Article development led by [acmqueue](http://queue.acm.org)
queue.acm.org

How do large-scale sites and applications remain SQL-based?

BY MICHAEL RYS

Scalable SQL

ONE OF THE leading motivators for NoSQL innovation is the desire to achieve very high scalability to handle the vagaries of Internet-size workloads. Yet many big social Web sites (such as Facebook, MySpace, and Twitter) and many other Web sites and distributed tier 1 applications that require high scalability (such as e-commerce and banking) reportedly remain SQL-based^a for their core data stores and services.

The question is, how do they do it?

The main goal of the NoSQL/big data movement is to achieve agility. Among the variety of agility dimensions—such as *model* agility (ease and speed of changing data models), *operational* agility (ease and speed of changing operational aspects), and

^a SQL is actually the name of a declarative query language, while more precisely this article concerns traditional relational database systems. Since it is common to talk about NoSQL as the opposite of relational database systems, we have taken the editorial liberty to use SQL as a synonym for relational database systems.

programming agility (ease and speed of application development)—one of the most important is the ability to quickly and seamlessly scale an application to accommodate large amounts of data, users, and connections. Scalable architectures are especially important for large distributed applications such as social networking sites, e-commerce Web sites, and point-of-sale/branch infrastructures for more traditional stores and enterprises where the scalability of the application is directly tied to the scalability and success of the business.

These applications have several scalability requirements:

- *Scalability in terms of user load.* The application needs to be able to scale to a large number of users, potentially in the millions.
- *Scalability in terms of data load.*

The application must be able to scale to a large amount of data, potentially in petabytes, either produced by a few or produced as the aggregate of many users.

► *Computational scalability.* Operations on the data should be able to scale for both an increasing number of users and increasing data sizes.

► *Scale agility.* In order to scale to increasing or decreasing application load, the architecture and operational environment should provide the ability to add or remove resources quickly, without application changes or impact on the availability of the application.

Scalable Architectures

Several major architectural approaches achieve high-level scalability. Most of them provide scale-out based on some form of functional and/or data

partitioning and distributing the work across many processing nodes.

Functional partitioning often follows the service-oriented paradigm of building the application with several independent services each performing a specific task (see Figure 1). This allows the application to scale out by assigning separate resources to these services as needed. Functional scale-out partitioning alone, however, often does not provide enough scalability since the number of tasks is limited and not in direct relationship to the big drivers of scalability requirements: the number of users and size of data. So functional partitioning is often combined with data partitioning.

Data partitioning distributes the application's processing over a set of data partitions (Figure 2). Different forms of data partitioning are de-

played based on the topology of the processing nodes and the characteristics of the data. For example, if the user base is geographically dispersed and there is a locality requirement for scalability and performance reasons, such as in worldwide social networking sites, then data often is partitioned according to those geographic boundaries. On the other hand, data may be more randomly partitioned—for example, based on customer IDs—if the scale-out requirements are more constrained by the cost of running data-analysis algorithms over the data. In this case, equal partition sizes are more important.

Once an application is built using a distributed model to achieve scale, it will have to deal with a set of requirements above and beyond simple centralized application structures:

► Because of the distribution of both data and processing, the database that in a centralized application model would provide a *consistent* view of the data and transactional execution is now distributed among many databases. Thus, the application (or a middle tier) has to provide an additional transactional/consistency layer to provide consistency across the partitions.

► In addition, changes to the applications have to be rolled out to all the partitions in a way that will not interfere with the consistency guarantees and requirements of the application. For example, if the application issues distributed queries against a set of tables that are partitioned across several nodes, and the application is updating the schema of some of these distributed tables, then either the schema change needs to be backward-compatible so it can be rolled out locally without affecting the ongoing queries, or the schema must be updated globally, thus impacting the application's *availability* during the rollout phase.

► Finally, there is an increased probability of partition node failures and network *partitioning*. Therefore, nodes

need to be made redundant and applications have to be resilient to network partitioning.

Furthermore, all three of these requirements have to be fulfilled without negatively impacting the availability of the application's services, the main reason why the application probably was scaled out in the first place.

In 2000, Eric Brewer made the conjecture that it is impossible for a distributed Web service to provide all three guarantees—consistency, availability, partition tolerance—at the same time.¹ This conjecture is now commonly known as the CAP theorem² and is one of the main arguments why traditional relational database techniques that provide strong ACID guarantees (atomic transactions, transactional consistency and isolation, and data durability) cannot provide both the partition tolerance and availability required by large-scale distributed applications.

So why are many of the leading social networking sites (Facebook, MySpace, Twitter), e-commerce Web sites (hotel reservation systems and shopping sites), and large banking

applications still implemented using traditional database systems such as MySQL (Facebook,³ Twitter⁹) or SQL Server (MySpace,⁷ Choice Hotels International,⁶ Bank Itau⁵) instead of using the new NoSQL systems?

How Do You Scale Out with SQL?

The high-level answer is that the application architecture is still weighing the same trade-offs required by the CAP theorem. Given that the availability of the application has to be guaranteed for business reasons, and that partition and node failures cannot be excluded, the application architecture has to make some compromises around the level of provided consistency. Note this does not mean that relational databases cannot be used per se; it means the strong consistency guarantee of a single partition node cannot be made across all nodes and that the application architecture cannot use “traditional” database technologies such as distributed querying, full ACID transactions, and synchronous processing of requests without running into availability and scalability issues.

For example, traditional distributed query engines such as Microsoft SQL Server's linked servers assume close coupling of the data sources and are not able to adjust to quickly changing topologies—whether because of nodes being added or because of node failures. They operate synchronously and will wait for nodes to reply or fail the query in case of a node failure, thus impacting availability of the service.

What are some of the ways to build scalable applications using relational database systems as their underlying data stores? Basically the application architectures follow the same service-oriented, functional- and data-partitioning schemes outlined previously. Each leaf partition will be using a relational database, providing local consistency and query processing. To guarantee node availability, each node will be mirrored and made highly available. Depending on the service-level guarantee around failover and read versus update frequency, each mirror will be managed either synchronously or asynchronously.

Global consistency across the many locally consistent nodes will be provided to the level that the application

Figure 1. Functional partitioning of a commerce system.

Figure 2. Data partitioning of a commerce system with partitioning keys.

requires, most often relaxing the atomicity, strong consistency, and/or isolation of the global operation. Many techniques exist, such as open nested transaction systems (Saga,⁴ multilevel concurrency control¹⁰) and optimistic concurrency control approaches, and specific partitioning and application logics to reduce the risk of inconsistencies. For example, open nested multilevel transactions relax transactional isolation by allowing certain local changes to become globally visible before the global transaction commits. This increases transactional throughput at the risk of potentially costly compensation work when a global transaction and its impact have to be undone. Thus, the openness often is restricted to specific operations that are commutative and have a clearly defined compensating action. In practice, such advanced transaction models have not yet been widely used, even though some transaction managers provide them.

More frequently, the application partitions data in a first step to avoid local conflicts and then uses optimistic approaches that assume that conflicts rarely occur. This approach takes into account the idea that most people are in fact fine with eventually consistent states of the global data.

Accepting short-term “incorrect” global states and results is actually pretty common in our day-to-day lives. Even bank transactions are often “eventually consistent.” For example, redeeming a check or settling an investment transaction will not be fully consistent at a global level at the time the transaction is executed. The money will potentially go into the seller’s account before it gets deducted from the buyer’s account, but there is a guarantee that the money will eventually be deducted and the global state will become consistent.

Using eventual consistency is a more complex application design paradigm than assuming a globally consistent state at all times. The programmer has to determine the acceptable level of inconsistencies—how long the data can be kept in an inconsistent state. The platform provider has to design the system in a way that programmers can easily understand the possible inconsistencies and provide mechanisms

Besides providing a scalable architecture, Service Broker provides a communication fabric guaranteeing that messages to a service are delivered reliably, in order, and exactly once.

to handle them when they appear. Often the agility and scalability gains are worth the additional complexity of the application architecture.

Using eventual consistency as an acceptable global consistency guarantee also allows the application to provide availability during network failures and thus achieve higher scalability. On the one hand, updating a node that has become unavailable will no longer block or fail the global transaction, as long as the system can guarantee that it will eventually be updated. On the other hand, eventual consistency allows the application to operate on older data and still provide useful results; sometimes it even allows partial results if a node cannot be queried (although this is a decision the application has to make). It also means that the architecture can be built using asynchronous services that will provide for higher scalability because the functional services and individual data partitions can do their work without blocking the application.

An Example of How to Scale with SQL

As we already mentioned, several applications with high scalability requirements are being built on top of traditional relational database systems. For example, Twitter uses the NoSQL database Cassandra for some of its needs, but its core database system that manages tweets is still using the MySQL relational database system.⁹

The following example presents a high-level overview of how MySpace achieves scalability of its architecture using Microsoft SQL Server. MySpace is still one of the largest social networking sites. In 2009 it used 440 SQL Server instances to manage 130 million users and one petabyte of data with 4.4 million concurrently active users at peak time.⁷

As outlined earlier, MySpace has chosen to use both functional and data partitioning. Data partitions are geographically distributed to be closer to the users in an area, as well as becoming further partitioned by user IDs for scale. This makes sense since most users will want to access their own data most frequently. Obviously, since MySpace is a social networking site where individual users connect and

leave messages and comments, operations not only target a single partition, but also need to update data across partitions. Given the large demands on availability and scalability, MySpace needs to achieve a balance between scale and correctness.

The basic approach is to perform most of the work in an asynchronous fashion. The asynchronous processing of the change events and interactions with the application provides high availability, and by having the partitions operate on the queued requests in a uniform fashion, the system is able to scale out easily. Using a reliable message infrastructure provides the guarantee that the changes eventually become visible, thus delivering eventual correctness.

Figure 3 provides a high-level abstraction of MySpace's service dispatcher architecture. It consists of a few dozen request routers that dispatch incoming requests to perform a certain user or system action—for example, posting a comment on a friend's picture, submitting a blog entry, or a system request such as deploying a new schema object. During steady state, the request routers are exact copies of each other, including a routing table mapping services to partitions.

The requests are asynchronously distributed across the routers and get dispatched to the individual account partitions (around 440 in the case of MySpace) and the requested service endpoint. Note that the account partitions provide all the same services and schemata at steady state, thus guaranteeing that every service can be provided by every node without being dependent on any other node.

Each of the routers and each of the partitions and services are implemented using SQL Server and SQL Server Service Broker. Service Broker is the key ingredient that enables this architecture to work reliably and efficiently. It provides the asynchronous messaging capabilities that allow the requests to flow at a high rate between the services. Each service exposes a queue to accept requests and the ability to dispatch workers on each item in the queue. Service Broker, like other service-bus and asynchronous messaging components, allows scaling out by simply adding multiple instances of the same service across different partitions. Requests are load balanced across these service instances without having to change the application logic. An interesting difference to some of the other message buses such as MQ-

Series, RabbitMQ, NServiceBus, and Microsoft Message Queuing (MSMQ) is that Service Broker is deeply built into the database engine.

Besides providing a scalable architecture, Service Broker provides a communication fabric guaranteeing messages to a service are delivered reliably, in order, and exactly once. This guarantees that even in case of a network partition or a node failure, a message is not lost but will eventually be delivered once the node has been reconnected. Since every service will be performed by the database server, local consistency is provided at the level specified for the specific transaction. The use of Service Broker to build and scale the services will provide global eventual consistency.

The availability of each partition can be improved by providing a failover copy using database mirroring. If a failover occurs, the Service Broker connection also automatically and transparently fails over.

The application scale-out architecture as described avoids a single point of failure and contention by replicating all the routing information across all the request routers and providing the services on all partitions. The asynchronous processing using Service Broker provides scalability, as well as eventual consistency. The architecture, services, and partitioning, however, will evolve over time. Therefore, the changes to the routing information when data gets repartitioned and the updates to services and schemas also need to be maintained in a scalable way. It would not be good if a global lock is being taken across all the request routers when adding a new partition to the routing table.

To address this, the current architecture uses the same Service Broker-based approach to roll out changes to the services and schemas. A repartition of the account services will be updated asynchronously. To detect a change in the partition by a router before its routing table has been updated, the partitions will fail a request if the partition assumption is invalid and will provide updated information back to the router, which then retries the request based on the new routing information.

A similar architecture is also being used for several e-commerce Web sites

Figure 3. High-level diagram of a scalable service dispatch architecture using SQL Server Service Broker.

that build on relational databases. For example, Bank Itau provides a scalable branch banking system⁵ and Choice Hotels International has a highly scalable online hotel reservation system⁶ using asynchronous messaging.

Summary and Outlook

Building scalable database applications is not necessarily a question of whether one should use a relational database system or a NoSQL system. It is more a question of choosing the right application architecture that is agile enough to scale. For example, combining asynchronous messaging with a relational database system provides the powerful infrastructure that enables developers to build highly scalable, agile applications that provide partition tolerance and availability while providing a high level of eventual consistency.

Scale-out applications with SQL are being built using similar architectural principles as scale-out applications using NoSQL while providing more mature infrastructure for declarative query processing, optimizations, indexing, and data storage/high availability. In addition, scaling out an existing SQL application without having to replace the data tier with a different database system that has different configuration, management, and troubleshooting requirements is very appealing.

Other aspects such as data models, agility requirements, query optimization, data-processing logic, existing infrastructures, and individual capabilities, strengths, and weaknesses will have to be considered as well when deciding between a SQL and NoSQL database system. Discussing these aspects are unfortunately outside the scope of this article.

All database systems, however, whether relational or NoSQL, still need to provide additional services that make it easier for the developer to build massively scalable applications. For example, relational database systems should add integrated support for data-partitioning scale-out such as sharding.⁸ NoSQL databases are working on providing more of the traditional database capabilities such as secondary indices, declarative query languages, among others.

Until the database systems provide simple-to-use scale-out services, de-

All database systems, be they relational or NoSQL, still need to provide additional services that make it easier for the developer to build massively scalable applications.

velopers will have to design their applications with scale-out in mind and use more generic application patterns such as asynchronous messaging, functional and data partitioning, and fault tolerance to build fault-resilient systems that provide high availability and scalability.

Acknowledgments. I would like to express my gratitude to Erik Meijer, Luis Vargas, and Terry Coatta for their reviews and insightful comments that greatly improved this article.

Related articles on queue.acm.org

Data in Flight

Julian Hyde

<http://queue.acm.org/detail.cfm?id=1667562>

Cybercrime 2.0: When the Cloud Turns Dark

Niels Provos, Moheeb Abu Rajab,

Panayiotis Mavrommatis

<http://queue.acm.org/detail.cfm?id=1517412>

Beyond Relational Databases

Margo Seltzer

<http://queue.acm.org/detail.cfm?id=1059807>

References

1. Brewer, E. A. Towards robust distributed systems. (Invited talk) Principles of Distributed Computing (Portland, OR, July 2000).
2. CAP; http://en.wikipedia.org/wiki/CAP_theorem.
3. Facebook; <http://blog.facebook.com/blog.php?post=7899307130>.
4. Garcia-Molina, H., Salem, K. Sagas. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. 249–259; <http://www.informatik.uni-trier.de/~ley/db/conf/sigmod/sigmod87.html#Garcia-MolinaS87>.
5. Helland, P., Stelzmueller, C. SQLCAT: SQL service broker: high-performance distributed applications in real-world deployments. PASS Summit (2009); <http://www.softconference.com/pass/sessionDetail.asp?SID=174551>.
6. Microsoft Case Studies. Global hotel company delivers reservations in milliseconds with highly reliable system (2011); <http://www.microsoft.com/casestudies/Microsoft-SQL-Server-Service-Broker/Choice-Hotels-International/Global-Hotel-Company-Delivers-Reservations-in-Milliseconds-with-Highly-Reliable-System/4000009199>.
7. Microsoft Case Studies. MySpace uses SQL Server Service Broker to protect integrity of 1 petabyte of data (2009); http://www.microsoft.com/Casestudies/Case_Study_Detail.aspx?casestudyid=4000004532.
8. MSDN Blogs; <http://blogs.msdn.com/b/cbiyikoglu/archive/tags/federations/>.
9. Twitter; <http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html>.
10. Weikum, G. A theoretical foundation of multilevel concurrency control. *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (1986), 31–43.

Michael Rys (mrys@microsoft.com) is principal program manager on the SQL Server RDBMS team at Microsoft. He is responsible for the Beyond Relational Data and Services scenario that includes unstructured and semi-structured data management, search, Spatial, XML, and others. He represents Microsoft Corp. in the W3C XML Query working group and the ANSI SQL standardization effort. For more information, see http://sqlblog.com/blogs/michael_rys/default.aspx or follow Rys at @SQLServerMike (when he finds time to tweet).

© 2011 ACM 0001-0782/11/06 \$10.00