no query mechanism for non-primary attributes;

Document stores. Includes CouchDB, MongoDB, SimpleDB, and Terrastore in which the data model consists of objects with a variable number of attributes, some allowing nested objects. Collections of objects are searched via constraints on multiple attributes through a (non-SQL) query language or procedural mechanism;

Extensible record stores. Includes BigTable, Cassandra, HBase, HyperTable, and PNUTS providing variable-width record sets that can be partitioned vertically and horizontally across multiple nodes. They are generally not accessed through SQL; and

SQL DBMSs. Focus on SO application scalability, including MySQL Cluster, other MySQL derivatives, VoltDB, NimbusDB, and Clustrix. They retain SQL and ACID (Atomicity, Consistency, Isolation, and Durability)[a] transactions, but their implementations are often very different from those of GPTRS systems.

We do not claim this classification is precise or exhaustive, though it does cover the major classes of newcomer. Moreover, the market is changing rapidly, so the reader is advised to check other sources for the latest. For a more thorough discussion and references for these systems, see Cattell[4] and the table here.

The NoSQL movement is driven largely by the systems in the first three categories, restricting the traditional notion of ACID transactions by allowing only single-record operations to be transactions and/or by relaxing ACID semantics, by, say, supporting only "eventual consistency" on multiple versions of data.

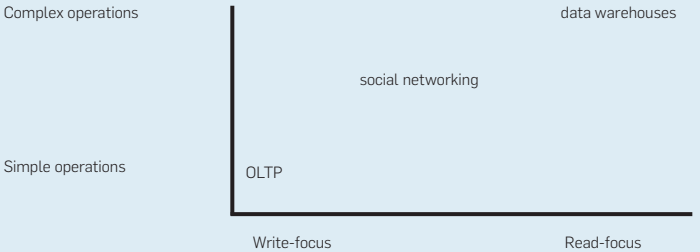
These systems are driven by a variety of motivations. For some, it is dissatisfaction with the relational model or the "heaviness" of RDBMSs. For others, it is the needs of large Web properties with some of the most demanding SO problems around. Large Web properties were frequently start-ups lucky enough to experience explosive growth, the so-called hockey-stick effect. They typically use an open source DBMS, because it is free or already understood by the staff. A single-node DBMS solution might be built for version 1, which quickly exhibits scalability problems. The conventional wisdom is then to "shard," or partitioning the application data over multiple nodes that share the load. A table can be partitioned this way; for example, employee names can be partitioned onto 26 nodes by putting all the "A"s on node 1 and so forth. It is now up to application logic to direct each query and update to the correct node. However, such sharding in application logic has a number of severe drawbacks:

- If a cross-shard filter or join must be performed, then it must be coded in the application;
- If updates are required within a transaction to multiple shards, then the application is responsible for somehow guaranteeing data consistency across nodes;
- Node failures are more common as the system scales. A difficult problem is how to maintain consistent replicas, detect failures, fail over to replicas, and replace failed nodes in a running system;
- Making schema changes without taking shards "offline" is a challenge; and
- Reprovisioning the hardware to add additional nodes or change the configuration is extremely tedious and, likewise, much more difficult if the shards cannot be taken offline.

Many developers of sharded Web applications experience severe pain because they must perform these functions in application-level logic; much of the NoSQL movement targets this pain point. However, with the large number of new systems and the wide

a ACID; see http://en.wikipedia.org/wiki/ACID

A characterization of DBMS applications.



range of approaches they take, customers[b] might have difficulty understanding and choosing a system to meet their application requirements.

Here, we present 10 rules we advise any customer to consider with an SO application and in examining non-GPTRS systems. They are a mix of DBMS requirements and guidelines concerning good SO application design. We state them in the context of customers running software in their own environment, though most also apply to software-as-a-service environments.

We lay out each rule, then indicate why it is necessary:

Rule 1. Look for shared-nothing scalability. A DBMS can run on three hardware architectures: The oldest—shared-memory multiprocessing (SMP)—means the DBMS runs on a single node consisting of a collection of cores sharing a common main memory and disk system. SMP is limited by main memory bandwidth to a relatively small number of cores. Clearly, the number of cores will increase in future systems, but it remains to be seen if main memory bandwidth will increase commensurately. Hence, multicore systems face performance limitations with DBMS software. Customers choosing an SMP system were forced to perform sharding themselves to obtain scalability across SMP nodes and face the painful problems noted earlier. Popular systems running on SMP configurations are MySQL, PostgreSQL, and Microsoft SQL Server.

A second option is to choose a DBMS that runs on disk clusters, where a collection of CPUs with private main memories share a common disk system. This architecture was popularized in the 1980s and 1990s by DEC, HP, and Sun but involves serious scalability problems in the context of a DBMS. Due to the private buffer pool in each node's main memory, the same disk block can be in multiple buffer pools. Hence, careful synchronization of these buffer-pool blocks is required. Similarly, a private lock table is included in each node's main memory. Careful synchronization is again required but limits the scalability of a shared disk configuration to a small number of nodes, typically fewer than 10.

Oracle RAC is a popular example of a DBMS running shared disk, and it is difficult to find RAC configurations with a double-digit number of nodes. Oracle recently announced Exadata and Exadata 2 running shared disk at the top level of a two-tier hierarchy while running shared-nothing at the bottom level.

The final architecture is a shared-nothing configuration, where each node shares neither main memory nor disk; rather, the nodes in a collection of self-contained nodes are connected to one another through networking. Essentially, all DBMSs oriented toward the data warehouse market since 1995 run shared-nothing, including Greenplum, Vertica, Asterdata, Paraccel, Netezza, and Teradata. Moreover, DB2 can run shared-nothing, as do many NoSQL engines. Shared-nothing engines normally perform automatic sharding (partitioning) of data to achieve parallelism. Shared-nothing systems scale only if data objects are partitioned across the system's nodes in a manner that balances the load. If there is data skew or "hot spots," then a shared-nothing system degrades in performance to the speed of the overloaded node. The application must also make the overwhelming majority of transactions "single-sharded," a point covered further in Rule 6.

Unless limited by application data/operation skew, well-designed, shared-nothing systems should continue to scale until networking bandwidth is exhausted or until the needs of the application are met. Many NoSQL

System information sources.

Systems	Link
Asterdata	http://asterdata.com
BigTable	http://labs.google.com/papers/bigtable.html
Clustrix	http://clustrix.com
CouchDB	http://couchdb.apache.org
DB2	http://ibm.com/software/data/db2
Dynamo	http://portal.acm.org/citation.cfm?id=1294281
Exadata	http://oracle.com/exadata
Greenplum	http://greenplum.com
Hadoop	http://hadoop.apache.org
HBase	http://hbase.apache.org
HyperTable	http://hypertable.org
MongoDB	http://mongodb.org
MySQL	http://mysql.com/products/enterprise
MySQL Cluster	http://mysql.com/products/database/cluster
Netezza	http://netezza.com
NimbusDB	http://nimbusdb.com
Oracle	http://oracle.com
Oracle RAC	http://oracle.com/rac
Paraccel	http://paraccel.com
PNUTs	http://research.yahoo.com/pub/2304
PostgreSQL	http://postgresql.org
Riak	http://basho.com/Riak.html
Scalaris	http://code.google.com/p/scalaris
SimpleDB	http://amazon.com/simpledb
SQL Server	http://microsoft.com/sqlserver
Teradata	http://teradata.com
Terrastore	http://code.google.com/p/terrastore
Tokyo Cabinet	http://1978th.net/tokyocabinet
Vertica	http://vertica.com
Voldemort	http://project-voldemort.com
VoltDB	http://voltdb.com

b We use the term "customer" to refer to any organization evaluating or using one of these systems, even though some of them are open source, with no vendor.

systems reportedly run 100 nodes or more, and BigTable reportedly runs on thousands of nodes.

The DBMS needs of Web applications can drive DBMS scalability upward in a hurry; for example, Facebook was recently sharding 4,000 MySQL instances in application logic. If it chose a DBMS, it would have to scale at least to this number of nodes. An SMP or shared-disk DBMS has no chance at this level of scalability. Shared-nothing DBMSs are the only game in town.

Rule 2. High-level languages are good and need not hurt performance. Work in a SQL transaction can include the following components:

- Overhead resulting from the optimizer choosing an inferior execution plan;
- Overhead of communicating with the DBMS;
- Overhead inherent in coding in a high-level language;
- Overhead for services (such as concurrency control, crash recovery, and data integrity); and
- Truly useful work to be performed, no matter what.

Here, we cover the first three, leaving the last two for Rule 3. Hierarchical and network systems were the dominant DBMS solutions in the 1960s and 1970s, offering a low-level procedural interface to data. The high-level language of RDBMSs was instrumental in displacing these DBMSs for three reasons:

- A high-level language system requires the programmer write less code that is easier to understand;
- Users state what they want instead of writing a disk-oriented algorithm on how to access the data they need; a programmer need not understand complex storage optimizations; and
- A high-level language system has a better chance of allowing a program to survive a change in the schema without maintenance or recoding; as such, low-level systems require far more maintenance.

One charge leveled at RDBMSs in the 1970s and 1980s was they could not be as efficient as low-level systems. The claim was that automatic query optimizers could not do as good a job as smart programmers. Though early optimizers were primitive, they were quickly as good as all but the best human programmers. Moreover, most organizations could never attract and retain this level of talent. Hence, this source of overhead has largely disappeared and today is only an issue on very complex queries rarely found in SO applications.

The second source of overhead is communicating with the DBMS. For security reasons, RDBMSs insist on the application being run in a separate address space, using ODBC or JDBC for DBMS interaction. The overhead of these communication protocols is high; running a SQL transaction requires several back-and-forth messages over TCP/IP. Consequently, any programmer seriously interested in performance runs transactions using a stored-procedure interface, rather than SQL commands over ODBC/JDBC. In the case of stored procedures, a transaction is a single over-and-back message. The DBMS further reduces communication overhead by batching multiple transactions in one call. The communication cost is a function of the interface selected, can be minimized, and has nothing to do with the language level of the interaction.

The third source of overhead is coding in SQL rather than in a low-level procedural language. Since most serious SQL engines compile to machine code or at least to a Java-style intermediate representation, this overhead is not large; that is, standard language compilation converts a high-level specification into a very efficient low-level runtime executable.

Hence, one of the key lessons in the DBMS field over the past 25 years is that high-level languages are good and do not hurt performance. Some new systems provide SQL or a more limited higher-level language; others provide only a "database assembly language," or individual index and object operations. This low-level interface may be adequate for very simple applications, but, in all other cases, high-level languages provide compelling advantages.

Rule 3. Plan to carefully leverage main memory databases. Consider a cluster of 16 nodes, each with 64GB of main memory. Any shared-nothing DBMS thereby has access to about 1TB of main memory. Such a hardware configuration would have been considered extreme a few years ago but is commonplace today. Moreover, memory per node will increase in the future, and the number of nodes in a cluster is also likely to increase. Hence, typical future clusters will have increasing terabytes of main memory.

As a result, if a database is a couple of terabytes or less (a very large SO database), customers should consider main-memory deployment. If a database is larger, customers should consider main-memory deployment when practical. In addition, flash memory has become a promising storage medium, as prices have decreased.

Given the random-access speed of RAM versus disk, a DBMS can potentially run thousands of times faster. However, the DBMS must be architected properly to utilize main memory efficiently; only modest improvements are achievable by simply running a DBMS on a machine with more memory.

To understand why, consider the CPU overhead in DBMSs. In 2008, Harizopoulos et al.[6] measured performance using part of a major SO benchmark, TPC-C, on the Shore open-source DBMS. This DBMS was chosen because the source code was available for instrumentation and because it was a typical GPTRS implementation. Based on simple measures of other GPTRS systems, the Shore results are representative of those systems as well.

Harizopoulos et al.[6] used a database size that allowed all data to fit in main memory, as it was consistent with most SO applications. Since Shore, like other GPTRS systems, is disk-based, it meant all data would reside in the main memory buffer pool. Their goal was to categorize DBMS overhead on TPC-C; they ran the DBMS in the same address space as the application driver, avoiding any TCP/IP cost. They then looked at the components of CPU usage that perform useful work or deliver DBMS services.

Following is the CPU cycle usage for various tasks in the new-order transaction of TPC-C; since Harizopoulos et al.[6] noted some shortcomings that are fixed in most commercial GPTRSs, these results have been scaled to assume removal of these sources of overhead:

Useful work (13%). This is the CPU

cost for actually finding relevant records and performing retrieval or update of relevant attributes;

Locking (20%). This is the CPU cost of setting and releasing locks, detecting deadlock, and managing the lock table;

Logging (23%). When a record is updated, the before and after images of the change are written to a log. Shore then groups transactions together in a "group commit," forcing the relevant portions of the log to disk;

Buffer pool overhead (33%). Since all data resides in the buffer pool, any retrievals or updates require finding the relevant block in the buffer pool. The appropriate record(s) must then be located and the relevant attributes in the record found. Blocks on which there is an open database cursor must be "pinned" in main memory. Moreover, Least-Recently-Used or another replacement algorithm is utilized, requiring the recording of additional information; and

Multithreading overhead (11%). Since most DBMSs are multithreaded, multiple operations are going on in parallel. Unfortunately, the lock table is a shared data structure that must be "latched" to serialize access by the various parallel threads. In addition, B-tree indexes and resource-management information must be similarly protected. Latches (mutexes) must be set and released when shared data structures are accessed. See Harizopoulos et al.[6] for a more detailed discussion, including on why the latching overhead may be understated.

A conventional disk-based DBMS clearly spends the overwhelming majority of its cycles on overhead activity. To go a lot faster, the DBMS must avoid all the overhead components discussed here; for example, a main memory DBMS with conventional multithreading, locking, and recovery is only marginally faster than its disk-based counterpart. A NoSQL or other database engine will not dramatically outperform a GPTRS implementation, unless all these overhead components are addressed or the GPTRS solution has not been properly architected (by, say, using conversational SQL rather than a compiled stored procedure interface).

We look at single-machine performance in our analysis, but this performance has a direct effect on the multi-machine scalability discussed in Rule 1, as well as in our other rules.

Shared-nothing DBMSs are the only game in town.

Rule 4. High availability and automatic recovery are essential for SO scalability. In 1990, a typical DBMS application would run on what we would now consider very expensive hardware. If the hardware failed, the customer would restore working hardware, reload the operating system and DBMS, then recover the database to the state of the last completed transaction by performing an undo of incomplete transactions and a `redo` of completed transactions using a DBMS log. This process could take time (several minutes to an hour or more) during which the application would be unavailable.

Few customers today are willing to accept any downtime in their SO applications, and most want to run redundant hardware and use data replication to maintain a second copy of all objects. On a hardware failure, the system switches over to the backup and continues operation. Effectively, customers want "nonstop" operation, as pioneered in the 1980s by Tandem Computers.

Furthermore, many large Web properties run large numbers of shared-nothing nodes in their configurations, where the probability of failure rises as the number of "moving parts" increases. This failure rate renders human intervention impractical in the recovery process; instead, shared-nothing DBMS software must automatically detect and repair failed nodes.

Any DBMS acquired for SO applications should have built-in high availability, supporting nonstop operation. Three high-availability caveats should be addressed. The first is that there is a multitude of kinds of failure, including:

- Application, where the application corrupts the database;
- DBMS, where the bug can be recreated (so-called Bohr bugs);
- DBMS, where the bug cannot be recreated (so-called Heisenbugs);
- Hardware, of all kinds;
- Lost network packets;
- Denial-of-service attacks; and
- Network partitions.

Any DBMS will continue operation for some but not for all these failure

modes. The cost of recovering from all possible failure modes is very high. Hence, high availability is a statistical effort, or how much availability is desired against particular classes of failures.

The second caveat is the so-called CAP, or consistency, availability, and partition-tolerance, theorem.[3] In the presence of certain failures, it states that a distributed system can have only two out of these three characteristics: consistency, availability, and partition-tolerance. Hence, there are theoretical limits on what is possible in the high-availability arena.

Moreover, many site administrators want to guard against disasters (such as earthquakes and floods). Though rare, recovery from disasters is important and should be viewed as an extension of high availability, supported by replication over a wide-area network.

Rule 5. Online everything. An SO DBMS should have a single state: "up." From the user's point of view, it should never fail and never have to be taken offline. In addition to failure recovery, we need to consider operations that require the database be taken offline in many current implementations:

Schema changes. Attributes must be added to an existing database without interruption in service;

Index changes. Indexes should be added or dropped without interruption in service;

Reprovisioning. It should be possible to increase the number of nodes used to process transactions, without interruption in service; for example, a configuration might go from 10 nodes to 15 nodes to accommodate an increase in load; and

Software upgrade. It should be possible to move from version N of a DBMS to version $N+1$ without interruption of service.

Though supporting these operations is a challenge, 100% uptime should be the goal. As an SO system scales to dozens of nodes and/or millions of users on the Internet, downtime and manual intervention are not practical.

Rule 6. Avoid multi-node operations. Two characteristics are necessary for achieving SO scalability over a cluster of servers:

Even split. The database and application load must be split evenly over the servers. Read-scalability can be achieved by replicating data, but general read/write scalability requires sharding (partitioning) the data over nodes according to a primary key; and

Scalability advantage. Applications rarely perform operations spanning more than one server or shard. If a large number of servers is involved in processing an operation, the scalability advantage may be lost because of redundant work, cross-server communication, or required operation synchronization.

Avoid multi-shard operations to the greatest extent possible, including queries that must go to multiple shards, as well as multi-shard updates requiring ACID properties.

Suppose a customer has an employee table and partitions it based on employee age. If it wants to know the salary of a specific employee, it must then send the query to all nodes, requiring a slew of messages. Only one node will find the desired data; the others will run a redundant query that finds nothing. Furthermore, if an application performs an update that crosses shards, giving, say, a raise to all employees in the shoe department, then the system must pay all of the synchronization overhead of ensuring the transaction is performed on every node.

Hence, a database administrator (DBA) should choose a sharding key to make as many operations single-shard as possible. Fortunately, most applications naturally involve single-shard transactions, if the data is partitioned properly; for example, if purchase orders (POs) and their details are both sharded on PO number, then the vast majority of transactions (such as new PO and update a specific PO) go to a single node.

The percentage of single-node transactions can be increased further by replicating read-only data; for example, a list of customers and their addresses can be replicated at all sites. In many business-to-business environments, customers are added or deleted or change their addresses infrequently. Hence, complete replication allows inserting the address of a customer into a new PO as a single-node operation. Therefore, selective replication of read-mostly data can be advantageous.

In summary, programmers should avoid multi-shard operations to the greatest extent possible, including queries that must go to multiple shards, as well as multi-shard updates

requiring ACID properties. Customers should carefully think through their application and database design to accomplish this goal. If that goal is unachievable with the current application design, they should consider a redesign that achieves higher "single-shardedness."

Rule 7. Don't try to build ACID consistency yourself. In general, the key-value stores, document stores, and extensible record stores we mentioned have abandoned transactional ACID semantics for a weaker form of atomicity, isolation, and consistency, providing one or more of the following alternative mechanisms:

- Creating new versions of objects on every write that result in parallel versions when there are multiple asynchronous writes; it is up to application logic to resolve the resulting conflict;
- Providing an "update-if-current" operation that changes an object only if it matches a specified value; this way, an application can read an object it plans to later update and then make changes only if the value is still current;
- Providing ACID semantics but only for read and write operations of a single object, attribute, or shard; and
- Providing "quorum" read-and-write operations that guarantee the latest version among "eventually consistent" replicas.

It is possible to build your own ACID semantics on any of these systems, given enough additional code. However, the task is so difficult, we wouldn't wish it on our worst enemy. If you need ACID semantics, you want to use a DBMS that provides them; it is much easier to deal with this at the DBMS level than at the application level.

Any operation requiring coordinated updates to two objects is likely to need ACID guarantees. Consider a transaction that moves $10 between two user accounts; with an ACID system, a programmer can simply write:

```
Begin transaction
Decrement account A
Increment account B
Commit transaction
```

Without an ACID system, there is no easy way to perform this coordinated action. Other cases requiring ACID semantics include charging customers' accounts only if their orders ship and synchronously updating bilateral "friend" references. Standard ACID semantics give the programmer the all-or-nothing guarantee needed to maintain data integrity in them. Although some applications do not need such coordination, a commitment to a non-ACID system precludes extending such applications in the future in a way that requires coordination. DBMS applications often live a long time and are subject to unknown future requirements.

We understand the NoSQL movement's motivation for abandoning transactions, given its belief that transactional updates are expensive in traditional GPTRS systems. However, newer SQL engines can offer both ACID and high performance by carefully eliminating all overhead in Rule 3, at least for applications that obey Rule 6 (avoid multinode operations). If you need ACID transactions and cannot follow Rule 6, then you will likely incur substantial overhead, no matter whether you code the ACID yourself or let the DBMS do it. Letting the DBMS do it is a no-brainer.

We have heard the argument for abandoning ACID transactions based on the CAP theorem,[3] stating you can have only two of three characteristics: C consistency, A availability, and P partition-tolerance. The argument is that partitions happen, hence one must abandon consistency to achieve high availability. We take issue for three reasons: First, some applications really do need consistency and cannot give it up. Second, the CAP theorem deals with only a subset of possible failures, as noted in Rule 4, and one is left with how to cope with the rest. And third, we are not convinced that partitions are a substantial issue for data sharded on a LAN, particularly with redundant LANs and applications on the same site; in this case, partitions may be rare, and one is better off choosing consistency (all the time) over availability during a very rare event.

Though true that WAN partitions are much more likely than LAN partitions, WAN replication is normally used for read-only copies or disaster recovery (such as when an entire data center goes offline); WAN latency is too high for synchronous replication or sharding. Few users expect to recover from major disasters without short availability hiccups, so the CAP theorem may be less relevant in this situation.

We advise customers who need ACID to seek a DBMS that provides it, rather than code it themselves, minimizing the overhead of distributed transactions through good database and application design.

Rule 8. Look for administrative simplicity. One of our favorite complaints about relational DBMSs is their poor out-of-the-box behavior. Most products include many tuning knobs that allow adjustment of DBMS behavior; moreover, our experience is that a DBA skilled in a particular vendor's product, can make it go a factor of two or more faster than one unskilled in the given product.

As such, it is a daunting task to bring in a new DBMS, especially one distributed over many nodes; it requires installation, schema construction, application design, data distribution, tuning, and monitoring. Even getting a high-performance version of TPC-C running on a new engine takes weeks, though code and schema are readily available. Moreover, once an application is in production, it still requires substantial DBA resources to keep it running.

When considering a new DBMS, one should carefully consider the out-of-the-box experience. Never let the vendor do a proof-of-concept exercise for you. Do the proof of concept yourself, so you see the out-of-the-box situation up close. Also, carefully consider application-monitoring tools in your decision.

Lastly, pay particular attention to Rule 5. Some of the most difficult administrative issues (such as schema changes and reprovisioning) in most systems require human intervention.

Rule 9. Pay attention to node performance. A common refrain heard these days is "Go for linear scalability; that way you can always provision to meet your application needs, while node performance is less important." Though true that linear scalability is important, ignoring node performance is a big mistake. One should always remember that linear scalability

means overall performance is a multiple of the number of nodes times node performance. The faster the node performance, the fewer nodes one needs.

It is common for solutions to differ in node performance by an order of magnitude or more; for example, in DBMS-style queries, parallel DBMSs outperform Hadoop by more than an order of magnitude,[1] and, similarly, H-store (the prototype predecessor to VoltDB) has been shown to have even higher throughput on TPC-C compared to the products from major vendors.[8] For example, consider a customer choosing between two database solutions, each offering linear scalability. If solution A offers node performance a factor of 20 better than solution B, the customer might require 50 hardware nodes with solution A versus 1,000 nodes with solution B.

Such a wide difference in hardware cost, rack space, cooling, and power consumption is obviously non-trivial between the two solutions. More important, if each node fails on average every three years, then solution B will see a failure every day, while solution A will see a failure less than once a month. This dramatic difference will heavily influence how much redundancy is installed and how much administrative time is required to deal with reliability. Node performance makes everything else easier.

Rule 10. Open source gives you more control over your future. This final rule is not a technical point but still important to mention, and, hence, perhaps, should be a suggestion rather than a rule. The landscape is littered with situations where a company acquired a vendor's product, only to face expensive upgrades in the following years, large maintenance bills for often-inferior technical support, and the inability to avoid these fees because the cost of switching to a different product would require extensive recoding. The best way to avoid "vendor malpractice" is to use an open source product. Open source eliminates expensive licenses and upgrades and often provides multiple alternatives for support, new features, and bug fixes, including the option of doing them in-house.

For these reasons, many newer Web-oriented shops are adamant about using only open source systems. Also, several vendors have proved it possible to make a viable business with an open source model. We expect it to be more popular over time, and customers are well advised to consider its advantages.

The best way to avoid "vendor malpractice" is to use an open source product.

Conclusion

The 10 rules we've presented specify the desirable properties of any SO datastore. Customers looking at distributed data-storage solutions are well advised to view the systems they are considering in the context of this rule set, as well as in that of their own unique application requirements. The large number of systems available today range considerably in capabilities and limitations.

Acknowledgments

We would like to thank Andy Pavlo, Rick Hillegas, Rune Humborstad, Stavros Harizopoulos, Dan DeMaggio, Dan Weinreb, Daniel Abadi, Evan Jones, Greg Luck, and Bobbi Heath for their valuable input on this article.

References

1. Abadi, D. et al. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 SIGMOD Conference on Management of Data* (Providence, RI, June 29). ACM Press, New York, 2009, 165–178.
2. Astrahan, M.M. et al. System R: A relational approach to data management. *ACM Transactions on Database Systems 1*, 2 (June 1976), 97–137.
3. Brewer, E. Towards Robust Distributed Systems. Keynote at Conference on Principles of Distribute Computing (Portland, OR, July 2000); http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
4. Cattell, R. Scalable SQL and NoSQL datastores. *ACM SIGMOD Record 40*, 2 (June 2011).
5. Codd, E.F. A relational model of data for large shared databanks. *Commun. ACM 13*, 6 (June 1970), 377–387.
6. Harizopoulos, S. et al. OLTP: Through the looking glass and what we found there. In *Proceedings of the 2008 SIGMOD Conference on Management of Data* (Vancouver, B.C., June 10). ACM Press, New York, 2008, 981–992.
7. Selinger, P. Access path selection in a relational data management system. In *Proceedings of the 1979 ACM SIGMOD Conference on Management of Data* (Boston, May 30). ACM Press, New York, 1979, 23–24.
8. Stonebraker, M. et al. The end of an architectural era (It's time for a complete rewrite). In *Proceedings of the 2007 Very Large Databases Conference* (Vienna, Austria, Sept. 23). ACM Press, New York, 2007, 399–410.
9. Stonebraker, M. et al. The design and implementation of Ingres. *ACM Transactions on Database Systems 1*, 3 (Sept. 1976), 189–222.

Michael Stonebraker (stonebraker@csail.mit.edu) is an adjunct professor in the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology, consultant and founder, Paradigm4, Inc., consultant and founder, Goby, Inc., and consultant and founder, VoltDB, Inc.

Rick Cattell (rick@cattell.net) is a database technology consultant at Cattell.Net and on the technical advisory board of Schooner Information Technologies.