Ten Rules for Scalable Performance in "Simple Operation" Datastores

By

Michael Stonebraker and Rick Cattell

Abstract

In this paper we present 10 rules that any customer would be wise to keep in mind when considering a database management system (DBMS) application that requires scalable performance, and that reads or writes a small number of objects in each transaction. Such simple operation (SO) applications include on-line transaction processing (OLTP) and also interactive web applications such as social networking. Some of our rules concern the underlying DBMS; the rest are guidelines for architecting scalable DBMS applications. Most existing SO DBMSs satisfy only a few of our rules, so this paper also represents a mandate for future improvement.

Introduction

The relational model of data was proposed in 1970 by Ted Codd [1] as the desired solution for the DBMS problems of the day, namely business data processing. Early relational systems included System R [2] and Ingres [3], and almost all commercial relational DBMS (RDBMS) implementations trace their roots to these two systems.

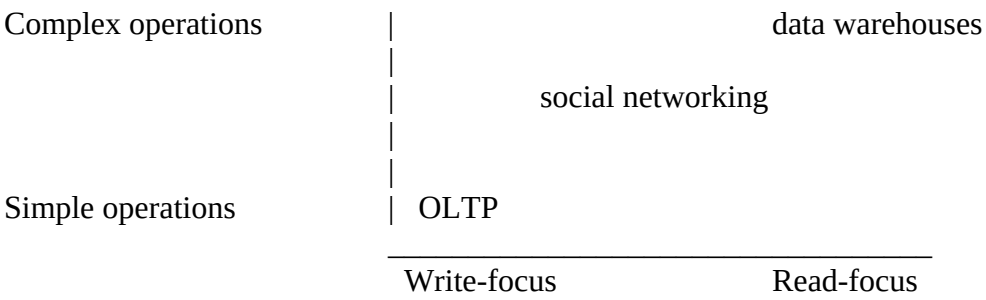
As such, unless you squint, the dominant commercial vendors (Oracle, IBM, Microsoft) as well as the major open source systems (MySQL, PostgreSQL) all look about the same, and we term these systems **general-purpose traditional row stores** (GPTRS). They share the following features:

- 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 (WAL) for crash recovery
- SQL as the access language
- A "row-oriented" query optimizer and executor (pioneered in System R [4])

In the 1970's and 1980's, there was only a single major DBMS market, business data processing, now called On-Line Transaction Processing (OLTP). More recently, DBMSs have come to be used in a wide variety of new markets, including data warehouses,

scientific databases, social networking sites, and gaming sites. We characterize the modern-day DBMS space in Figure 1.

Here, there are two axes, with the horizontal axis indicating whether the application is read-focused or write-focused. The vertical axis shows whether the 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. Of course, many applications are somewhere in the middle; for example social networking applications involve mostly simple operations but have a balance of reads and writes. Hence, one should view Figure 1 as a continuum in both directions, with any given applications placed somewhere in the interior of the figure.



A Characterization of DBMS Applications
Figure 1

FIGURE NEEDS TO BE REDRAWN

The major commercial engines and open-source implementations of the relational model are positioned as “one-size-fits-all” systems. In other words, their implementations are claimed to be appropriate for all locations in Figure 1.

There is at least some dissatisfaction with one-size-fits-all in certain quarters. Witness for example the commercial success of the so-called column stores in the data warehouse market. Here, only those columns needed in the query are retrieved from disk, avoiding bandwidth for unused data. In addition, superior compression can be obtained, since only one kind of object exists on each storage block, rather than several-to-many. Finally, main memory bandwidth is economized by a query executor that operates on compressed data. For these reasons columns stores are remarkably faster than row stores on typical data warehouse workloads, and we expect them to dominate this market over time.

The focus of this paper is on simple operation (SO) applications, i.e. the lower portion of Figure 1. In this market, there have been quite a number of new, non-GPTRS systems. Loosely speaking, we can classify these recent arrivals as follows:

1. Key-value stores, including Dynamo, Voldemort, Tokyo Cabinet, Scalaris, and Riak. These systems have the simplest data model: a collection of objects, each with a key and a payload. They provide little or no ability to interpret the payload as a multi-attribute object, and there is no query mechanism for non-primary attributes.
2. Document stores, including CouchDB, MongoDB, and SimpleDB. Here the data model consists of objects with a variable number of attributes; some allow nested objects. Collections of objects can be searched via constraints on multiple attributes through a (non SQL) query language or procedural mechanism.
3. Extensible record stores, including BigTable, PNUTs, HBase, HyperTable, and Cassandra. These provide variable-width tables that can be partitioned vertically and horizontally across multiple nodes. They are generally not accessed through SQL.
4. SQL DBMSs focused on SO application scalability, including MySQL Cluster and other MySQL derivatives, VoltDB, NimbusDB, and Clustrix. These systems retain SQL and ACID transactions, but often have very different implementations than GPTRS systems.

We do not claim this classification is precise nor exhaustive, however we believe it covers the major classes of newcomers. Readers seeking a more thorough discussion and references for these systems can consult [10].

The NoSQL movement is largely driven by the systems in the first three categories. These systems restrict the traditional notion of ACID (A: Atomic, C: consistent, I: Isolated, and D: Durable) transactions [5], either by allowing only single-record operations to be transactions and/or by relaxing ACID semantics, for example by supporting only "eventual consistency" or multiple versions of data.

There are a variety of motivations for the above systems. In some cases, it is dissatisfaction with the relational model or the "heaviness" of RDBMSs. In other cases, these new solutions are motivated by the needs of large web properties, which have some of the most demanding SO problems around. Frequently, a large web property was a start-up lucky enough to see explosive growth; the so-called "hockey stick" effect. Typically, they used an open source DBMS, because it was free or already understood by the staff. A single node DBMS solution was built for Version 1, which quickly exhibited scalability problems. The conventional wisdom was then to "shard" the application so that data could be stored on multiple nodes. In this way, a table is partitioned, for example, employee names could be partitioned onto 26 nodes by putting all of the "A's" on node 1, and so forth. It is now up to application logic to direct every query and update to the correct node. 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 inside a transaction to multiple shards, then it is the responsibility of the application to somehow guarantee data consistency across nodes.
- Node failures become more common as the system scales. It is a hard problem to maintain consistent replicas, detect failures, fail over to replicas, and replace failed nodes in a running system.
- Making schema changes without taking shards "off line" is a challenge.
- Reprovisioning the hardware to add additional nodes or change the configuration is extremely tedious. Again, this is much harder if the shards cannot be take offline.

As such, many developers of these sharded web applications are in severe pain because they must perform these functions in application-level logic. Much of the NoSQL movement seems to be directed at this pain point. However, with the large number of new systems and the wide range of approaches they take, it can be very difficult for clients to understand and choose a system to meet their application requirements.

In this paper, we present 10 rules, which we would advise any client to consider who has an SO application and is examining non-GPTRS systems. The rules are a mix of DBMS requirements and guidelines concerning good SO application design. Moreover, we state these rules in the context of clients running software in their own environment; however most of the rules also apply to software-as-a-service (SaaS) environments.

In each case, we present the rule and then indicate why we believe it is necessary.

Rule #1: Look for shared-nothing scalability

There are three hardware architectures on which a DBMS can run. The oldest one, 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 will be 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, multi-core systems will face performance limitations with DBMS software. Clients who chose an SMP system were forced to perform sharding themselves in order to obtain scalability across SMP nodes, and they face the painful problems noted above. Popular systems that currently run on SMP configurations are MySQL, PostgreSQL, and Microsoft SQL Server.

The second option is to choose a DBMS that runs on disk clusters. Here, a collection of CPUs with private main memories share a common disk system. This architecture was popularized in the 1980's and 90's by Sun, DEC, and HP, but it has serious scalability problems in a DBMS context. Because there is a private buffer pool in the main memory of each node, it is possible for the same disk block to be in multiple buffer pools. Hence, careful synchronization of these buffer pool blocks is required. Similarly there is a private lock table in the main memory of each node. Again, careful synchronization is

required. Such synchronization issues limit the scalability of a shared disk configuration to a small number of nodes (typically less than 10).

The main example of a DBMS running shared disk is Oracle RAC, and it is difficult to find RAC configurations with a double-digit number of nodes. Recently, Oracle announced Exadata and Exadata 2, which run shared disk at the top level of a two-tier hierarchy, but run shared-nothing at the bottom level. Hence, Exadata is a blended architecture that does not fall neatly into one of our buckets.

The final architecture is a shared-nothing configuration, where each node shares neither main memory nor disk. Rather, a collection of self-contained nodes are connected to each other by networking. Essentially all DBMSs oriented toward the data warehouse market built in the last two decades run shared-nothing, including Greenplum, Vertica, Asterdata, Paraccel, Netezza, and Teradata. Moreover, DB2 runs shared-nothing, as do many NoSQL engines . Shared-nothing engines normally perform automatic sharding (partitioning) of data to achieve parallelism. Shared-nothing systems will only scale if data objects are partitioned across the nodes in the system in a manner that balances the load. If there is data skew or "hot spots", then a shared-nothing system will degrade in performance to the speed of the overloaded node. In addition, the application must make the overwhelming majority of transactions "single-sharded", a point we discuss 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 systems are reported to run a hundred nodes or more. BigTable is reported to run on thousands of nodes.

The DBMS needs of web applications may drive DBMS scalability upward in a hurry. For example, Facebook is presently sharding 4000 MySQL instances in application logic. If they choose to consider 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. Hence, shared-nothing DBMSs are the only game in town.

Rule #2: High-level languages are good and need not hurt performance

The work in a SQL transaction may include the following five pieces:

a) overhead resulting from the optimizer choosing an inferior execution plan
b) overhead of communicating with the DBMS
c) overhead inherent in coding in a high-level language
d) overhead for services such as concurrency control, crash recovery and data integrity
e) truly useful work, which must be performed no matter what

We discuss the first three components in this rule, leaving the last two for the next rule.

In the 1960's and 1970's, hierarchical and network systems were the dominant DBMS solutions, offering a low-level procedural interface to data. The high-level language of RDBMSs were instrumental in displacing these DBMSs because:

- A high-level language system requires the programmer to write less code that is easier to understand.
- A user states what he wants instead of writing a disk-oriented algorithm on how to access the data he needs. A programmer does not need to understand complex storage optimizations.
- 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 of the charges leveled at RDBMSs in the 1970s and 80's was that they would not be as efficient as low-level systems. The claim was made that automatic query optimizers could not do as good a job as smart programmers. Although early optimizers were primitive, they quickly became as good as all but the best human programmers. Moreover, most clients cannot attract and retain this level of talent. Hence, this source of overhead has largely disappeared, and is only an issue on very complex queries, which are rarely found in SO applications.

The second source of overhead is communicating with the DBMS. Currently, RDBMSs insist on the application being run in a separate address space for security reasons, and use 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, anybody seriously interested in performance runs transactions using a stored procedure interface, rather than SQL commands over ODBC/JDBC. In this case, a transaction is a single over-and-back message. The DBMS can further reduce communication overhead by batching multiple transactions in one call. In summary, 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.

Third, there is the overhead of coding in SQL rather than 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. Put differently, standard language compilation converts a high level specification into a very efficient low-level run-time executable.

Hence, one of the key lessons in the DBMS field during the last quarter of a century is that high level languages are good and do not hurt performance. Some of the new systems do provide SQL or a more limited higher-level language. Others provide only a "database assembly language" – individual index and object operations. For very simple applications this may be adequate, but in all other cases we believe that a high-level language provides compelling advantages.

Rule #3: Plan to carefully leverage main memory databases

Consider a cluster of 16 nodes, each with 64 Gbytes of main memory. Any shared-nothing DBMS thereby has access to about 1 Tbyte of main memory. Such a hardware configuration would have been considered extreme a few years ago, but is now commonplace. Moreover, memory per node will obviously increase in the future, and the number of nodes in a cluster is also likely to rise. Hence, typical clusters of the future will have increasing terabytes of main memory.

As a result, if your database is a couple of terabytes or less (a very large SO database), you should consider main memory deployment. If your database is larger you should consider main memory deployment when practical in the future. In addition, flash memory and other technologies have become a promising alternative to disk 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 will be achieved simply by running a DBMS on a machine with more memory.

To understand why, consider the CPU overhead in DBMSs. In [6] we measured performance using part of one of the major SO benchmarks, TPC-C, on the Shore open source DBMS. This DBMS was chosen because the source code was available for instrumentation, and because it is a typical GPTRS implementation. From simple measures of some other GPTRS systems, we believe the Shore results are representative of them as well.

We used a database size that allowed all data to fit in main memory, since that is consistent with most SO applications. Since Shore, like other GPTRS systems, is disk-based, that means all of the data resides in the main memory buffer pool. Our goal was to categorize DBMS overhead on TPC-C. We ensured that a good query plan was chosen by the optimizer, and ran the DBMS in the same address space as the application driver, thereby avoiding any TCP/IP cost. We then looked at the components of CPU usage, which perform useful work or deliver DBMS services.

In [6] some shortcomings of the Shore B-tree implementation are noted, which have been fixed in most commercial GPTRS implementations. Therefore, the following results were scaled to assume removal of this source of overhead. We report actual cycles used, rather than CPU instruction counts, for the new-order transaction on TPC-C:

> **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 image and after image of the change must be written to a log. Shore then groups transactions together in a "group commit" that forces the relevant portions of the log to disk. The CPU cost of this activity is noted here.

Buffer pool overhead: 33%. Since all data resides in the buffer pool, any retrievals or updates require the relevant block to be found in the buffer pool. Then, the appropriate record(s) must be located and relevant attributes in the record found. Blocks on which there is an open database cursor must be "pinned" in main memory. Moreover, an LRU or other replacement algorithm will be used, requiring additional information to be recorded.

Multi-threading overhead: 11%. Since most DBMSs are multi-threaded, there are multiple operations going on in parallel. Unfortunately, the lock table is a shared data structure, which 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.

The interested reader is directed to [6] for a more detailed discussion, including a commentary on why the latching overhead may be understated.

Clearly a conventional disk-based DBMS spends the overwhelming majority of its cycles on overhead activity. To go a *lot* faster, one must deal with *all* of the overhead components noted above. For example, a main memory DBMS with conventional multi-threading, locking and recovery will be only marginally faster than its disk-based counterpart. Put differently, a NoSQL or other database engine will not dramatically outperform a GPTRS implementation, unless:

1. all of these overhead components are addressed, or

2. the GPTRS solution has not been properly architected, for example by using conversational SQL rather than a stored procedure interface.

Of course, we are looking at single-machine performance in our analysis, but this will have a direct effect on the multi-machine scalability discussed in Rule 1 and our other rules.

Rule #4: High availability (HA) and automatic recovery are essential for SO scalability

A quarter of a century ago, a typical DBMS application would run on what we would now consider very expensive hardware. If the hardware failed, the client would restore working hardware, reload the operating system and DBMS, and then recover the database to the state of the last completed transaction by performing an undo of incomplete

transactions and a redo of completed ones, using a DBMS log. This process could take a while (several minutes to an hour or more), and the application would be unavailable for this period of time.

Few clients today are willing to accept down time in their SO applications. Instead, most everyone wants to run redundant hardware and use data replication to have a second copy of all objects. On a hardware failure, the system switches over to the backup and continues operation. Effectively, people want "non-stop" operation, as pioneered in the 1980's by Tandem Computers.

Furthermore, many large web properties are running large numbers of shared-nothing nodes in their configurations. In such worlds, the probability of failure rises as the number of "moving parts" increases. Effectively, this 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 (HA), so that non-stop operation can be supported. Three HA caveats should be clearly noted. First, there are a multitude of kinds of failures, including:

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

Any DBMS will continue operation for some but not all of the failure modes above. The cost of recovering from all possible failure modes is very high. Hence, high availability is a statistical effort, namely how much availability is desired against what classes of failures.

The second caveat is the so-called CAP theorem [7]. In the presence of certain of the above failures, this theorem states that you can have any two of: Consistency, Availability, and Partition-tolerance. Hence, there are theoretical limits on what is possible in the high-availability arena.

Lastly, every site administrator wants to guard against disasters (earthquakes, floods, etc.). Although these are rare, continued operation is still desirable. Hence, disaster recovery (DR) should be considered as an extension of HA, supported by replication over a wide area network.

Rule #5: On-line everything

An SO DBMS should have a single state: "up". From a user's point of view, it should never fail and should never have to be taken offline. In addition to failure recovery just discussed, we need to consider operations that require the database to 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 that are used to process transactions without interruption in service. For example a configuration might go from 10 nodes to 15 to accommodate an increase in load.
- Software upgrade: It should be possible to move from version I of a DBMS to version I + 1 without interruption of service.

Obviously some of the above capabilities are challenging to support. However, 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 interventions are simply not practical.

Rule #6: Avoid multi-node operations

Two things are necessary to achieve SO scalability over a cluster of servers:

1. The database and application load must be split evenly over the servers. Read-scalability can be simply achieved by replicating data, but general read/write scalability requires sharding (partitioning) the data over nodes according to a primary key.
2. Applications must rarely perform operations that span more than one server or shard. If a large number of servers are involved in processing an operation the scalability advantages may be lost, either because of redundant work, cross-server communication or required operation synchronization.

For example, suppose you have an employee table, and partition it on employee age. If you want to know the salary of a specific employee, you must then send the query to all nodes, requiring a slew of messages. Only one will find the desired data; the others will run a redundant query that finds nothing. Furthermore, if you perform an update that crosses shards, for example giving a raise to all employees in the shoe department, then you must pay all of the synchronization overhead of ensuring that the transaction is performed on every node.

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

The percentage of single node transactions can be further increased 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, deleted or change their address very infrequently. Hence, complete replication will allow inserting the address of a customer into a new purchase order as a single node operation. Therefore, selective replication of read-mostly data can be advantageous.

In summary: avoid multi-shard operations to the greatest extent possible. This includes queries that must go to multiple shards, and also multi-shard updates requiring ACID properties. Carefully think through your application and database design to accomplish this goal. If this goal is unachievable with your current application design, then 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. They provide one or more of the following mechanisms:

- Some create new versions of objects on every write, resulting in parallel versions when there are multiple asynchronous writes. It is up to application logic to resolve the resulting conflict.
- Some provide an "update if current" operation, changing an object only if it matches a specified value. In this way, an application can read an object that it plans to later update and then make changes only if the value is still current.
- Some provide ACID semantics, but only for read and write operations of a single object, attribute, or shard.
- Some provide "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, with enough additional code. However, this is a difficult task we wouldn't wish 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 the application level.

Any operation that requires coordinated updates to two objects is likely to need ACID guarantees. For example, consider a transaction that moves $10 between two user accounts. With an ACID system, one 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 examples requiring ACID semantics would include charging a customer's account only if their order ships, or synchronously updating bilateral "friend" references. Standard

ACID semantics give you the all-or-nothing guarantee you need to maintain data integrity in these cases. Although there are some applications that do not need such coordination right now, a commitment to a non-ACID system precludes extending such an application 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 their 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 multi-node 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. It is a no-brainer to let the DBMS do it.

We have also seen arguments to abandon ACID transactions based on the CAP theorem [7], which states that you can only have two of C: consistency, A: availability, P: partition-tolerance. The argument is that partitions happen; hence one must abandon consistency to get high availability. We take issue with this argument for three reasons. First, some applications really need consistency, and cannot give it up. Second, the CAP theorem only deals with a subset of possible failures, as noted in Rule 4, and one is left with the problem of coping with the rest. 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.

It is true that WAN partitions are much more likely than LAN partitions. However, WAN replication is normally used for read-only copies or disaster recovery, e.g. an entire data center going offline; WAN latency is too high for synchronous replication or sharding. Few users expect to recover from major disasters without short availability "hiccups". As such, the CAP theorem may be less relevant to this situation.

In summary, we advise clients who have a *need* for ACID to seek a DBMS that *provides* ACID rather than coding it themselves, and keep the overhead of distributed transactions to a minimum 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-box" behavior. Most products have numerous tuning knobs that can adjust 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 who is unskilled in the given product.

As such, it is a daunting task to bring in a new DBMS, especially one that is distributed over many nodes. This 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 is a several week task, even though code and schema are

readily available. Moreover, once one has an application in production, it still requires substantial DBA resources to keep it running.

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

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

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; node performance is less important. It is true that linear scalability is important, but we believe it a big mistake to ignore node performance. One should always remember that linear scalability means that overall performance is a multiple of the number of nodes times node performance. The faster the node performance; the less nodes you will need.

It is quite 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 order of magnitude [8]. Similarly H-store (the prototype predecessor to VoltDB) has been shown to have even higher throughput on TPC-C when compared to the products from major vendors [9].

Consider an example in which a customer is choosing between two solutions, each offering linear scalability. If solution A offers node performance that is a factor of 20 better than solution B, the customer might require 50 hardware nodes with solution A, versus 1000 nodes with solution B.

Obviously, this is a non-trivial difference in hardware cost, rack space, cooling and power consumption between the two solutions. More importantly, if a node fails on average every 3 years, then solution B will see a failure every day, while solution A will see failures 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.

In short, node performance makes everything else easier.

Rule #10: Open source gives you more control over your future

This last rule is not a technical point, but we felt it important to mention. Hence, perhaps this should be a suggestion rather than a rule. The landscape is littered with situations where a client acquired a vendor's product, only to face expensive upgrades in later

years, large maintenance bills for often inferior technical support, and an inability to avoid these fees because the cost of switching to a different product required extensive recoding. The best way we see 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 doing these in-house.

For these reasons many newer web-oriented shops are adamant about using only open source systems. Also, several vendors have proved that it is possible to make a viable business with an open source model. As such, we expect it will become more popular over time, and clients would be well advised to consider its advantages.

Summary

In this paper we have presented 10 rules which we believe specify the desirable properties for any SO data store. Clients looking at distributed data storage solutions would be well advised to look at systems they are considering in the context of our rule set as well as their unique application requirements. There are a large number of systems now available, and they range considerably in their capabilities and limitations.

References

[1] Codd, E. F., "A Relational Model of Data for Large Shared Databanks," CACM, June 1970.

[2] Astrahan, M. M. et. al., "System R: A Relational Approach to Data Management," ACM TODS, June 1976.

[3] Stonebraker, M. et. al., "The Design and Implementation of Ingres," ACM TODS, Sept. 1976.

[4] Selinger, P., "Access Path Selection in a Relational Data Management System," Proc. 1979 ACM SIGMOD Conference on Management of Data, Boston, Ma., June 1979.

[5] http://en.wikipedia.org/wiki/ACID

[6] Harizopoulos, S. et. al., "OLTP: Through the Looking Glass and What We Found There," Proc. 2008 SIGMOD Conference on Management of Data, Vancouver, B. C., June 2008.

[7] Eric Brewer, "Towards Robust Distributed Systems," http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[8] Abadi, D. et. al., " A Comparison of Approaches to Large Scale Data Analysis," Proc. 2009 SIGMOD Conference on Management of Data, Providence, RI, June 2009. (follow on paper in January 2010, CACM, along with a response by Hadoop advocates)

[9] Stonebraker, M. et. al., "The End of an Architectural Era (It's Time for a Complete Rewrite)", Proc. 2007 VLDB Conference, Vienna, Austria, Sept. 2007.

[10] Cattell, R., "High Performance Scalable Data Stores," http://cattell.net/datastores/.

Author Affiliations

Michael Stonebraker (stonebraker@csail.mit.edu) is a professor of Electrical Engineering and Computer Science, MIT, consultant and founder, Zetics, Inc, consultant and founder, Goby, Inc., consultant and founder, VoltDB, Inc., and board member, Vertica Systems, Inc.

Rick Cattell (rick@cattell.net) is consultant and principal, Cattell.Net, and consultant and technical advisor board member, Schooner Information Technologies.

SIDEBAR: System References

	Sources for more information
Asterdata	asterdata.com
BigTable	labs.google.com/papers/bigtable.html
Clustrix	clustrix.com
CouchDB	couchdb.apache.org
DB2	ibm.com/software/data/db2
Dynamo	portal.acm.org/citation.cfm?id=1294281
Exadata	oracle.com/exadata
Greenplum	greenplum.com
Hadoop	hadoop.apache.org
HBase	hbase.apache.org
HyperTable	hypertable.org
MongoDB	mongodb.org
MySQL	mysql.com/products/enterprise
MySQL Cluster	mysql.com/products/database/cluster
Netezza	netezza.com
NimbusDB	nimbusdb.com
Oracle	oracle.com
Oracle RAC	oracle.com/rac
Paraccel	paraccel.com
PNUTs	research.yahoo.com/pub/2304
PostgreSQL	postgresql.org
Riak	basho.com/Riak.html
Scalaris	code.google.com/p/scalaris
SimpleDB	amazon.com/simpledb

SQL Server	microsoft.com/sqlserver
Teradata	teradata.com
Terrastore	code.google.com/p/terrastore
Tokyo Cabinet	1978th.net/tokyocabinet
Vertica	vertica.com
Voldemort	project-voldemort.com
VoltDB	voltdb.com