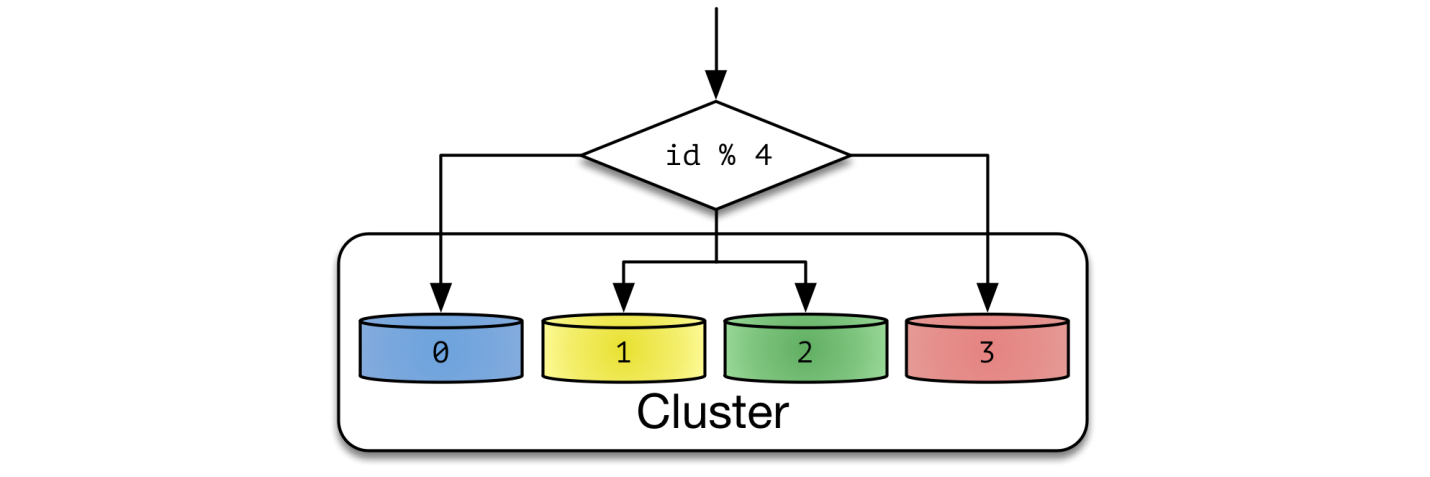
# **Case 1 — Algorithmic Sharding**

One way to categorize sharding is [algorithmic versus dynamic](http://blog.clustrix.com/2013/01/17/sharding-in-theory-and-practice-part-two/). In algorithmic sharding, the client can determine a given partition’s database without any help. In dynamic sharding, a separate locator service tracks the partitions amongst the nodes.



An algorithmically sharded database, with a simple sharding function

Algorithmically sharded databases use a sharding function *(partition\_key) -> database\_id* to locate data. A simple sharding function may be “*hash(key) % NUM\_DB*”.

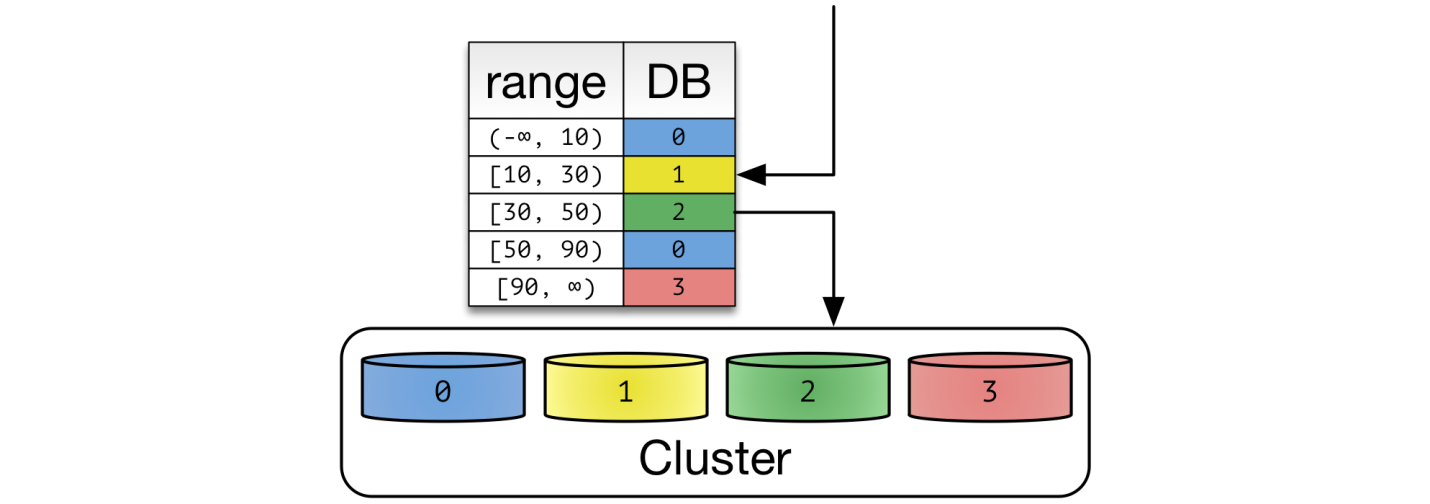
Reads are performed within a single database as long as a partition key is given. Queries without a partition key require searching every database node. Non-partitioned queries do not scale with respect to the size of cluster, thus they are discouraged.

Algorithmic sharding distributes data by its sharding function only. It doesn’t consider the payload size or space utilization. To uniformly distribute data, each partition should be similarly sized. Fine grained partitions reduce hotspots — a single database will contain many partitions, and the sum of data between databases is statistically likely to be similar. For this reason, algorithmic sharding is suitable for key-value databases with homogeneous values.

Resharding data can be challenging. It requires updating the sharding function and moving data around the cluster. Doing both at the same time while maintaining consistency and availability is hard. Clever choice of sharding function can reduce the amount of transferred data. [Consistent Hashing](http://www.paperplanes.de/2011/12/9/the-magic-of-consistent-hashing.html) is such an algorithm.

Examples of such system include Memcached. Memcached is not sharded on its own, but expects client libraries to distribute data within a cluster. Such logic is fairly easy to implement at the application level.

# **Case 2— Dynamic Sharding**

****

A dynamic sharding scheme using range based partitioning.

In dynamic sharding, an external **locator service** determines the location of entries. It can be implemented in multiple ways. If the cardinality of partition keys is relatively low, the locator can be assigned per individual key. Otherwise, a single locator can address a range of partition keys.

1. g.

****List partitioning:**** In this scheme, each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, we can decide all users living in Iceland, Norway, Sweden, Finland or Denmark will be stored in a partition for the Nordic countries.

To read and write data, clients need to consult the locator service first. Operation by primary key becomes fairly trivial. Other queries also become efficient depending on the structure of locators. In the example of range-based partition keys, range queries are efficient because the locator service reduces the number of candidate databases. Queries without a partition key will need to search all databases.

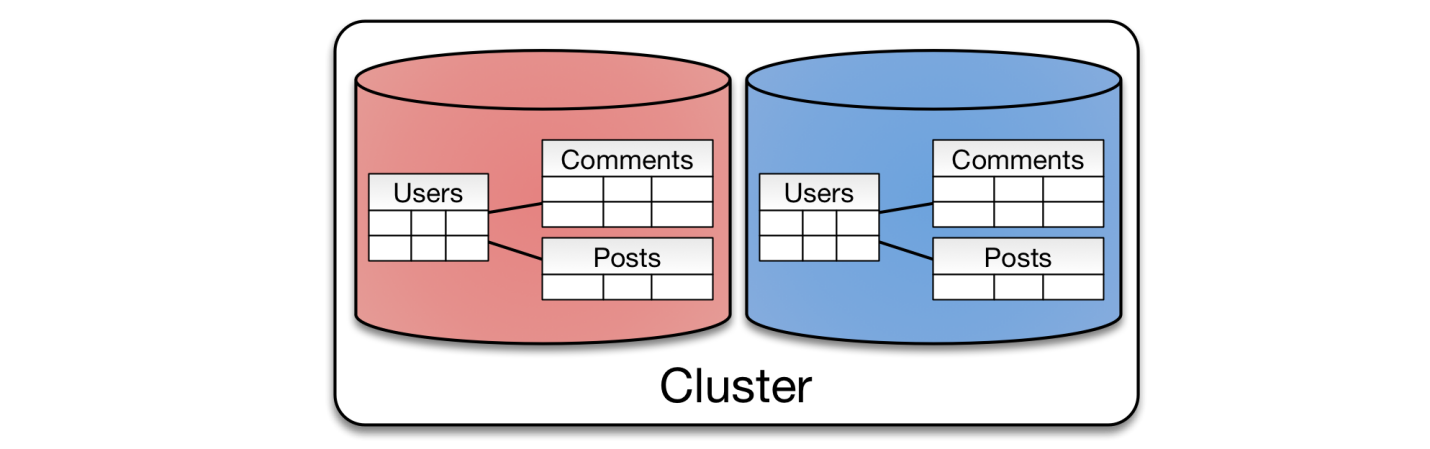
Dynamic sharding is more resilient to nonuniform distribution of data. Locators can be created, split, and reassigned to redistribute data. However, relocation of data and update of locators need to be done in unison. This process has many corner cases with a lot of interesting theoretical, operational, and implementational challenges.

The locator service becomes a single point of contention and failure. Every database operation needs to access it, thus performance and availability are a must. However, locators cannot be cached or replicated simply. Out of date locators will route operations to incorrect databases. Misrouted writes are especially bad — they become undiscoverable after the routing issue is resolved.

Since the effect of misrouted traffic is so devastating, many systems opt for a high consistency solution. Consensus algorithms and synchronous replications are used to store this data. Fortunately, locator data tends to be small, so computational costs associated with such a heavyweight solution tends to be low.

Due to its robustness, dynamic sharding is used in many popular databases. **HDFS** uses a [Name Node](http://blog.cloudera.com/blog/2012/03/high-availability-for-the-hadoop-distributed-file-system-hdfs/) to store filesystem metadata. Unfortunately, the name node is a single point of failure in HDFS. **Apache HBase** splits row keys into ranges. The range server is responsible for storing multiple regions. Region information is stored in Zookeeper to ensure consistency and redundancy. In **MongoDB**, the [ConfigServer](http://docs.mongodb.org/manual/core/sharded-cluster-config-servers/#sharding-config-server) stores the sharding information, and mongos performs the query routing. ConfigServer uses synchronous replication to ensure consistency. When a config server loses redundancy, it goes into read-only mode for safety. Normal database operations are unaffected, but shards cannot be created or moved.

# **Case 3 — Entity Groups**

****

Entity Groups partitions all related tables together

Previous examples are geared towards key-value operations. However, many databases have more expressive querying and manipulation capabilities. Traditional RDBMS features such as joins, indexes and transactions reduce complexity for an application.

The concept of entity groups is very simple. Store related entities in the same partition to provide additional capabilities within a single partition. Specifically:

1. Queries within a single physical shard are efficient.
2. Stronger consistency semantics can be achieved within a shard.

This is a popular approach to shard a relational database. In a typical web application data is naturally isolated per user. Partitioning by user gives scalability of sharding while retaining most of its flexibility. It normally starts off as a simple company-specific solution, where resharding operations are done manually by developers. Mature solutions like [Youtube’s Vitess](https://github.com/youtube/vitess) and [Tumblr’s Jetpants](https://github.com/tumblr/jetpants) can automate most operational tasks.

Queries spanning multiple partitions typically have looser consistency guarantees than a single partition query. They also tend to be inefficient, so such queries should be done sparingly.

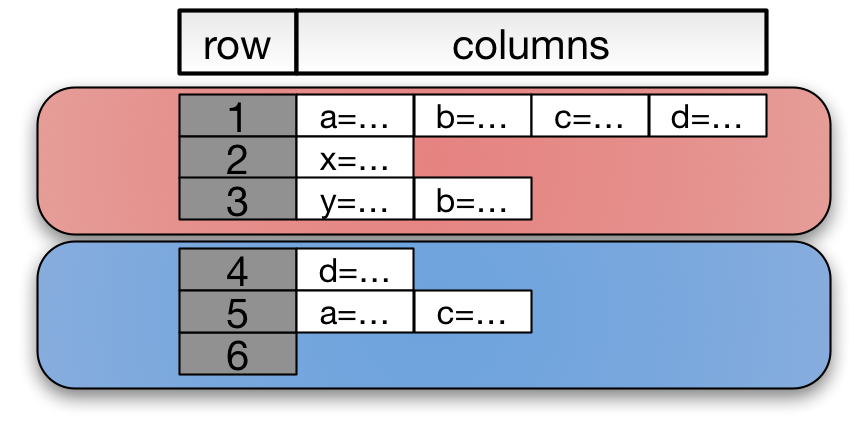
However, a particular cross-partition query may be required frequently and efficiently. In this case, data needs to be stored in multiple partitions to support efficient reads. For example, chat messages between two users may be stored twice — partitioned by both senders and recipients. All messages sent or received by a given user are stored in a single partition. In general, many-to-many relationships between partitions may need to be duplicated.

Entity groups can be implemented either algorithmically or dynamically. They are usually implemented dynamically since the total size per group can vary greatly. The same caveats for updating locators and moving data around applies here. Instead of individual tables, an entire entity group needs to be moved together.

Other than sharded RDBMS solutions, **Google Megastore** is an example of such a system. Megastore is publicly exposed via Google App Engine’s [Datastore API](https://cloud.google.com/appengine/docs/python/datastore/entities).

# **Case 4 — Hierarchical keys &**

# **Column-Oriented Databases**

****

Column-oriented databases partition its data by row keys.

Column-oriented databases are an extension of key-value stores. They add expressiveness of entity groups with a **hierarchical primary key**. A primary key is composed of a pair *(row key, column key)*. Entries with the same partition key are stored together. Range queries on columns limited to a single partition are efficient. That’s why a column key is referred as a *range key* in DynamoDB.

This model has been popular since mid 2000s. The restriction given by hierarchical keys allows databases to implement data-agnostic sharding mechanisms and efficient storage engines. Meanwhile, hierarchical keys are expressive enough to represent sophisticated relationships. Column-oriented databases can model a problem such as [time series](http://planetcassandra.org/getting-started-with-time-series-data-modeling/) efficiently.

Column-oriented databases can be sharded either algorithmically or dynamically. With small and numerous small partitions, they haveconstraints similarto key-value stores. Otherwise, dynamic sharding is more suitable.

The term *column database* is losing popularity. Both HBase and Cassandra once marketed themselves as column databases, but not anymore. If I need to categorize these systems today, I would call them hierarchical key-value stores, since this is the most distinctive characteristic between them.

Originally published in 2005, [Google BigTable](http://en.wikipedia.org/wiki/BigTable) popularized column-oriented databases amongst the public. [Apache HBase](http://hbase.apache.org/) is a BigTable-like database implemented on top of Hadoop ecosystem. [Apache Cassandra](http://cassandra.apache.org/)previously described itself as a column database — entries were stored in column families with row and column keys. CQL3, the latest API for Cassandra, presents a flattened data model — *[(partition key, column key)](http://www.datastax.com/documentation/cql/3.1/cql/ddl/ddl_compound_keys_c.html)* is simply a composite primary key. Amazon’s Dynamo popularized highly available databases. [Amazon DynamoDB](http://aws.amazon.com/dynamodb/) is a platform-as-a-service offering of Dynamo. DynamoDB uses [(hash key, range key)](http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithTables.html) as its primary key.