Distributed Algorithms in NoSQL Databases

Posted on September 18, 2012

9

Scalability is one of the main drivers of the NoSQL movement. As such, it encompasses distributed system coordination, failover, resource management and many other capabilities. It sounds like a big umbrella, and it is. Although it can hardly be said that NoSQL movement brought fundamentally new techniques into distributed data processing, it triggered an avalanche of practical studies and real-life trials of different combinations of protocols and algorithms. These developments gradually highlight a system of relevant database building blocks with proven practical efficiency. In this article I'm trying to provide more or less systematic description of techniques related to distributed operations in NoSQL databases.

In the rest of this article we study a number of distributed activities like replication of failure detection that could happen in a database. These activities, highlighted in bold below, are grouped into three major sections:

- Data Consistency. Historically, NoSQL paid a lot of attention to tradeoffs between consistency, fault-tolerance and performance to serve geographically distributed systems, low-latency or highly available applications. Fundamentally, these tradeoffs spin around data consistency, so this section is devoted **data replication** and **data repair**.
- Data Placement. A database should accommodate itself to different data distributions, cluster topologies and hardware configurations. In this section we discuss how to **distribute or rebalance data** in such a way that failures are handled rapidly, persistence guarantees are maintained, queries are efficient, and system resource like RAM or disk space are used evenly throughout the cluster.
- System Coordination. Coordination techniques like **leader election** are used in many databases to implements fault-tolerance and strong data consistency. However, even decentralized databases typically track their global state, **detect failures and topology changes**. This section describes several important techniques that are used to keep the system in a coherent state.

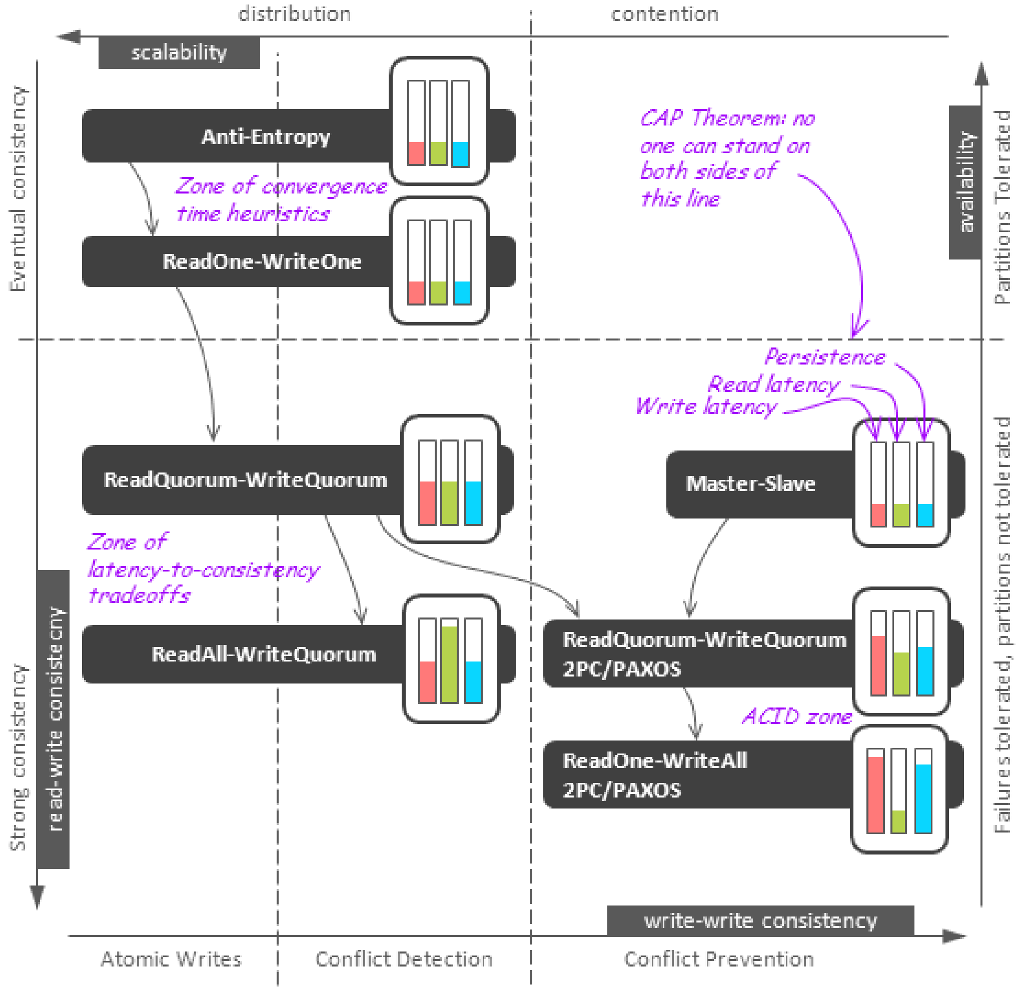
Data Consistency

It is well known and fairly obvious that in geographically distributed systems or other environments with probable network partitions or delays it is not generally possible to maintain high availability without sacrificing consistency because isolated parts of the database have to operate independently in case of network partition. This fact is often referred to as the CAP theorem. However, consistency is a very expensive thing in distributed systems, so it can be traded not only to availability. It is often involved into multiple tradeoffs. To study these tradeoffs, we first note that consistency issues in distributed systems are induced by the replication and the spatial separation of coupled data, so we have to start with goals and desired properties of the replication:

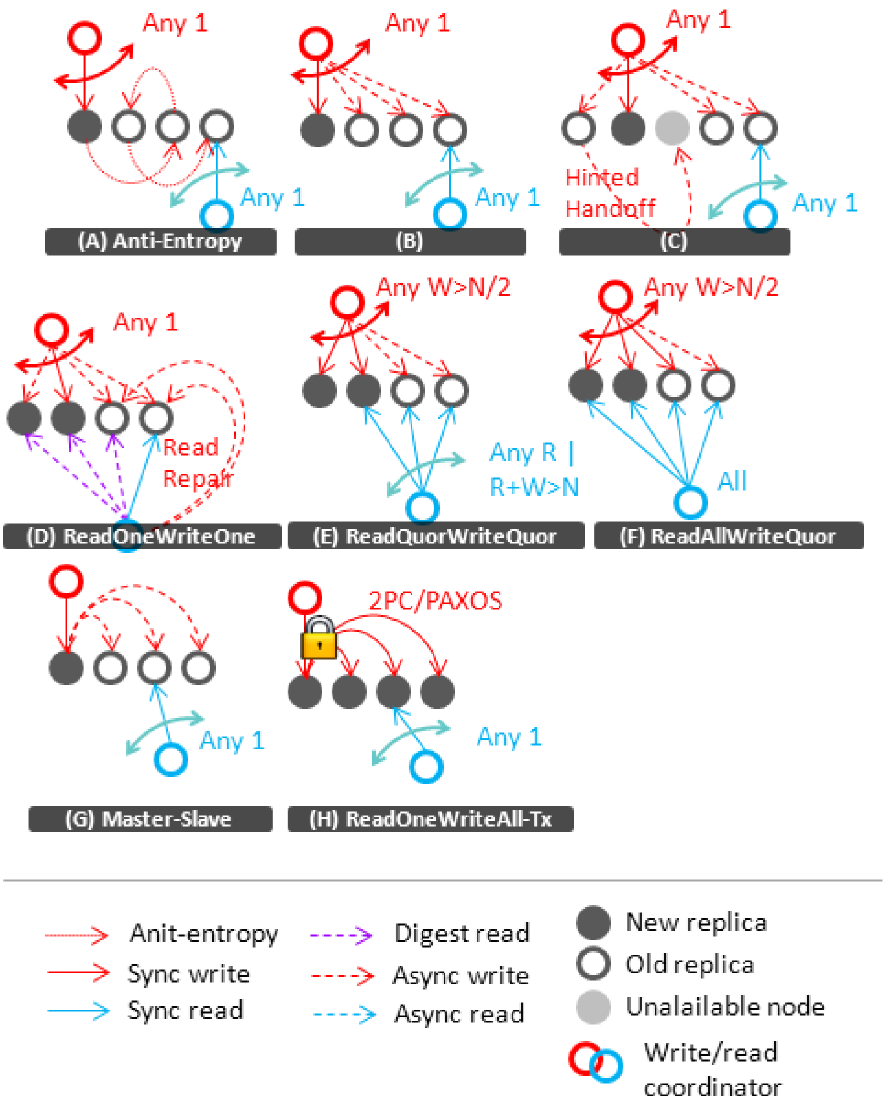
- Availability. Isolated parts of the database can serve read/write requests in case of network partition.
- Read/Write latency. Read/Write requests are processes with a minimal latency.
- Read/Write scalability. Read/Write load can be balanced across multiple nodes.
- Fault-tolerance. Ability to serve read/write requests does not depend on availability of any particular node.
- Data persistence. Node failures within certain limits do not cause data loss.

- Consistency. Consistency is a much more complicated property than the previous ones, so we have to discuss different options in detail. It beyond this article to go deeply into theoretical consistency and concurrency models, so we use a very lean framework of simple properties.
  - Read-Write consistency. From the read-write perspective, the basic goal of a database is to minimize a replica convergence time (how long does it take to propagate an update to all replicas) and guarantee eventual consistency. Besides these weak guarantees, one can be interested in stronger consistency properties:
    - Read-after-write consistency. The effect of a write operation on data item X, will always be seen by a successive read operation on X.
    - Read-after-read consistency. If some client reads the value of a data item X, any successive read operation on X will always return that same or a more recent value.
  - Write-Write consistency. Write-write conflicts appear in case of database partition, so a database should either handle these conflicts somehow or guarantee that concurrent writes will not be processed by different partitions. From this perspective, a database can offer different consistency models:
    - Atomic Writes. If a database provides an API where a write request can only be an independent atomic assignment of a value, one possible way to avoid write-write conflicts is to pick the "most recent" version of each entity. This guarantees that all nodes will end up with the same version of data irrespectively to the order of updates which can be affected by network failures and delays. Data version can be specified by a timestamps or application-specific metric. This approach is used for example in Cassandra.
    - Atomic Read-modify-write. Applications often do a read-modify-write sequence instead of independent atomic writes. If two clients read the same version of data, modify it and write back concurrently, the latest update will silently override the first one in the atomic writes model. This behavior can be semantically inappropriate (for example, if both clients add a value to a list). A database can offer at least two solutions:
      - Conflict prevention. Read-modify-write can be thought as a particular case of transaction, so distributed locking or consensus protocols like PAXOS [20, 21] are both a solution. This is a generic technique that can support both atomic read-modify-write semantics and arbitrary isolated transactions. An alternative approach is to prevent distributed concurrent writes entirely and route all writes of a particular data item to a single node (global master or shard master). To prevent conflicts, a database must sacrifice availability in case of network partitioning and stop all but one partition. This approach is used in many systems with strong consistency guarantees (e.g. most RDBMSs, HBase, MongoDB).
      - Conflict detection. A database track concurrent conflicting updates and either rollback one of the conflicting updates or preserve both versions for resolving on the client side. Concurrent updates are typically tracked by using vector clocks [19] (which can be though as a generalization of the optimistic locking) or by preserving an entire version history. This approach is used in systems like Riak, Voldemort, CouchDB.

Now let's take a closer look at commonly used replication techniques and classify them in accordance with the described properties. The first figure below depicts logical relationships between different techniques and their coordinates in the system of the consistency-scalability-availability-latency tradeoffs. The second figure illustrates each technique in detail.



(http://highlyscalable.files.wordpress.com/2012/09/consistency-plot-3.png)

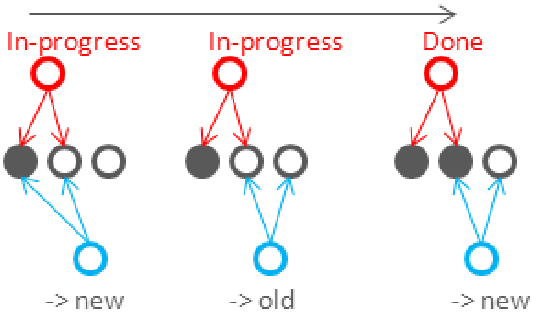


(http://highlyscalable.files.wordpress.com/2012/09/consistency-catalog.png)
Replication factor 4. It is assumed that read/write coordinator can be either an external client or a proxy node within a database.

Let's go through all these techniques moving from weak to strong consistency guarantees:

- (A, Anti-Entropy) Weakest consistency guarantees are provided by the following strategy. Writer updates any arbitrary selected replica. Reader reads any replica and sees the old data until a new version is not propagated via background anti-entropy protocol (more on anti-entropy protocols in the next section). The main properties of this approach are:
  - High propagation latency makes it quite impractical for data synchronization, so it is typically used only as an auxiliary background process that detects and repairs unplanned inconsistencies. However, databases like Cassandra use anti-entropy as a primary way to propagate information about database topology and other metadata.
  - Consistency guarantees are poor: write-write conflicts and read-write discrepancies are very probable even in absence of failures.
  - Superior availability and robustness against network partitions. This schema provides good performance because individual updates are replaced by asynchronous batch processing.
  - Persistence guarantees are weak because new data are initially stored on a single replica.
- (B) An obvious improvement of the previous schema is to send an update to all (available) replicas asynchronously as soon as the update request hits any replica. It can be considered as a kind of targeted anti-entropy.
  - In comparison with pure anti-entropy, this greatly improves consistency with a relatively small performance penalty. However, formal consistency and persistence guarantees remain the same.
  - If some replica is temporary unavailable due to network failures or node failure/replacement, updates should be eventually delivered to it by the anti-entropy process.

- (C) In the previous schema, failures can be handled better using the hinted handoff technique [8]. Updates that are intended for unavailable nodes are recorded on the coordinator or any other node with a hint that they should be delivered to a certain node as soon as it will become available. This improves persistence guarantees and replica convergence time.
- (D, Read One Write One) Since the carrier of hinted handoffs can fail before deferred updates were propagated, it makes sense to enforce consistency by so-called read repairs. Each read (or randomly selected reads) triggers an asynchronous process that requests a digest (a kind of signature/hash) of the requested data from all replicas and reconciles inconsistencies if detected. We use term ReadOne-WriteOne for combination of techniques A, B, C and D – they all do not provide strict consistency guarantees, but are efficient enough to be used in practice as an self-contained approach.
- (E, Read Quorum Write Quorum) The strategies above are heuristic enhancements that decrease replicas convergence time. To provide guarantees beyond eventual consistency, one has to sacrifice availability and guarantee an overlap between read and write sets. A common generalization is to write synchronously W replicas instead of one and touch R replicas during reading.
  - First, this allows one to manage persistence guarantees setting W>1.
  - Second, this improves consistency for R+W>N because synchronously written set will overlap with the set that is contacted during reading (in the figure above W=2, R=3, N=4), so reader will touch at least one fresh replica and select it as a result. This guarantees consistency if read and write requests are issued sequentially (e.g. by one client, read-your-writes consistency), but do not guarantee global read-after-read consistency. Consider an example in the figure below to see why reads can be inconsistent. In this example R=2, W=2, N=3. However, writing of two replicas is not transactional, so clients can fetch both old and new values until writing is not completed:



(http://highlyscalable.files.wordpress.com/2012/09/consistency-concurrent-quorum.png)

  - Different values of R and W allows to trade write latency and persistence to read latency and vice versa.
  - Concurrent writers can write to disjoint quorums if W<=N/2. Setting W>N/2 guarantees immediate conflict detection in Atomic Read-modify-write with rollbacks model.
  - Strictly speaking, this schema is not tolerant to network partitions, although it tolerates failures of separate nodes. In practice, heuristics like sloppy quorum [8] can be used to sacrifice consistency provided by a standard quorum schema in favor of availability in certain scenarios.

- (F, Read All Write Quorum) The problem with read-after-read consistency can be alleviated by contacting all replicas during reading (reader can fetch data or check digests). This ensures that a new version of data becomes visible to the readers as soon as it appears on at least one node. Network partitions of course can lead to violation of this guarantee.
- (G, Master-Slave) The techniques above are often used to provide either Atomic Writes or Read-modify-write with Conflict Detection consistency levels. To achieve a Conflict Prevention level, one has to use a kind of centralization or locking. A simplest strategy is to use master-slave asynchronous replication. All writes for a particular data item are routed to a central node that executes write operations sequentially. This makes master a bottleneck, so it becomes crucial to partition data into independent shards to be scalable.
- (H, Transactional Read Quorum Write Quorum and Read One Write All) Quorum approach can also be reinforced by transactional techniques to prevent write-write conflicts. A well-known approach is to use two-phase commit protocol. However, two-phase commit is not perfectly reliable because coordinator failures can cause resource blocking. PAXOS commit protocol [20, 21] is a more reliable alterative, but with a price or performance penalty. A small step forward and we end up with the Read One Write All approach where writes update all replicas in a transactional fashion. This approach provides strong fault-tolerant consistency but with

a price of performance and availability.

It is worth noting that the analysis above highlights a number of tradeoffs:

- **Consistency-availability tradeoff**. This strict tradeoff is formalized by the CAP theorem. In case of network partition, a database should either stop all partitions except one or accept the possibility of data conflicts.
- **Consistency-scalability tradeoff**. One can see that even read-write consistency guarantees impose serious limitations on a replica set scalability, and write-write conflicts can be handled in a relatively scalable fashion only in the Atomic Writes model. The Atomic Read-modify-write model introduces short casual dependencies between data and this immediately requires global locking to prevent conflicts. This shows that *even a slight spatial or casual dependency between data entries or operations could kill scalability*, so separation of data into independent shards and careful data modeling (http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/) is extremely important for scalability.
- **Consistency-latency tradeoff**. As it was shown above, there exists a tendency to Read-All and Write-All techniques when strong consistency or persistence guarantees are provides by a database. These guarantees are clearly in inverse proportion to requests latency. Quorum techniques are a middle ground.
- **Failover-consistency/scalability/latency tradeoff**. It is interesting that contention between failover and consistency/scalability/latency is not really severe. Failures of up to N/2 nodes can often be tolerated with reasonable performance/consistency penalty. However, this tradeoff is visible, for example, in the difference between 2-phase commit and PAXOS protocols. Another example of this tradeoff is ability to lift certain consistency guarantees like read-your-writes using sticky sessions which complicate failover [22].

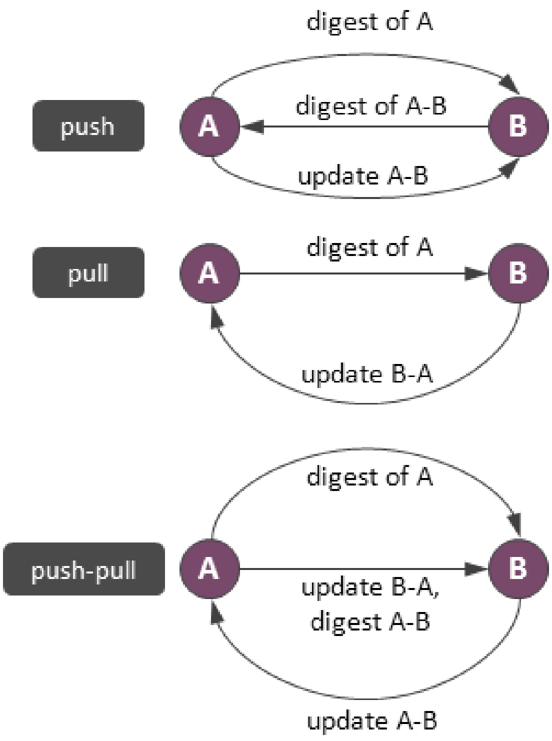
Anti-Entropy Protocols, Gossips

Let us start our study with the following problem statement:

There is a set of nodes and each data item is replicated to a subset of nodes. Each node serves update requests even if there is no network connection to other nodes. Each node periodically synchronizes its state with other nodes is such a way that if no updates take place for a long time, all replicas will gradually become consistent. How this synchronization should be organized – when synchronization is triggered, how a peer to synchronize with is chosen, what is the data exchange protocol? Let us assume that two nodes can always merge their versions of data selecting a newest version or preserving both versions for further application-side resolution.

This problem appears both in data consistency maintenance and in synchronization of a cluster state (propagation of the cluster membership information and so on). Although the problem above can be solved by means of a global coordinator that monitors a database and builds a global synchronization plan or schedule, decentralized databases take advantage of more fault-tolerant approach. The main idea is to use well-studied epidemic protocols [7] that are relatively simple, provide a pretty good convergence time, and can tolerate almost any failures or network partitions. Although there are different classes of epidemic algorithms, we focus on anti-entropy protocols because of their intensive usage in NoSQL databases.

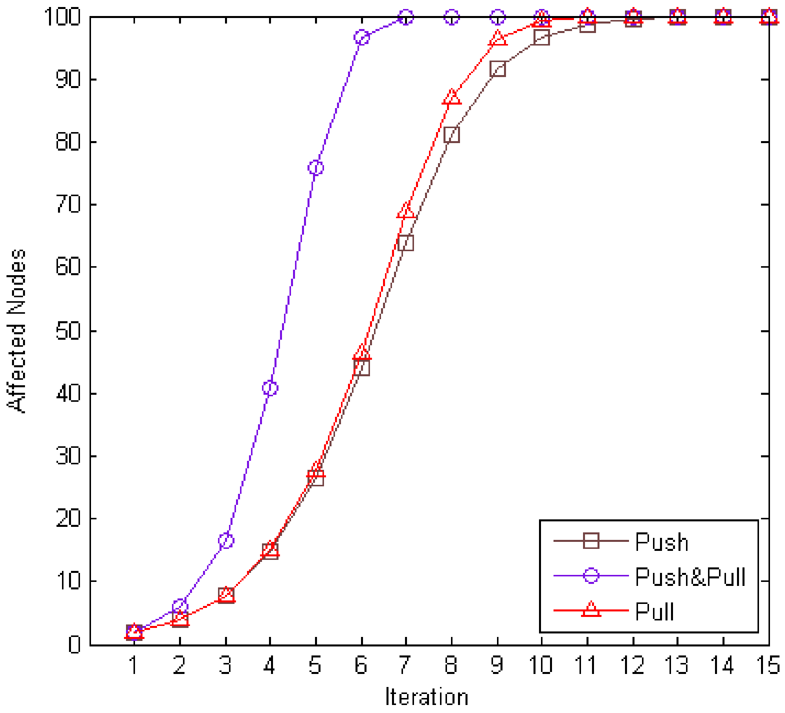
Anti-entropy protocols assume that synchronization is performed by a fixed schedule – every node regularly chooses another node at random or by some rule and exchanges database contents, resolving differences. There are three flavors of anti-entropy protocols: push, pull, and push-pull. The idea of the push protocol is to simply select a random peer and push a current state of data to it. In practice, it is quite silly to push the entire database, so nodes typically work in accordance with the protocol which is depicted in the figure below.



(http://highlyscalable.files.wordpress.com/2012/09/gossips.png)

Node A which is initiator of synchronization prepares a digest (a set of checksums) which is a fingerprint of its data. Node B receives this digest, determines the difference between the digest and its local data and sends a digest of the difference back to A. Finally, A sends an update to B and B updates itself. Pull and push-pull protocols work similarly, as it shown in the figure above.

Anti-entropy protocols provide reasonable good convergence time and scalability. The following figure shows simulation results for propagation of an update in the cluster of 100 nodes. On each iteration, each node contacts one randomly selected peer.



(http://highlyscalable.files.wordpress.com/2012/09/epidemic-dynamics.png)

One can see that the pull style provides better convergence than the push, and this can be proven theoretically [7]. Also, push has a problem with a "convergence tail" when a small percent of nodes remains unaffected during many iterations, although almost all nodes are already touched. The Push-Pull approach greatly improves efficiency in comparison with the original push or pulls techniques, so it is typically used in practice. Anti-entropy is scalable because the average conversion time grows as a logarithmic function of the cluster size.

Although these techniques look pretty simple, there are many studies [5] regarding performance of anti-entropy protocols under different constraints. One can leverage knowledge of the network topology to replace a random peer selection by a more efficient schema [10]; adjust transmit rates or use advanced rules to select data to be synchronized if the network bandwidth is limited [9]. Computation of digest can also be challenging, so a database can maintain a journal of the recent updates to facilitate digests computing.

Eventually Consistent Data Types

In the previous section we assumed that *two nodes can always merge their versions of data*. However, reconciliation of conflicting updates is not a trivial task and it is surprisingly difficult to make all replicas to converge to a semantically correct value. A well-known example is that deleted items can resurface in the Amazon Dynamo database [8].

Let us consider a simple example that illustrates the problem: a database maintains a logically global counter and each database node can serve increment/decrement operations. Although each node can maintain its own local counter as a single scalar value, but these local counters cannot be merged by simple addition/subtraction. Consider an example: there are 3 nodes A, B, and C and increment operation was applied 3 times, once per node. If A pulls value from B and adds it to the local copy, C pulls from B, C pulls from A, then C ends up with value 4 which is incorrect. One possible way to overcome these issues is to use a data structure similar to vector clock [19] and maintain a pair of counters for each node [1]:

```
class Counter {
   int[] plus
   int[] minus
   int NODE_ID

   increment() {
      plus[NODE_ID]++
   }

   decrement() {
      minus[NODE_ID]++
   }

   get() {
      return sum(plus) – sum(minus)
   }

   merge(Counter other) {
      for i in 1..MAX_ID {
         plus[i] = max(plus[i], other.plus[i])
         minus[i] = max(minus[i], other.minus[i])
      }
   }
}
```

Cassandra uses a very similar approach to provide counters as a part of its functionality [11]. It is possible to design more complex eventually consistent data structures that can leverage either state-based or operation-based replication principles. For example, [1] contains a catalog of such structures that includes:

- Counters (increment and decrement operations)
- Sets (add and remove operations)
- Graphs (addEdge/addVertex, removeEdge/removeVertex operations)
- Lists (insertAt(position) and removeAt(position) operations)

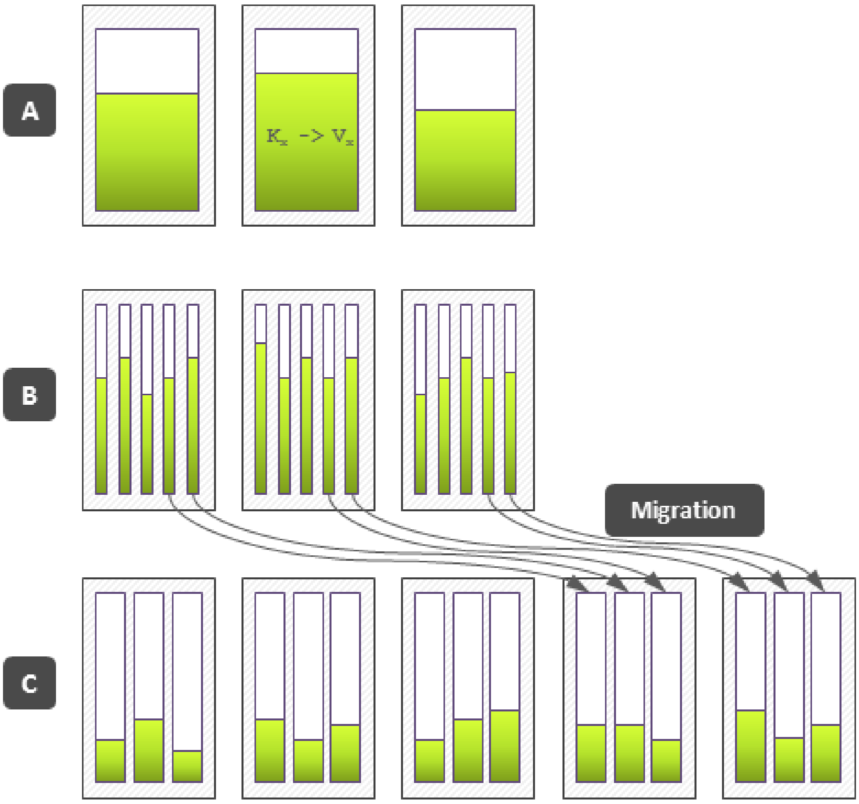
However, eventually consistent data types are often limited in functionality and impose performance overheads.

Data Placement

This section is dedicated to algorithms that control data placement inside a distributed database. These algorithms are responsible for mapping between data items and physical nodes, migration of data from one node to another and global allocation of resources like RAM throughout the database.

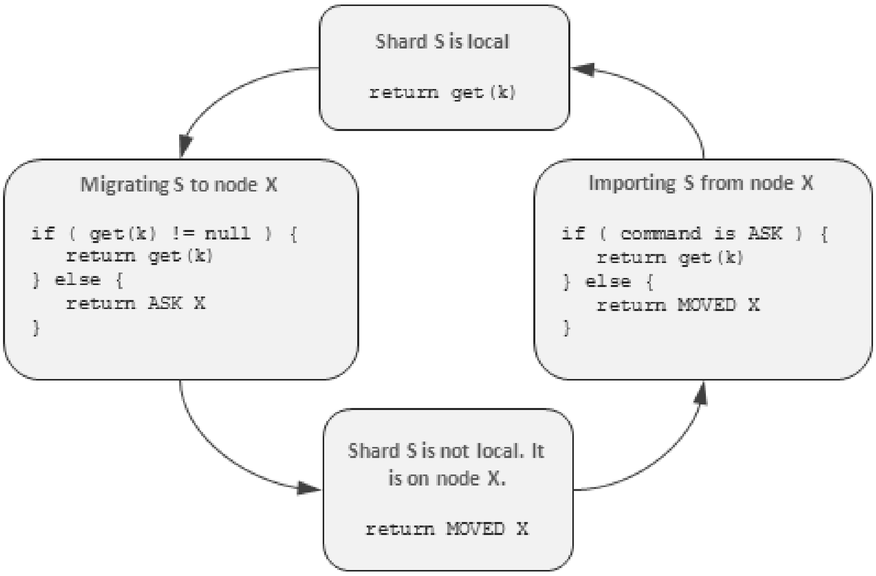
Rebalancing

Let us start with a simple protocol that is aimed to provide outage-free data migration between cluster nodes. This task arises in situations like cluster expansion (new nodes are added), failover (some node goes done), or rebalancing (data became unevenly distributed across the nodes). Consider a situation that is depicted in the section (A) of the figure below – there are three nodes and each node contains a portion of data (we assume a key-value data model without loss of generality) that is distributed across the nodes according to an arbitrary data placement policy:



(http://highlyscalable.files.wordpress.com/2012/09/rebalancing.png)

If one does not have a database that supports data rebalancing internally, he probably will deploy several instances of the database to each node as it is shown in the section (B) of the figure above. This allows one to perform a manual cluster expansion by turning a separate instance off, copying it to a new node, and turning it on, as it is shown in the section (C). Although an automatic database is able to track each record separately, many systems including MongoDB, Oracle Coherence, and upcoming Redis Cluster use the described technique internally, i.e. group records into shards which are minimal units of migration for sake of efficiency. It is quite obvious that a number of shards should be quite large in comparison with the number of nodes to provide the even load distribution. An outage-free shard migration can be done according to the simple protocol that redirects client from the exporting to the importing node during a migration of the shard. The following figure depicts a state machine for get(key) logic as it going to be implemented in Redis Cluster:



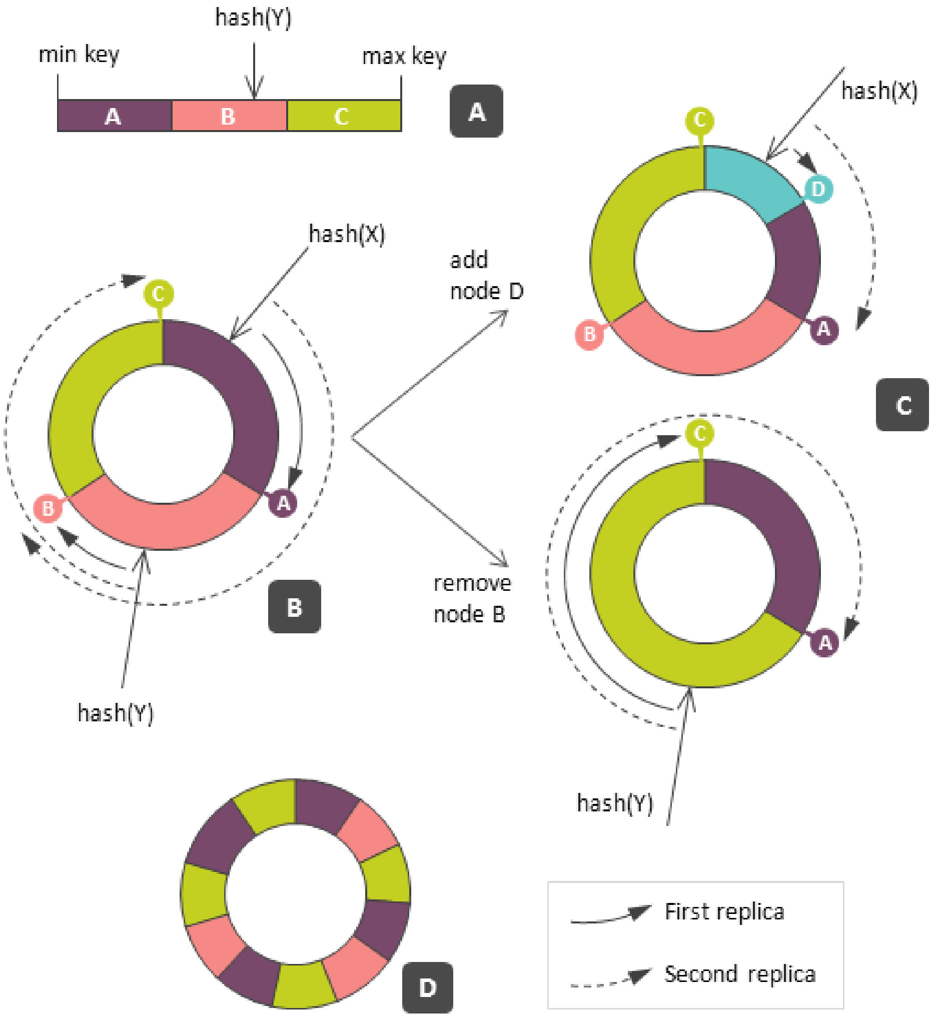
(http://highlyscalable.files.wordpress.com/2012/09/redis-rebalancing-protocol.png)

It is assumed that each node knows a topology of the cluster and is able to map any key to a shard and a shard to a cluster node. If the node determines that the requested key belongs to a local shard, then it looks it up locally (the upper square in the picture above). If the node determines that the requested key belongs to another node X, than it sends a permanent redirection command to the client (the lower square in the figure above). Permanent redirection means that the client is able to cache the mapping between the shard and the node. If the shard migration is in progress, the exporting and the importing nodes mark this shard accordingly and start to move its records locking each record separately. The exporting node first looks up the key locally and, if not found, redirects the client to the importing node assuming that key is already migrated. This redirect is a one-time and should not be cached. The importing node processes redirects locally, but regular queries are permanently redirected until migration is not completed.

Sharding and Replication in Dynamic Environments

The next question we have to address is how to map records to physical nodes. A straightforward approach is to have a table of key ranges where each range is assigned to a node or to use procedures like *NodeID = hash(key) % TotalNodes*. However, modulus-based hashing does not explicitly address cluster reconfiguration because addition or removal of nodes causes complete data reshuffling throughout the cluster. As a result, it is difficult to handle replication and failover.

There are different ways to enhance the basic approach from the replication and failover perspectives. The most famous technique is a consistent hashing. There are many descriptions of the consistent hashing technique in the web, so I provide a basic description just for sake of completeness. The following figure depicts the basic ideas of consistent hashing:

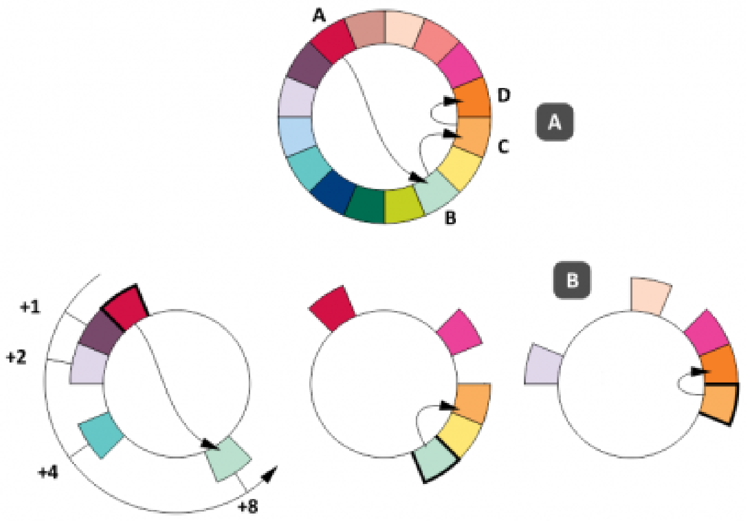


(http://highlyscalable.files.wordpress.com/2012/09/consistent-hashing.png)

Consistent hashing is basically a mapping schema for key-value store – it maps keys (hashed keys are typically used) to physical nodes. A space of hashed keys is an ordered space of binary strings of a fixed length, so it is quite obvious that each range of keys is assigned to some node as it depicted in the figure (A) for 3 nodes, namely, A, B, and C. To cope with replication, it is convenient to close a key space into a ring and traverse it clockwise until all replicas are mapped, as it shown in the figure (B). In other words, item Y should be placed on node B because its key corresponds to B's range, first replica should be placed on C, second replica on A and so on.

The benefit of this schema is efficient addition and removal of a node because it causes data rebalancing only in neighbor sectors. As it shown in the figures (C), addition of the node D affects only item X but not Y. Similarly, removal (or failure) of the node B affects Y and the replica of X, but not X itself. However, as it was pointed in [8], the dark side of this benefit is vulnerability to overloads – all the burden of rebalancing is handled by neighbors only and makes them to replicate high volumes of data. This problem can be alleviated by mapping each node not to a one range, but to a set of ranges, as it shown in the figure (D). This is a tradeoff – it avoids skew in loads during rebalancing, but keeps the total rebalancing effort reasonably low in comparison with module-based mapping.

Maintenance of a complete and coherent vision of a hashing ring may be problematic in very large deployments. Although it is not a typical problem for databases because of relatively small clusters, it is interesting to study how data placement was combined with the network routing in peer-to-peer networks. A good example is the Chord algorithm [2] that trades completeness of the ring vision by a single node to efficiency of the query routing. The Chord algorithm is similar to consistent hashing in the sense that it uses a concept of a ring to map keys to nodes. However, a particular node maintains only a short list of peers with exponentially growing offset on the logical ring (see the picture below). This allows one to locate a key in several network hops using a kind of binary search:



(http://highlyscalable.files.wordpress.com/2012/09/chord.png)

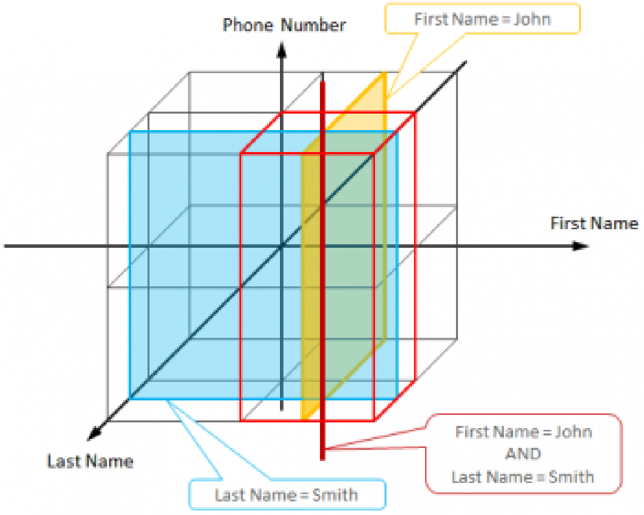
This figure depicts a cluster of 16 nodes and illustrates how node A looks up a key that is physically located on node D. Part (A) depicts the route and part (B) depicts partial visions of the ring for nodes A, B, and C. More information about data replication in decentralized systems can be found in [15].

Multi-Attribute Sharding

Although consistent hashing offers an efficient data placement strategy when data items are accessed by a primary key, things become much more complex when querying by multiple attributes is required. A straightforward approach (that is used, for example, in MongoDB) is to distribute data by a primary key regardless to other attributes. As a result, queries that restrict the primary key can be routed to a limited number of nodes, but other queries have to be processed by all nodes in the cluster. This skew in query efficiency leads us to the following problem statement:

There is a set of data items and each item has a set of attributes along with their values. Is there a data placement strategy that limits a number of nodes that should be contacted to process a query that restricts an arbitrary subset of the attributes?

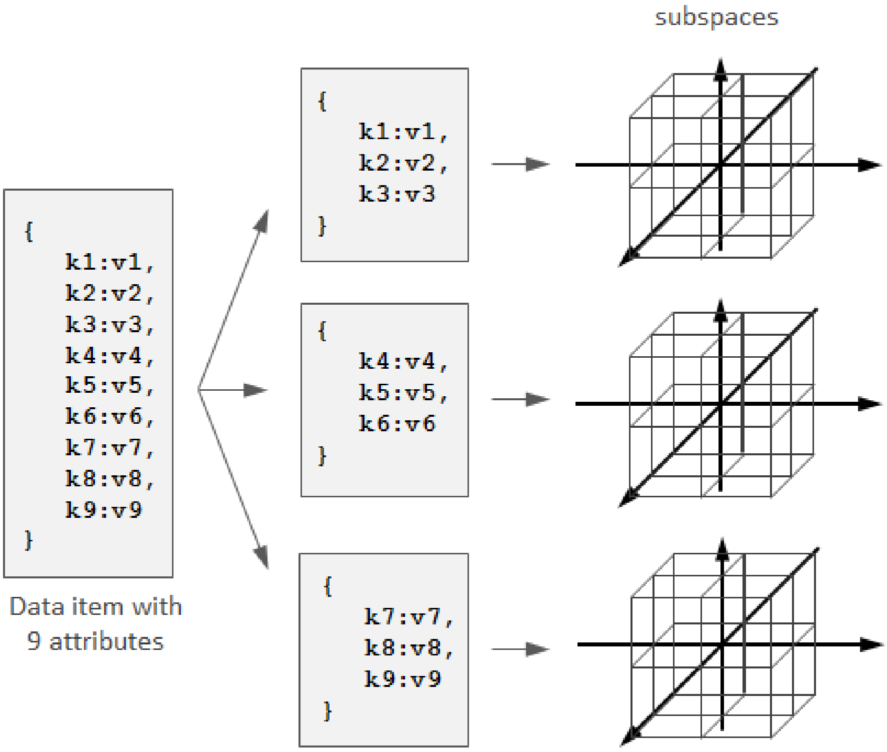
One possible solution was implemented in the HyperDex database. The basic idea is to treat each attribute as an axis in a multidimensional space and map blocks in the space to physical nodes. A query corresponds to a hyperplane that intersects a subset of blocks in the space, so only this subset of blocks should be touched during the query processing. Consider the following example from [6]:



(http://highlyscalable.files.wordpress.com/2012/09/hyperspace-sharding.png)

Each data item is a user account that is attributed by First Name, Last Name, and Phone Number. These attributes are treated as a three-dimensional space and one possible data placement strategy is to map each octant to a dedicated physical node. Queries like "First Name = John" correspond to a plane that intersects 4 octants, hence only 4 nodes should be involved into processing. Queries that restrict two attributes correspond to a line that intersects two octants as it shown in the figure above, hence only 2 nodes should be involved into processing.

The problem with this approach is that dimensionality of the space grows as an exponential function of the attributes count. As a result, queries that restrict only a few attributes tend to involve many blocks and, consequently, involve many servers. One can alleviate this by splitting one data item with multiple attributes into multiple sub-items and mapping them to the several independent subspaces instead of one large hyperspace:


subspaces
{ k1:v1, k2:v2, k3:v3, k4:v4, k5:v5, k6:v6, k7:v7, k8:v8, k9:v9 }
Data item with 9 attributes
{ k1:v1, k2:v2, k3:v3 }
{ k4:v4, k5:v5, k6:v6 }
{ k7:v7, k8:v8, k9:v9 }

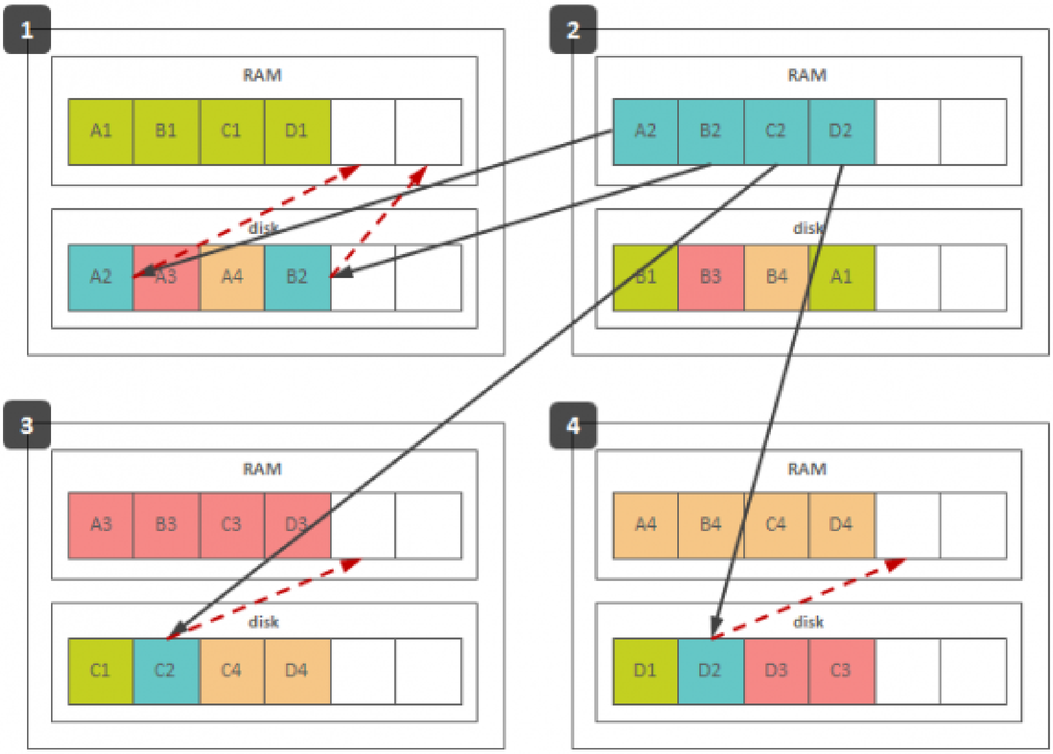
(http://highlyscalable.files.wordpress.com/2012/09/hyperspace-sharding-2.png)

This provides more narrowed query-to-nodes mapping, but complicates coordination because one data item becomes scattered across several independent subspaces with their own physical locations and transactional updates become required. More information about this technique and implementation details can be found in [6].

Passivated Replicas

Some applications with heavy random reads can require all data to fit RAM. In these cases, sharding with independent master-slave replication of each replica (like in MongoDB) typically requires at least double amount of RAM because each chunk of data is stored both on a master and on a slave. A slave should have the same amount of RAM as a master in order to replace the master in case of failure. However, shards can be placed in such a way that amount of required RAM can be reduced, assuming that the system tolerates short-time outages or performance degradation in case of failures.

The following figure depicts 4 nodes that host 16 shards, primary copies are stored in RAM and replicas are stored on disk:



(http://highlyscalable.files.wordpress.com/2012/09/replica-passivation.png)

The gray arrows highlight replication of shards from node #2. Shards from the other nodes are replicated symmetrically. The red arrows depict how the passivated replicas will be loaded into RAM in case of failure of node #2. Even distribution of replicas throughout the cluster allows one to have only a small memory reserve that will be used to activate replicas in case of failure. In the figure above, the cluster is able to survive a single node failure having only 1/3 of RAM in reserve. It is worth noting that replica activation (loading from disk to RAM) takes some time and cause temporally performance degradation or outage of the corresponding data during failure recovery.

System Coordination

In this section we discuss a couple of techniques that relates to system coordination. Distributed coordination is an extremely large area that was a subject of intensive study during several decades. In this article, we, of course, consider only a couple of applied techniques. A comprehensive description of distributed locking, consensus protocols and other fundamental primitives can be found in numerous books or web resources [17, 18, 21].

Failure Detection

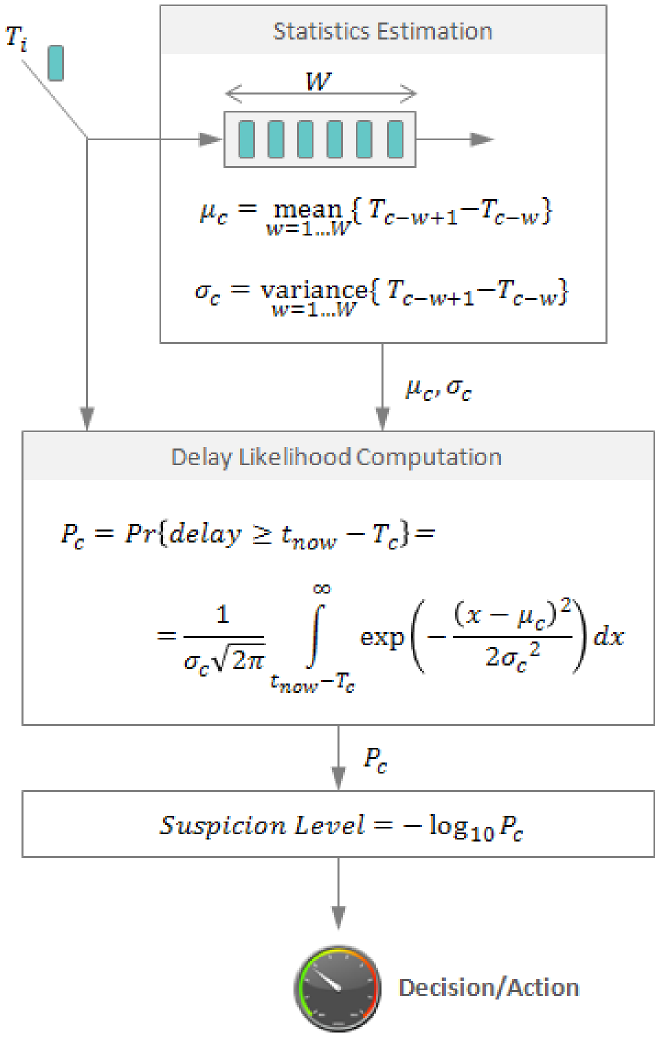
Failure detection is a fundamental component of any fault tolerant distributed system. Practically all failure detection protocols are based on a heartbeat messages which are a pretty simple concept – monitored components periodically send a heartbeat message to the monitoring process (or the monitoring process polls monitored components) and absence of heartbeat messages for a long time is interpreted as a failure. However, real distributed systems impose a number of additional requirements that should be addressed:

- Automatic adaptation. Failure detection should be robust to the temporary network failures and delays, dynamic changes in the cluster topology, workload or bandwidth. This is a fundamentally difficult problem because there is no way to distinguish crashed process from a slow one [13]. As a result, failure detection is

always a tradeoff between a failure detection time (how long does it take to detect a real failure) and the false-alarm probability. Parameters of this tradeoff should be adjusted dynamically and automatically.
- Flexibility. At first glance, failure detector should produce a boolean output, a monitored process considered to be either live or dead. Nevertheless, it can be argued that boolean output is insufficient in practice. Let us consider an example from [12] that resembles Hadoop MapReduce. There is a distributed application that consists of a master and several workers. The master has a list of jobs and submits them to the workers. The master can distinguish different "degrees of failure". If the master starts to suspect that some worker went down, it stops to submit new jobs to this worker. Next, as time goes by and there are no heartbeat messages, the master resubmits jobs that were running on this worker to the other workers. Finally, the master becomes completely confident that the worker is down and releases all corresponding resources.
- Scalability and robustness. Failure detection as a system process should scale up as well as the system does. It also should be robust and consistent, i.e. all nodes in the system should have a consistent view of running and failed processes even in case of communication problems.

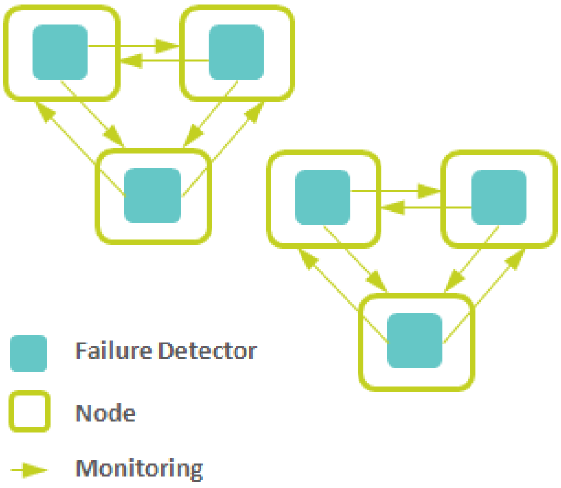
A possible way to address the first two requirements is so-called Phi Accrual Failure Detector [12] that is used with some modifications in Cassandra [16]. The basic workflow is as follows (see the figure below):

- For each monitored resource, Detector collects arrival times Ti of heartbeat messages.
- Mean and variance are constantly computed for the recent arrival times (on a sliding window of size W) in the Statistics Estimation block.
- Assuming that distribution of arrival times is known (the figure below contains a formula for normal distribution), one can compute the probability of the current heartbeat delay (difference between the current time t_now and the last arrival time Tc). This probability is a measure of confidence in a failure. As suggested in [12], this value can be rescaled using the logarithmic function for sake of usability. In this case output 1 means that the likeness of the mistake is about 10%, output 2 means 1% and so on.


T_i
Statistics Estimation
W
$\mu_c = \underset{w=1\ldots W}{\text{mean}}\{T_{c-w+1} - T_{c-w}\}$
$\sigma_c = \underset{w=1\ldots W}{\text{variance}}\{T_{c-w+1} - T_{c-w}\}$
μ_c, σ_c
Delay Likelihood Computation
$P_c = Pr\{delay \geq t_{now} - T_c\} =$
$= \frac{1}{\sigma_c\sqrt{2\pi}} \int_{t_{now}-T_c}^{\infty} \exp\left(-\frac{(x-\mu_c)^2}{2\sigma_c{}^2}\right) dx$
P_c
$Suspicion\ Level = -\log_{10} P_c$
Decision/Action

(http://highlyscalable.files.wordpress.com/2012/09/phi-accrual-failure-detector.png)

The scalability requirement can be addressed in significant degree by hierarchically organized monitoring zones that prevent flooding of the network with heartbeat messages [14] and synchronization of different zones via gossip protocol or central fault-tolerant repository. This approach is illustrated below (there are two zones and all six failure detectors talk to each other via gossip protocol or robust repository like ZooKeeper):



(http://highlyscalable.files.wordpress.com/2012/09/monitoring-zones.png)

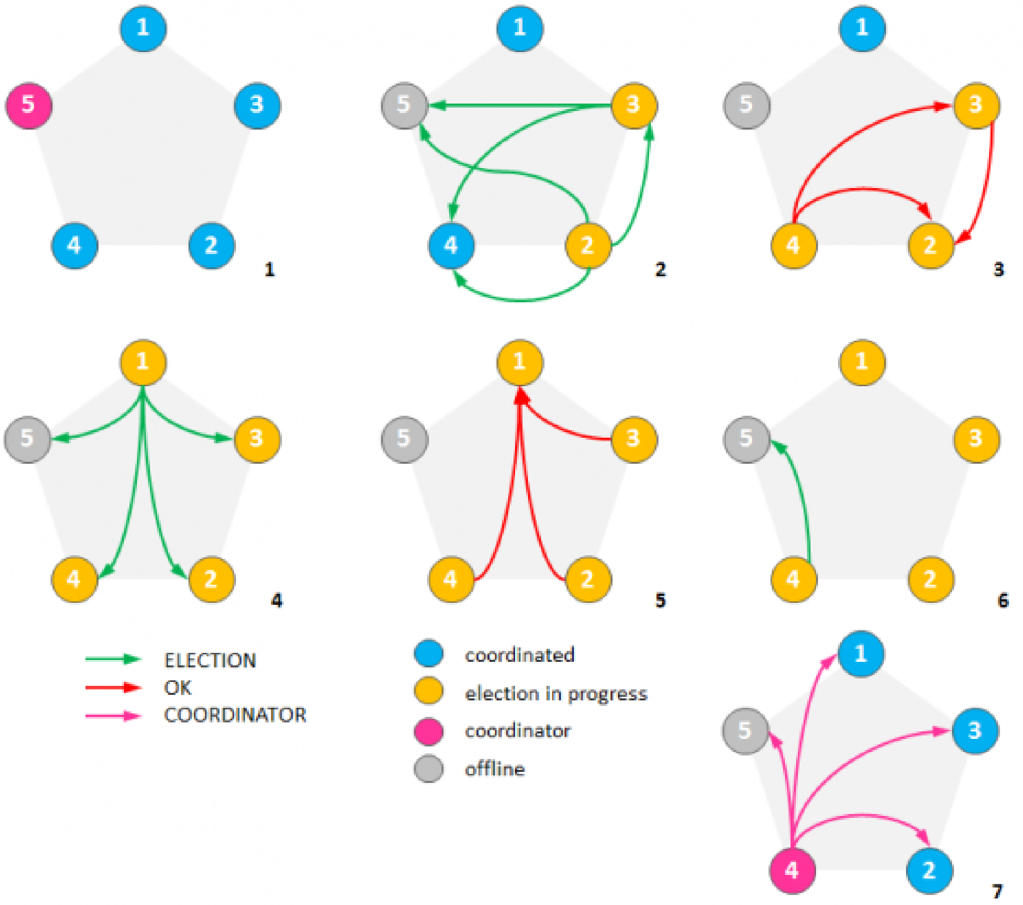
Coordinator Election

Coordinator election is an important technique for databases with strict consistency guarantees. First, it allows one to organize failover of a master node in master-slave systems. Second, it allows one to prevent write-write conflicts in case of network partition by terminating partitions that do not include a majority of nodes.

Bully algorithm is a relatively simple approach to coordinator election. MongoDB uses a version of this algorithm to elect leaders in replica sets. The main idea of the bully algorithm is that each member of the cluster can declare itself as a coordinator and announce this claim to other nodes. Other nodes can either accept this claim or reject it by entering the competition for being a coordinator. Node that does not face any further contention becomes a coordinator. Nodes use some attribute to decide who wins and who loses. This attribute can be a static ID or some recency metric like the last transaction ID (the most up-to-date node wins).

An example of the bully algorithm execution is shown in the figure below. Static ID is used as a comparison metric, a node with a greater ID wins.

1. Initially five nodes are in the cluster and node 5 is a globally accepted coordinator.
2. Let us assume that node 5 goes down and nodes 3 and 2 detect this simultaneously. Both nodes start election procedure and send election messages to the nodes with greater IDs.
3. Node 4 kicks out nodes 2 and 3 from the competition by sending OK. Node 3 kicks out node 2.
4. Imagine that node 1 detects failure of 5 now and an election message to the all nodes with greater IDs.
5. Nodes 2, 3, and 4 kick out node 1.
6. Node 4 sends an election message to node 5.
7. Node 5 does not respond, so node 4 declares itself as a coordinator and announce this fact to all other peers.



(http://highlyscalable.files.wordpress.com/2012/09/bully-algorithm.png)

Coordinator election process can count a number of nodes that participate in it and check that at least a half of cluster nodes are attend. This guarantees that only one partition can elect a coordinator in case of network partition.

References

1. M. Shapiro et al. A Comprehensive Study of Convergent and Commutative Replicated Data Types (http://hal.inria.fr/docs/00/55/55/88/PDF/techreport.pdf)
2. I. Stoica et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications (http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf)
3. R. J. Honicky, E.L.Miller. Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution (http://www.ssrc.ucsc.edu/Papers/honicky-ipdps04.pdf)
4. G. Shah. Distributed Data Structures for Peer-to-Peer Systems (http://cs-www.cs.yale.edu/homes/shah/pubs/thesis.pdf)
5. A. Montresor, Gossip Protocols for Large-Scale Distributed Systems (http://sbrc2010.inf.ufrgs.br/resources/presentations/tutorial/tutorial-montresor.pdf)
6. R. Escriva, B. Wong, E.G. Sirer. HyperDex: A Distributed, Searchable Key-Value Store (http://hyperdex.org/papers/hyperdex.pdf)
7. A. Demers et al. Epidemic Algorithms for Replicated Database Maintenance (http://net.pku.edu.cn/~course/cs501/2009/reading/1987-SPDC-Epidemic%20algorithms%20for%20replicated%20database%20maintenance.pdf)
8. G. DeCandia, et al. Dynamo: Amazon’s Highly Available Key-value Store (http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf)
9. R. van Resesse et al. Efficient Reconciliation and Flow Control for Anti-Entropy Protocols (http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf)

10. S. Ranganathan et al. Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters (http://www.hcs.ufl.edu/pubs/CC2000.pdf)
11. http://www.slideshare.net/kakugawa/distributed-counters-in-cassandra-cassandra-summit-2010 (http://www.slideshare.net/kakugawa/distributed-counters-in-cassandra-cassandra-summit-2010)
12. N. Hayashibara, X. Defago, R. Yared, T. Katayama. The Phi Accrual Failure Detector (http://cassandra-shawn.googlecode.com/files/The%20Phi%20Accrual%20Failure%20Detector.pdf)
13. M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process (http://www.cs.mcgill.ca/~carl/impossible.pdf)
14. N. Hayashibara, A. Cherif, T. Katayama. Failure Detectors for Large-Scale Distributed Systems (http://ddg.jaist.ac.jp/pub/HCK02.pdf)
15. M. Leslie, J. Davies, and T. Huffman. A Comparison Of Replication Strategies for Reliable Decentralised Storage
16. A. Lakshman, P.Malik. Cassandra – A Decentralized Structured Storage System (http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf)
17. N. A. Lynch. Distributed Algorithms
18. G. Tel. Introduction to Distributed Algorithms
19. http://basho.com/blog/technical/2010/04/05/why-vector-clocks-are-hard/ (http://basho.com/blog/technical/2010/04/05/why-vector-clocks-are-hard/)
20. L. Lamport. Paxos Made Simple (http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf)
21. J. Chase. Distributed Systems, Failures, and Consensus (http://www.cs.duke.edu/courses/fall07/cps212/consensus.pdf)
22. W. Vogels. Eventualy Consistent – Revisited (http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)
23. J. C. Corbett et al. Spanner: Google's Globally-Distributed Database (http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/spanner-osdi2012.pdf)

Tagged: *database*, *algorithm*, *nosql*, *consistency*, *protocol*, *sharding*, *cassandra*
Posted in: *NoSQL (http://highlyscalable.wordpress.com/category/nosql-2/)*
9 Responses "Distributed Algorithms in NoSQL Databases" →

1.

Brian

September 19, 2012

Very crisp description of consistency models. Thank you.

Reply

2.

galacticaspirations

September 20, 2012

Amazing post.

Reply

3.

S. Mojtaba Banaie

September 20, 2012

thanks for your clear and complete post.

Reply

4.

Scalabl3 (@scalabl3)

September 21, 2012

this is a great post! but in your tweet you tagged @couchbase but didn't describe anything about couchbase here (no mention)? Do you need more information from our core engineers about how couchbase fits in?

Reply

5. noone

   September 21, 2012

   I continue to be amazed by the quality, depth, and range of your blog posts. I learn a ton from these, and imagine others do as well. Very nice job

   Reply

6. Alex Baranau

   September 23, 2012

   Very good post. Thank you!

   Small comment/addition:

   > "An alternative approach is to prevent distributed concurrent writes entirely and route all writes of a particular data item to a single node (master)"

   Not necessarily master: e.g. in HBase (which uses the mentioned approach) all writes of particular record are routed to a single RegionServer node directly, which is "slave". I.e. master is not a bottleneck.

   nb. you can put "This approach is used e.g. in HBase" in the end of that paragraph

   Reply

   - Ilya Katsov

     September 24, 2012

     Alex,
     Thanks for pointing this out. I'll fix this. What I actually meant is a "shard master", like a primary node in MongoDB's replica set or a region server in HBase.

     Reply

     - Alex Baranau

       September 24, 2012

       Right, that is what I thought you meant. Just wanted to help fixing the wording, as it could mislead someone.

       Thanx again for a great post!

7. Hermann Schmidt

   September 24, 2012

   Awesome, you did it again. Thank you for the deep research. Now I need some time to let it sink in.

   Reply