Distributed Database Systems 101

Or, Distributed Databases - what the FK does 'web scale' actually mean?

Distributed database systems are complex critters and come in a number of different flavours. If I dig deep in to the depths of my dimly remembered distributed systems papers I did at university (roughly 15 years ago) I'll try to explain some of the key engineering problems to building a distributed database system.

First, some terminology

ACID (Atomicity, Consistency, Isolation and Durability) properties: These are the key invariants that have to be enforced for a transaction to be reliably implemented without causing undesirable side effects.

Atomicity requires that the transaction complete or rollback completely. Partially finished transactions should never be visible, and the system has to be built in a way that prevents this from happening.

Consistency requires that a transaction should never violate any invariants (such as declarative referential integrity) that are guaranteed by the database schema. For example, if a foreign key exists it should be impossible to insert a child record with a reverence to a non-existent parent.

Isolation requires that transactions should not interfere with each other. The system should guarantee the same results if the transactions are executed in parallel or sequentially. In practice most RDBMS products allow modes that trade off isolation against performance.

Durability requires that once committed, the transaction remains in persistent storage in a way that is robust to hardware or software failure.

I'll explain some of the technical hurdles these requirements present on distributed systems below.

Shared Disk Architecture: An architecture in which all processing nodes in a cluster have access to all of the storage. This can present a central bottleneck for data access. An example of a shared-disk system is Oracle RAC or Exadata.

Shared Nothing Architecture: An architecture in which processing nodes in a cluster have local storage that is not visible to other cluster nodes. Examples of shared-nothing systems are Teradata andNetezza.

Shared Memory Architecture: An architecture in which multiple CPUs (or nodes) can access a shared pool of memory. Most modern servers are of a shared memory type. Shared memory facilitates certain operations such as caches or atomic synchronisation primitives that are much harder to do on distributed systems.

Synchronisation: A generic term describing various methods for ensuring consistent access to a shared resource by multiple processes or threads. This is much harder to do on distributed systems than on shared memory systems, although some network architectures (e.g. Teradata's BYNET) had synchronisation primitives in the network protocol. Synchronisation can also come with a significant amount of overhead.

Semi-Join: A primitive used in joining data held in two different nodes of a distributed system. Essentially it consists of enough information about the rows to join being bundled up and passed by one node to the other in order to resolve the join. On a large query this could involve significant network traffic.

Eventual Consistency: A term used to describe transaction semantics that trade off immediate update (consistency on reads) on all nodes of a distributed system for performance (and therefore higher transaction throughput) on writes. Eventual consistency is a side effect of using Quorum Replication as a performance optimisation to speed up transaction commits in distributed databases where multiple copies of data are held on separate nodes.
Lamport's Algorithm: An algorithm for implementing mutual exclusion (synchronisation) across systems with no shared memory. Normally mutual exclusion within a system requires an atomic read-compare-write or similar instruction of a type normally only practical on a shared memory system. Other distributed synchronisation algorithms exist, but Lamport's was one of the first and is the best known. Like most distributed synchronisation mechanisms, Lamport's algorithm is heavily dependent on accurate timing and clock synchronisation beteen cluster nodes.
Two Phase Commit (2PC): A family of protocols that ensure that database updates involving multiple physical systems commit or roll back consistently. Whether 2PC is used within a system or across multiple systems via a transaction manager it carries a significant overhead.
In a two-phase commit protocol the transaction manager asks the participating nodes to persist the transaction in such a way that they can guarantee that it will commit, then signal this status. When all nodes have returned a 'happy' status it then signals the nodes to commit. The transaction is still regarded as open until all of the nodes send a reply indicating the commit is complete. If a node goes down before signalling the commit is complete the transaction manager will re-query the node when it comes back up until it gets a positive reply indicating the transaction has committed.

Multi-Version Concurrency Control (MVCC): Managing contention by writing new versions of the data to a different location and allowing other transactions to see the old version of the data until the new version is committed. This reduces database contention at the expense of some additional write traffic to write the new version and then mark the old version as obsolete.
Election Algorithm: Distributed systems involving multiple nodes are inherently less reliable than a single system as there are more failure modes. In many cases some mechanism is needed for clustered systems to deal with failure of a node. Election algorithms are a class of algorithms used to select a leader to coordinate a distributed computation in situations where the 'leader' node is not 100% determined or reliable.
Horizontal Partitioning: A table may be split across multiple nodes or storage volumes by its key. This allows a large data volume to be split into smaller chunks and distributed across storage nodes.
Sharding: A data set may be horizontally partitioned across multiple physical nodes in a shared-nothing architecture. Where this partitioning is not transparent (i.e. the client must be aware of the partition scheme and work out which node to query explicitly) this is known as sharding. Some systems (e.g. Teradata) do split data across nodes but the location is transparent to the client; the term is not normally used in conjunction with this type of system.
Consistent Hashing: An algorithm used to allocate data to partitions based on the key. It is characterised by even distribution of the hash keys and the ability to elastically expand or reduce the number of buckets efficiently. These attributes make it useful for partitioning data or load across a cluster of nodes where the size can change dynamically with nodes being added or dropping off the cluster (perhaps due to failure).
Multi-Master Replication: A technique that allows writes across multiple nodes in a cluster to be replicated to the other nodes. This technique facilitates scaling by allowing some tables to be partitioned or sharded across servers and others to be synchronised across the cluster. Writes must be replicated to all nodes as opposed to a quorum, so transaction commits are more expensive on a multi-master replicated architecture than on a quorum replicated system.

Non-Blocking Switch: A network switch that uses internal hardware parallelism to achieve throughput that is proportional to the number of ports with no internal bottlenecks. A naive implementation can use a crossbar mechanism, but this has O(N^2) complexity for N ports, limiting it to smaller switches. Larger switches can use more a complex internal topology called a non-blocking minimal spanning switch to achieve linear throughput scaling without needing O(N^2) hardware.

Making a distributed DBMS - how hard can it be?

Several technical challenges make this quite difficult to do in practice. Apart from the added complexity of building a distributed system the architect of a distributed DBMS has to overcome some tricky engineering problems.

Atomicity on distributed systems: If the data updated by a transaction is spread across multiple nodes the commit/rollback of the nodes must be coordinated. This adds a significant overhead on shared-nothing systems. On shared-disk systems this is less of an issue as all of the storage can be seen by all of the nodes so a single node can coordinate the commit.
Consistency on distributed systems: To take the foreign key example cited above the system must be able to evaluate a consistent state. For example, if the parent and child of a foreign key relationship could reside on different nodes some sort of distributed locking mechanism is needed to ensure that outdated information is not used to validate the transaction. If this is not enforced you could have (for example) a race condition where the parent is deleted after the its presence is verified before allowing the insert of the child.
Delayed enforcement of constraints (i.e. waiting until commit to validate DRI) requires the lock to be held for the duration of the transaction. This sort of distributed locking comes with a significant overhead.

If multiple copies of data are held (this may be necessary on shared-nothing systems to avoid unnecessary network traffic from semi-joins) then all copies of the data must be updated.

Isolation on distributed systems: Where data affected on a transaction resides on multiple system nodes the locks and version (if MVCC is in use) must be synchronised across the nodes. Guaranteeing serialisability of operations, particularly on shared-nothing architectures where redundant copies of data may be stored requires a distributed synchronisation mechanism such as Lamport's Algorithm, which also comes with a significant overhead in network traffic.
Durability on distributed systems: On a shared disk system the durability issue is essentially the same as a shared-memory system, with the exception that distributed synchronisation protocols are still required across nodes. The DBMS must journal writes to the log and write the data out consistently. On a shared-nothing system there may be multiple copies of the data or parts of the data stored on different nodes. A two-phase commit protocol is needed to ensure that the commit happens correctly across the nodes. This also incurs significant overhead.
On a shared-nothing system the loss of a node can mean data is not available to the system. To mitigate this data may be replicated across more than one node. Consistency in this situation means that the data must be replicated to all nodes where it normally resides. This can incur substantial overhead on writes.

One common optimisation made in NoSQL systems is the use of quorum replication and eventual consistency to allow the data to be replicated lazily while guaranteeing a certain level of resiliency of the data by writing to a quorum before reporting the transaction as committed. The data is then replicated lazily to the other nodes where copies of the data reside.

Note that 'eventual consistency' is a major trade-off on consistency that may not be acceptable if the data must be viewed consistently as soon as the transaction is committed. For example, on a financial application an updated balance should be available immediately.

Shared-Disk systems

A shared-disk system is one where all of the nodes have access to all of the storage. Thus, computation is independent of location. Many DBMS platforms can also work in this mode - Oracle RAC is an example of such an architecture.

Shared disk systems can scale substantially as they can support a M:M relationship between storage nodes and processing nodes. A SAN can have multiple controllers and multiple servers can run the database. These architectures have a switch as a central bottleneck but crossbar switches allow this switch to have a lot of bandwidth. Some processing can be offloaded onto the storage nodes (as in the case of Oracle's Exadata) which can reduce the traffic on the storage bandwidth.

Although the switch is theoretically a bottleneck the bandwidth available means that shared-disk architectures will scale quite effectively to large transaction volumes. Most mainstream DBMS architectures take this approach because it affords 'good enough' scalability and high reliability. With a redundant storage architecture such as fibre channel there is no single point of failure as there are at least two paths between any processing node and any storage node.

Shared-Nothing systems

Shared-nothing systems are systems where at least some of the data is held locally to a node and is not directly visible to other nodes. This removes the bottleneck of a central switch, allowing the database to scale (at least in theory) with the number of nodes. Horizontal partitioning allows the data to be split across nodes; this may be transparent to the client or not (see Sharding above).

Because the data is inherently distributed a query may require data from more than one node. If a join needs data from different nodes a semi-join operation is used to transfer enough data to support the join from one node to another. This can result in a large amount of network traffic, so optimising the distribution of the data can make a big difference to query performance.

Often, data is replicated across nodes of a shared-nothing system to reduce the necessity for semi-joins. This works quite well on data warehouse appliances as the dimensions are typically many orders of magnitude smaller than the fact tables and can be easily replicated across nodes. They are also typically loaded in batches so the replication overhead is less of an issue than it would be on a transactional application.

The inherent parallelism of a shared-nothing architecture makes them well suited to the sort of table-scan/aggregate queries characteristic of a data warehouse. This sort of operation can scale almost linearly with the number of processing nodes. Large joins across nodes tend to incur more overhead as the semi-join operations can generate lots of network traffic.

Moving large data volumes is less useful for transaction processing applications, where the overhead of multiple updates makes this type of architecture less attractive than a shared disk. Thus, this type of architecture tends not to be used widely out of data warehouse applications.

Sharding, Quorum Replication and Eventual Consistency

Quorum Replication is a facility where a DBMS replicates data for high availability. This is useful for systems intended to work on cheaper commodity hardware that has no built-in high-availability features like a SAN. In this type of system the data is replicated across multiple storage nodes for read performance and redundant storage to make the system resilient to hardware failure of a node.

However, replication of writes to all nodes is O(M x N) for M nodes and N writes. This makes writes expensive if the write must be replicated to all nodes before a transaction is allowed to commit. Quorum replication is a compromise that allows writes to be replicated to a subset of the nodes immediately and then lazily written out to the other nodes by a background task. Writes can be committed more quickly, while providing a certain degree of redundancy by ensuring that they are replicated to a minimal subset (quorum) of nodes before the transaction is reported as committed to the client.

This means that reads off nodes outside the quorum can see obsolete versions of the data until the background process has finished writing data to the rest of the nodes. The semantics are known as 'Eventual Consistency' and may or may not be acceptable depending on the requirements of your application but mean that transaction commits are closer to O(1) than O(n) in resource usage.

Sharding requires the client to be aware of the partitioning of data within the databases, often using a type of algorithm known as 'consistent hashing'. In a sharded database the client hashes the key to determine which server in the cluster to issue the query to. As the requests are distributed across nodes in the cluster there is no bottleneck with a single query coordinator node.

These techniques allow a database to scale at a near-linear rate by adding nodes to the cluster. Theoretically, quorum replication is only necessary if the underlying storage medium is to be considered unreliable. This is useful if commodity servers are to be used but is of less value if the underlying storage mechanism has its own high availability scheme (for example a SAN with mirrored controllers and multi-path connectivity to the hosts).

For example, Google's BigTable does not implement Quorum Replication by itself, although it does sit on GFS, a clustered file system that does use quorum replication. BigTable (or any shared-nothing system) could use a reliable storage system with multiple controllers and partition the data among the controllers. Parallel access would then be achieved through partitioning of the data.

Back to RDBMS platforms

There is no inherent reason that these techniques could not be used with a RDBMS. However lock and version management would be quite complex on such a system and any market for such a system is likely to be quite specialised. None of the mainstream RDBMS platforms use quorum replication and I'm not specifically aware of any RDBMS product (at least not one with any significant uptake) that does.

Shared-disk and shared-nothing systems can scale up to very large workloads. For instance, Oracle RAC can support 63 processing nodes (which could be large SMP machines in their own right) and an arbitrary number of storage controllers on the SAN. An IBM Sysplex (a cluster of zSeries mainframes) can support multiple mainframes (each with substantial processing power and I/O bandwidth of their own) and multiple SAN controllers. These architectures can support very large transaction volumes with ACID semantics, although they do assume reliable storage. Teradata, Netezza and other vendors make high-performance analytic platforms based on shared-nothing designs that scale to extremely large data volumes.

So far, the market for cheap but ultra-high volume fully ACID RDBMS platforms is dominated by MySQL, which supports sharding and multi-master replication. MySQL does not use quorum replication to optimise write throughput, so transaction commits are more expensive than on a NoSQL system. Sharding allows very high read throughputs (for example Facebook uses MySQL extensively), so this type of architecture scales well on read-heavy workloads.

An interesting debate

BigTable is a shared-nothing architecture (essentially a distributed key-value pair) as pointed out by Michael Hausenblas below. My original evaluation of it included the MapReduce engine, which is not a part of BigTable but would normally be used in conjunction with it in its most common implementations (e.g. Hadoop/HBase and Google's MapReduce framework).
Comparing this architecture with Teradata, which has physical affinity between storage and processing (i.e. the nodes have local storage rather than a shared SAN) you could argue that BigTable/MapReduce is a shared disk architecture through the globally visible parallel storage system.

The processing throughput of a MapReduce style system such as Hadoop is constrained by the bandwidth of a non-blocking network switch.[1] Non-blocking switches can, however, handle large bandwidth aggregates due to the parallelism inherent in the design, so they are seldom a significant practical constraint on performance. This means that a shared disk architecture (perhaps better referred to as a shared-storage system) can scale to large workloads even though the network switch is theoretically a central bottleneck.
The original point was to note that although this central bottleneck exists in shared-disk systems, a partitioned storage subsystem with multiple storage nodes (e.g. BigTable tablet servers or SAN controllers) can still scale up to large workloads. A non-blocking switch architecture can (in theory) handle as many current connections as it has ports.

[1] Of course the processing and I/O throughput available also constitutes a limit on performance but the network switch is a central point through which all traffic passes.

Source:

http://dba.stackexchange.com/questions/34892/why-cant-rdbms-cluster-the-way-nosql-does