

Cloud storage

In Chapter 1, we mentioned the Sloan Digital Sky Survey dataset, which is the result of several years of work. It is the most detailed 3D map of the universe ever made.

If you browse through the dataset, you will see that it has 273 TB of data, organized as 176,000,000 files in 680,000 directories. This data contains 260 millions of stars, 208 millions of galaxies, in total 1,231,000,000 objects on 4 spectra.

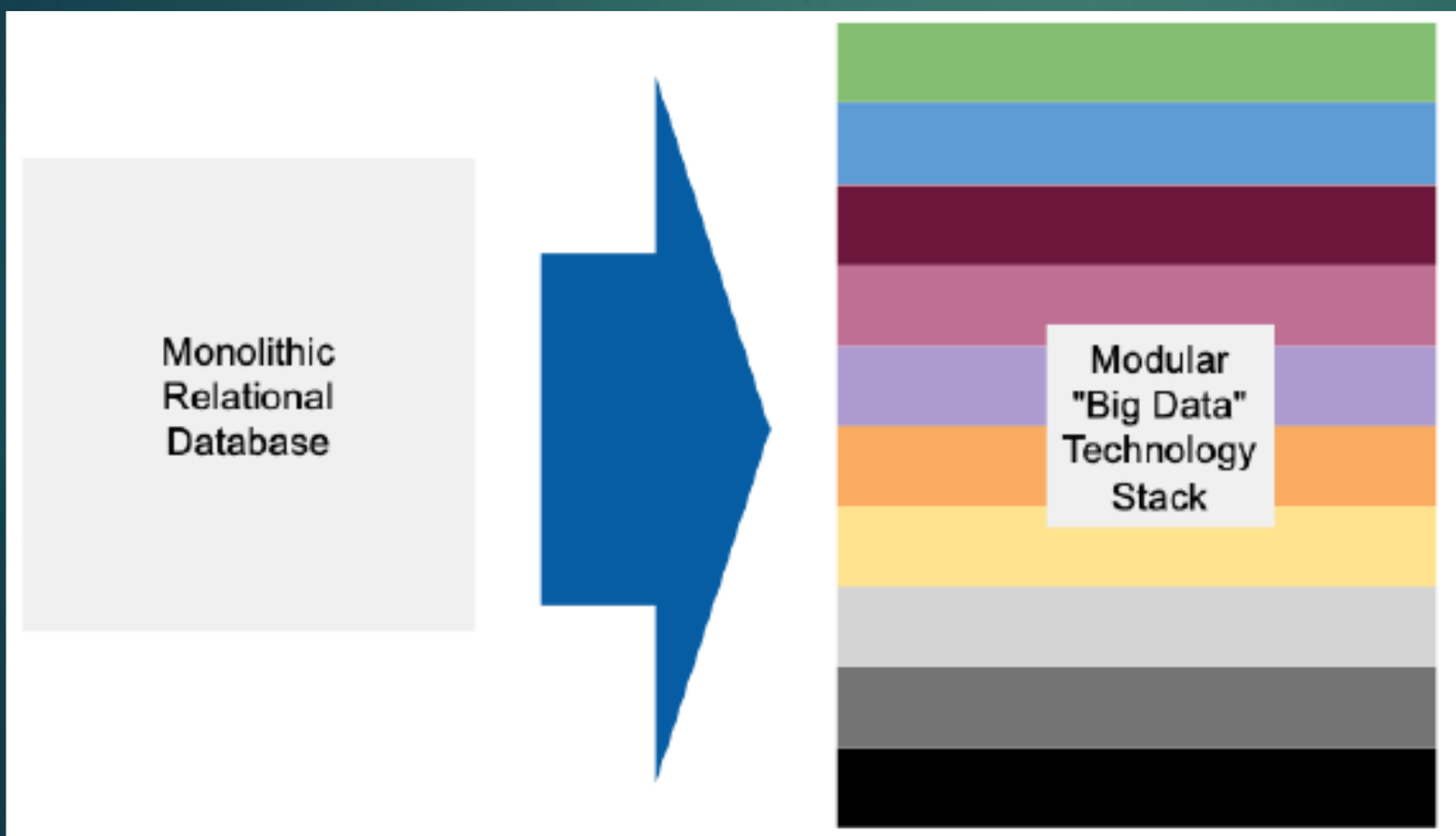
This is much more than any single disk in 2021 can store, which means that a solution with a single machine cannot possibly work.

So, how do we deal with 273 TB of data organized as 176,000,000 files in 680,000 directories?

This is the question we answer in this chapter.

Storing data

As a consequence, in this course, we will have to rebuild the entire technology stack, bottom to top, with those same concepts and insights that we got in the past decades, but on clusters of machines rather than on a single machine.



Indeed, the good news is that 99% of what we learn in the past 50+ years in the field of SQL and relational database management systems can be directly reused.

The relational algebra: selection, projection, grouping, sorting, joining, ... still makes sense on clusters.

Language features: SQL, declarative languages, functional languages, optimizations, query plans, indices... still make sense on clusters.

The table layout: tables, rows, columns, primary keys... still makes sense on clusters.

But not everything. We will have to let go, partially or fully, of consistency constraints. In the Big Data world, relational integrity may or may not hold. Domain integrity may or may not hold. Atomic integrity (also known as the first normal form) may or may not hold. And all normal forms may or may not hold.



Indeed, in what is called the NoSQL universe, these concepts are replaced with:

- Nested data: data that is not in first normal form;
- Heterogeneous data: data that does not fulfil domain integrity (it may not even have a schema!) and also not relational integrity.

Collectively, this alternate design is referred to as *data denormalization*. At large scales, it becomes desirable to sometimes denormalize, rather than normalize data, in order to get better performance.

The same goes for the ACID paradigm: Atomicity, Consistency, Isolation, Durability are very challenging to obtain on large systems (even though there is a lot of research going in this direction). As a consequence, this concept is replaced with thinking in terms of the so-called CAP theorem:

- (Atomic) Consistency
- Availability
- Partition tolerance

The technology stack

The technology stack we are going to rebuild goes from a level very close to the hardware (storage) and all the way to interaction with the end user (query language or even voice assistant).



Examples of storage systems involve the local file system on your laptop, network file systems (NFS), distributed file systems (HDFS from Hadoop, GFS from Google), as well as hosted and fully managed cloud storage (S3, Azure Blob Storage...).

Encodings are used to convert data (for textual syntax) to bits, as this is what computers understand. In the early days, ASCII was used, later on ISO-8859-1 for latin characters but the recommendation today is to stick with Unicode, for example UTF-8, for full compatibility also across (human) languages. There are also binary encodings such as BSON (which is a binary format corresponding roughly to JSON, used in MongoDB).

Syntax is used when data is stored as (human-readable) text. This includes among others raw text (for natural language processing), CSV (for tables), XML and JSON (for trees), RDF/XML and Turtle (for graphs) and XBRL (for trees). Data can also be stored in binary form (protocol buffers, BSON, Parquet...), in which case it cannot be read by a human in a text editor.

Models are a higher-level representation of the data not necessarily tied to a particular syntax or binary format. For tables, it is the relational model. For trees, there are several (the XML infoset, the JSONiq data model...) as well as for graphs (RDF triples and labeled property graphs). For cubes (often “abusively” referred to as OLAP), the most common model uses dimensional coordinates with dimensions organized as hierarchies.

Validation involves languages to add constraints to data: XML Schema, JSON schema, JSound (which are optional for trees), relational schemas (in relational database systems), XBRL taxonomies (for cubes), etc.

Processing frameworks come in different forms and are mostly (at least on the theoretical level) orthogonal to data shapes: the older, two-stage forms (MapReduce) and newer generations that handle generic Directed Acyclic Graphs of dataflows (Tez, Apache Spark, Apache Flink, Ray...). There are also lower level frameworks at the level of the virtual machine, accessed via a command-line shell (Amazon EC2...).

Indices accelerate data lookups. They are mostly orthogonal to data shapes: hash indices, B+-trees, geographical or spacial indices, etc.

Data stores are the generic product providing all the functionality to end users: a relational database management system (RDBMS), document stores (e.g., MongoDB, CouchBase, ElasticSearch), higher-level wrappers on data processing (Hive, HBase, Databricks, Cassandra...) or more general integrated products (Snowflake, MarkLogic...).

Higher-level languages (to be contrasted with data processing APIs like Pandas or Spark's DataFrames) are supported by some but not all data stores. SQL is the undisputed choice for tables. For trees, there are 20+ competing languages among which JSONiq, XQuery, N1QL, PartiQL... for graphs there is mostly Cypher and SPARQL. REST APIs (e.g. with GraphQL) also provide a (lower-level) access to a data store but are meant to be consumed by a host language, whereas query languages can be used on their own.

And finally, user interfaces can hide even the query language from the end user: spreadsheet software, Business Intelligence tools (Tableau, Qlikview, Access...), and even voice assistants (Siri, Alexa...).

But for this chapter, we are focusing on the storage level and in particular, with cloud storage services that require no installation.

Databases vs. Data lakes

There are two main paradigms for storing and retrieving data.

On the one hand, data can be *imported* into the database (this is called ETL, for Extract-Transform-Load. ETL is often used as a verb). This causes more work for the user, as they need to import all their data before they are able to query it. Relational database management systems require an ETL of the data as well as some NoSQL products (MongoDB, HBase...). The reason is that the data is internally stored as a proprietary format that is optimized to make queries faster. This includes in particular building indices on the data.

On the other hand, data can also just be stored on some file system (whether local or distributed or in the cloud) and queried in place (in situ) by a data processing engine either with an API (Pandas, Apache Spark...) or query language (RumbleDB...). This paradigm is called the *data lake* paradigm and gained a lot of popularity in the past two decades. It is slower, however users can start querying their data without the effort of ETLing. Data lakes are best used in a context in which a full scan of the data is needed (e.g. with MapReduce or Apache Spark).

From your laptop to a data center

Local file systems

Typically, on a single machine, the data is stored on the local file system. This is possible if it is no more than a few Terabytes, depending on the disk capacity. There are also newer generations of disks called SSDs (Solid State Disks) that are more expensive and usually a bit smaller, but there exist SSDs of 1 TB nowadays.

When a file is stored on the local filesystem, it consists of both content and metadata. The content is made of the bits of the file itself (e.g., the text encoded to bits in UTF-8 if it is a JSON or XML file). The content is also sliced in blocks that are roughly 4 kB of size (this varies with the file system, but the order of magnitude does not change) and a bit is never read individually: files are read block by block. This is to optimize the balance between throughput and latency (more on this in the next chapter).



The metadata can be seen as a small relational table whose attributes are: the name of the file, access rights, the owner, the group of the owner, the last modification time, the creation time, the size, etc.

Furthermore, files are organized in a hierarchy of files and directories familiar to most people.

More users, more files

How does this scale to more users? A disk can be made accessible via the network (LAN for Local Area Network) and shared with other people. There exists also larger networks (WAN for Wide Area Networks), however it is difficult to scale concurrent access and problems can arise already when two people work on a file at the same time (e.g., for sharing an Excel file, a special mode must be enabled to avoid issues, a feature that text files surely do not have).

The other scalability problem is that a local file system can easily support 1,000 files or even 1,000,000 files, but can hardly make it to billions of files.

In order to make this scale, the approach for large scales, namely here, object storage, is to:

- throw away the hierarchy: there are no directories;
- make the metadata flexible: attributes can differ from file to file (no schema);
- use a very simple, if not trivial, data model: a flat list of files (called objects) identified with an identifier (ID); blocks are not exposed to the user, i.e., an object is a blackbox;
- use a large number of cheap machines rather than some “super-computer”;

Scale up vs. scale out

When a system is too slow on a single machine, there are several ways out.

First, one can buy a bigger machine: more memory, more or faster CPU cores, a larger disk, etc. This is called *scaling up*.

Second, one can buy more, similar machines and share the work across them. This is called *scaling out*.

Scaling up has its limits: while increasing memory from 8 GB to 16 or 32 GB works, 64 GB or 128 GB quickly becomes more expensive, 6 TB is probably beyond what a private individual or small company wants to have, and 1 PB of working memory would require billions if not trillions of dollars of investment to hire a team of dozens of researchers in a Research and Development unit to maybe achieve this in 20 years: in other words the costs of scaling up grow up exponentially, if not hyperbolically (vertical asymptote), making it impractical.

Thus, scaling out is the better approach, and it is also the one taken in Big Data systems: this is why, in a cluster within a data center, there are thousands or even tens of thousands of servers, of all which have performance numbers similar, or slightly better, in order of magnitude, to your personal laptop.

There is also a third way: optimizing your code. In fact, you should always first try to improve your code before scaling out. The vast majority of data processing use cases fit on a single machine, and you can save a lot of money as well as get a faster system by squeezing the data on a machine (possibly compressed) and writing very efficient code.

Scaling out in the context of data processing is typically worth it in just two cases: either because the data does not fit on your laptop, e.b., you are looking at 50 TB of data, or because your laptop cannot read the data fast enough, i.e., your bottleneck is the disk I/O.

Data centers

So in a cluster, in a data center, there are thousands to tens of thousands of machines. Why not more? Well first, it is difficult to know because companies do not always fully communicate on their numbers. However, it is difficult to go beyond this because of pragmatic reasons having to do with power and cooling. A data center consumes as much electricity as an entire airport, and it becomes almost impossible to handle 100,000+ machines together. In fact, the trend at places like CERN is to have less servers, but more cores per server.

A server is also called a node in the context of a data center. Servers typically have between 1 and 64 cores and the number keeps going up.

The working memory available on a node ranges from 16 GB to 6 TB as of today.

The local storage available (SSD or HDD) ranges from 1 to 20 TB.

The local storage available (SSD or HDD) ranges from 1 to 20 TB.

The network bandwidth goes from 1 to 100 Gb/s but higher speeds are also possible especially in the context of High-Performance Computing (HPC). Bandwidth is the highest within the same cluster and can be slower across clusters or data centers, even though there also exist high-speed connections across data centers.

Nodes are typically flat, rectangular boxes that are piled up in what is called a rack, which looks like a tower. A cluster is just a room filled with racks put next to each other.

Each module in a rack can be a server, or pure storage (many disks), or a network switch, etc. The height of a module is standardized and measured in so-called rack units (RU). Each module has typically between 1 and 4 RU.

Data centers are typically concentrated with a few players, called cloud providers. The big three are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud. Most companies on the planet rent resources from these players, typically not directly physically but via the creation and “deletion” of virtual machines and/or of high-level services. Object storage is one of them.

Object stores

Now that we have the hardware in place and have out our disposal thousands of cheap commodity servers, we can come back to the problem at hand: how do we store plenty of data?

Most cloud providers offer a solution for cloud storage, which is typically called object storage. These solutions are typically as-a-service, meaning that users can directly go ahead and use them rather than have to install anything.

Amazon S3

Amazon's object storage system is called Simple Storage Service, abbreviated S3. From a logical perspective, S3 is extremely simple: objects are organized in buckets. Buckets are identified with a bucket ID, and each object within a bucket is identified with an Object ID.

Thus, any object in S3, worldwide, is uniquely identified with a Bucket ID and an Object ID. “Object”, in fact, is only a synonym for “file” and can thus be a text file, a picture, a video, a dataset (CSV, JSON...) etc. But in the context of a object storage, we say “object” rather than “file”. The main reason is that files are organized in hierarchies, but object stores have no hierarchy: just a flat list of objects organized in buckets.

An object can be at most 5 TB (to remember this: it is typically something that fits on a single disk). However, it is only possible to upload an object in a single chunk if it is less than 5 GB – otherwise, the upload must be done in several blocks. We will cover blocks in the next chapter and here focus our attention on objects seen as black boxes.

By default, users can have up to 100 buckets but can ask for more on request.

Azure Blob Storage

Azure Blob Storage is another object storage technology in the Azure cloud.

It is similar in spirit to Amazon S3, in that it is a fully managed cloud storage service. However, it differs from S3 in several ways:

- Its architecture is publicly documented in scientific papers, making it easier to understand how it works.
- Objects are identified with three, rather than two IDs: An Account, a Container (called partition in scientific papers) and a Blob.
- Azure Blob Storage exposes more details to the user. In particular, it exposes that objects are divided in several blocks more prominently than Amazon S3 (in S3, only advanced users would actually know and access blocks within an object, while the frontend hides them).
- it differentiates between Block Blobs, Append Blobs (for logging) and Page Blobs (for storing and accessing the memory of virtual machines).
- the maximum sizes are different and go from 195 GB for an Append Blob to 190.7 TB for a Block Blob.

Block storage will be covered in more details in the next chapter on HDFS.

On the physical level, Azure Blob Storage is organized in so-called storage stamps located in various data centers worldwide. Each storage stamp consists of 10 to 20 racks, with each rack containing around 18 storage nodes (the disks + servers). In all, a storage stamp can store up to ca. 30 PB of data. However, a storage stamp will not be filled more than 80% of its total capacity in order to avoid being full: if a storage stamp reaches capacity, then some data is going to be reallocated to another storage stamp in the background. And if there are not enough storage stamps, well new racks will need to be purchased and installed at the locations that make the most sense.

Guarantees and service level

Service Level Agreements

Most cloud services come with a cost, but also a promise. This promise is a contract between the provider and the user called a Service Level Agreement, abbreviated as SLA.

For example, S3 promises that it loses, each year, less than one object in 100 billion. This is formulated as a durability of 99.999999999%.

Another promise is availability: S3 is available 99.99% of the time, meaning that it will be down less than 1 hour per year.

SLAs very often contain numbers with a lot of 9s. 99% of availability means at most 4 days a year of downtime, 99.9% less than 10 hours, 99.999% six minutes, 99.9999% 32 seconds and 99.99999% less than 4 seconds.

Response times are also expressed in terms of the 99.9% percentile, for example one can promise that 99.9% of the requests will be served in less than 300ms. However, S3 does not make promises in terms of latency: it varies depending from where you request an object, and usually objects are large enough so that the bottleneck is in the time to download the object over the network, not in the latency. Rather, the promise is made in terms of throughput, which in this context is the number of objects read or written per second: typically 5,500 reads/s and 3,500 writes/s.

The CAP theorem

Early relational database management system relied on ACID properties (see Chapter 2). In order to scale out, many distributed systems have to make a compromise on the transactional guarantees that they offer. This is best explained with the so-called CAP theorem.

The CAP theorem is basically an impossibility triangle: a system cannot guarantee at the same time:

- (atomic) Consistency: at any point in time, the same request to any server returns the same result, in other words, all nodes see the same data;
- Availability: the system is available for requests at all times (SLA with very high availability);
- Partition tolerance: the system continues to function even if the network linking its machines is occasionally partitioned.

The CAP theorem is, in fact, not formally proven and it would be more appropriate to call it a conjecture. An intuitive way of explaining it is as follows.

When an update is made via some server, this update must propagate to other nodes to the extent that a specific replication factor for the data is met.

If there is a partition of the network leading to two disconnected sub-networks and the propagation of some data did not have a chance to complete, then there are two possible design decisions:

1. either the system is temporarily put to a halt (synchronous propagation) and thus becomes unavailable until the network partition is resolved and the propagation can be completed. In this case the system is atomically consistent but not available (CP).

2. or the servers continue to serve requests, but the data served by the two disconnected sub-networks will be different. In this case the system is available but not atomically consistent (AP). Such systems propagate updates asynchronously and are also often called *eventually consistent* in the sense that, if one hypothetically would no longer receive updates, there exists a time at which the system will be consistent. In practice of course, as updates keep arriving, the system rarely becomes fully consistent – a common misconception about eventual consistency.

A third possibility is for the system not to be partition tolerant: in this case, the system is available and consistent (AC) but only for as long as no network partition occurs. If a network partition occurs, this is unknown territory and no further guarantees exist.

Beware of systems that claim to be available, atomically consistent and partition-tolerant at the same time: this is often just marketing. In practice, the companies who claim such properties (ab)use the fact that a partition of their network is very rare. But when a partition happens, they have in fact to make a choice between A or C, even though users will barely notice.

REST APIs

Data stores, be it object stores, key-value stores, document stores, wide column stores, distributed file systems, etc expose their functionality via an API. A vast majority of them offers a so-called REST API.

REST means REpresentational State Transfer. In fact, it is really just “HTTP done right” in the sense that it closely uses HTTP functionality: methods and resources.

The benefit of offering a REST API is tremendous: it makes an integration with any host language very easy, because HTTP clients exist in almost any host language (Java, Python, PHP, R...) and it is straightforward to implement a “wrapper API” in the host language that forwards all requests through the REST API to the data store. If you design a new system in the future, it is almost a no-brainer decision to support a REST API.

Other possibilities are a native driver, but this needs to be done for each host language. MongoDB is an example of document store that works with drivers.

Another protocol is the XML-based SOAP protocol.

The HTTP protocol, which REST builds on, was invented back in 1989, thirty years ago, by Sir Tim Berners-Lee, who is also the creator of the Web.

A client and server communicated with the HTTP protocol interact in terms of *methods* applied to *resources*.

A resource can be anything: a document, a PDF, a person, a calendar entry, or even a physical object such as a smart plug (“Web of things”). A resource is referred to with what is called a URI. URI stands for Uniform Resource Identifier¹. A URI looks like so:

`http://www.example.com/api/collection/foobar?id=foobar#head`
where

- “http” is the scheme;
- “//www.example.com” is the authority.;
- “/api/collection/foobar” is the path;
- “?id=foobar” is the query;
- “#head” is the fragment.

There exist other schemes than http, for example mailto, ftp, hdfs, file, and so on. However, http is the most popular schemes even for resources that are “offline”, i.e., not actually reachable via HTTP.

A client can act on resources by invoking methods, with an optional body. The most important methods are:

- GET (without a body): this method returns a representation of the resource in some format (text, XML, JSON...). GET should have no side effects (beyond logging, of course).
- PUT: this method creates or updates a resource from a representation of a newer version of it, in some format (text, XML, JSON...). PUT has the side effect that a subsequent GET asking for the same format should return the same representation. PUT is idempotent, in that calling PUT with the same resource and body is identical to calling it just once.

- DELETE (without a body): this method deletes a resource. DELETE has the side effect that a subsequent GET asking for a representation of the resource should fail with a not-found (404) error.
- POST: this method is a blank-check, in that it acts on a resource in any way the data store seems fit; the behavior, of course, should be publicly documented. A typical use of POST is to create new resources but letting the REST server pick a resource URI for this new resource.

An example of URI for S3 is “http://bucket.s3.amazonaws.com” for a bucket and “http://bucket.s3.amazonaws.com/object-name” for an object.

Object stores in practice

Although object stores are based on a flat key-value model, most object storage services emulate a file hierarchy by allowing slash (/) characters in keys, and interpreting these slashes as virtual paths. As far as the storage layer is concerned, slashes are a character like any other. But on the logical level, a logical hierarchy enables more use cases for object stores.

Static website hosting

Object stores such as S3 or Azure Blob Storage can be used for static website hosting in a very straightforward way.

Static website hosting means that there is no dynamically generated content, as would be the case for example with a PHP or Java EE server (Tomcat, etc). In a static website, the HTML pages, pictures, CSS stylesheets, (client-side) JavaScript scripts, videos are delivered *as is*.

It is both very convenient and relatively cheap to drop HTML pages, pictures, CSS stylesheets and JavaScript scripts in an object storage service, and use this as a simple website backend. Most cloud providers provide a very simple way of accessing these objects via HTTP URLs.

It is common to place a content delivery network (CDN) service on top of the storage bucket of a website in order to accelerate and cache these files at multiple places on the planet. This is particularly useful if very high traffic is expected over short periods of time, for example for delivering a file that many people will rush to download at the same time. For the user, content delivered with a CDN feels instantaneous.

Dataset storage

Object stores are popular as data lakes, in the sense that datasets can be stored as objects in any desired format (CSV, JSON, ...).

As we will see later, a dataset often consists of several objects rather than just one. Hence, even though the size of an object is typically limited (e.g., 5 TB on S3), the size of a dataset is not. This way of splitting a dataset into multiple chunks is called sharding and will play a fundamental role throughout this course, as we will see when we study MapReduce and Spark.

Key-value stores

To what extent can we consider object stores such as Amazon S3 or Azure Blob Storage as a database system? While they are able to store large objects well into the TB range, which typically does not fit in a relational database system, their latency (100 milliseconds or more) is higher than typical database systems (between 1 and 9 milliseconds).

In fact, there exists a paradigm that has a similar key-value model (based on a get/put/delete API) with a low latency. The price to pay is that objects will be smaller (the order of magnitude being up to a few hundreds of kilobytes, 400 kB for Dynamo). Also, no metadata can be stored along objects, as is otherwise possible with S3 or Azure Blob Storage.

Key-value stores have known a moment of hype and glory for a while (the buzz word is “eventual consistency”), to the point that they were used beyond their design goals. One of the typical use cases for a key-value store is storing shopping carts for a very large online shop, another is for storing likes and comments on social media.

A key-value store differs from a typical relational database in three aspects:

- Its API is considerably simpler than that of a relational database (which comes with query languages)
- It does not ensure atomic consistency; instead, it guarantees eventual consistency, which we covered earlier in this Chapter.
- A key-value store scales out well, in that it is very fast also at large scales.

Key-value store API

A key-value store, on the high level, is based on the key-value model; it can be seen as some sort of a gigantic associative array (also logically known as “map” or “dict”) that is constantly modified and accessed.

A key-value store, on the high level typically has three access methods, which should not come as a surprise:

- `get(key)` retrieves the value associated with a key.

- `put(key, value)` associates a new value with the key.

- `delete(key)` removes any association of a value with the key.

In practice, the API is slightly more complex because of conflict resolution. In the case of Dynamo, for example, `get(key)` not only returns value(s), but also a context indicating potential conflicts, which for now can be seen as a black box that we will get back to. Likewise, `put` has a third parameter, `context` (which is typically forwarded “as is” from a previous `get` call), and `put(key, context, value)` associates the key with a new value while resolving any conflict.

Design principles

We will focus on a specific key-value technology for our explanations: Amazon Dynamo. It is itself based (with some modifications) on the Chord protocol, which is a Distributed Hash Table.

Distributed Hash Tables are generally highly scalable, robust against failure and self organizing. However, they do not provide any support for range queries, guarantees on data integrity (which is pushed to the user), and do not deal with any security issues.

On the physical level, a distributed hash table is made of nodes (the machines we have in a data center, piled up in racks) that work following a few design principles. The first design principle is *incremental stability*. This means that new nodes can join the system at any time, and nodes can leave the system at any time, sometimes gracefully, sometimes in a sudden crash.

The second principle is symmetry: no node is particular in any way

The third principle is decentralization: there is no “central node” that orchestrates the others. One may wonder what the difference with symmetry is? A symmetric system could still have protocols that elect a leader that then takes up an orchestration role. But this is not the case in a distributed hash table.

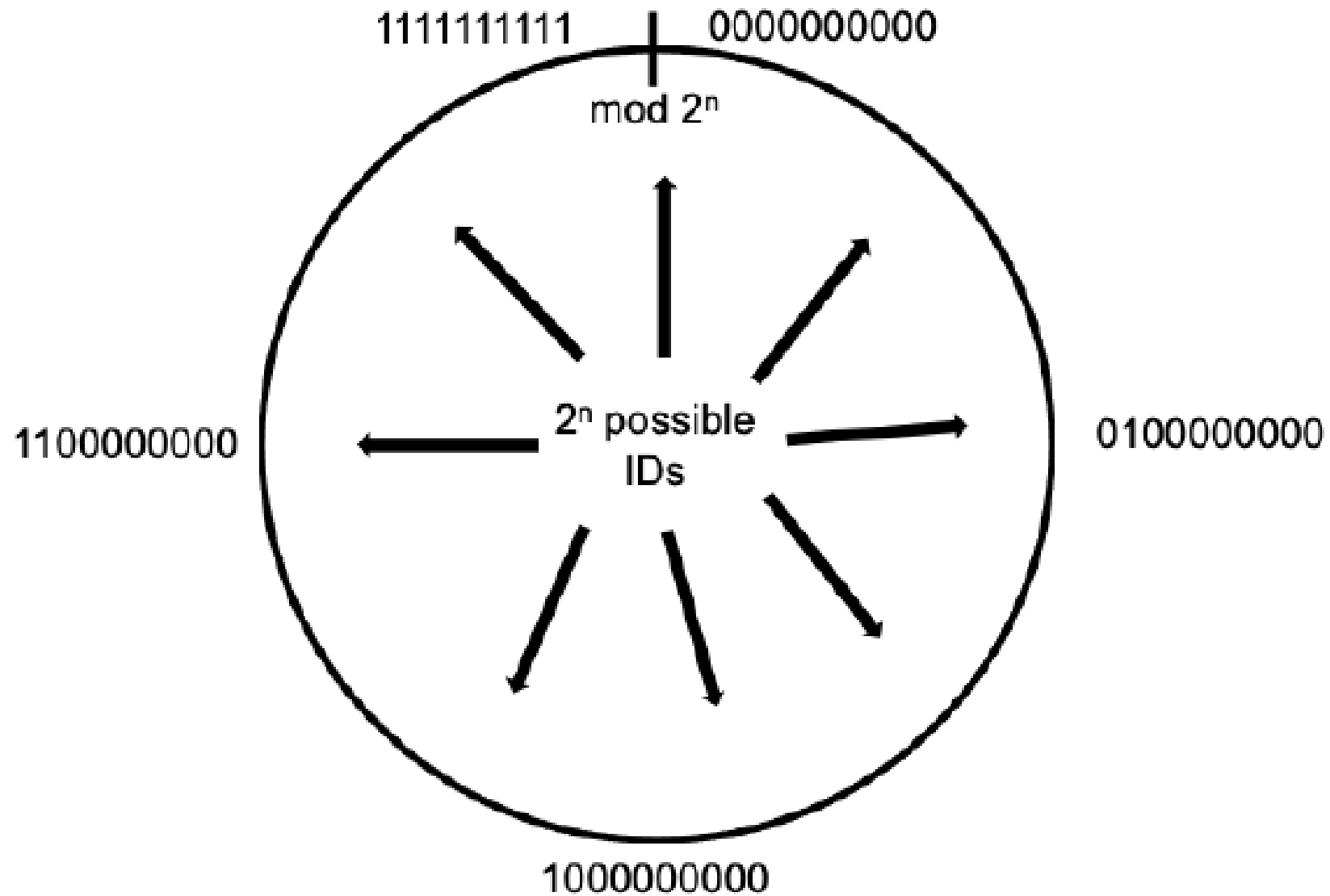
The fourth principle is heterogeneity: the nodes may have different CPU power, amounts of memory, etc.

A ring to rule them all

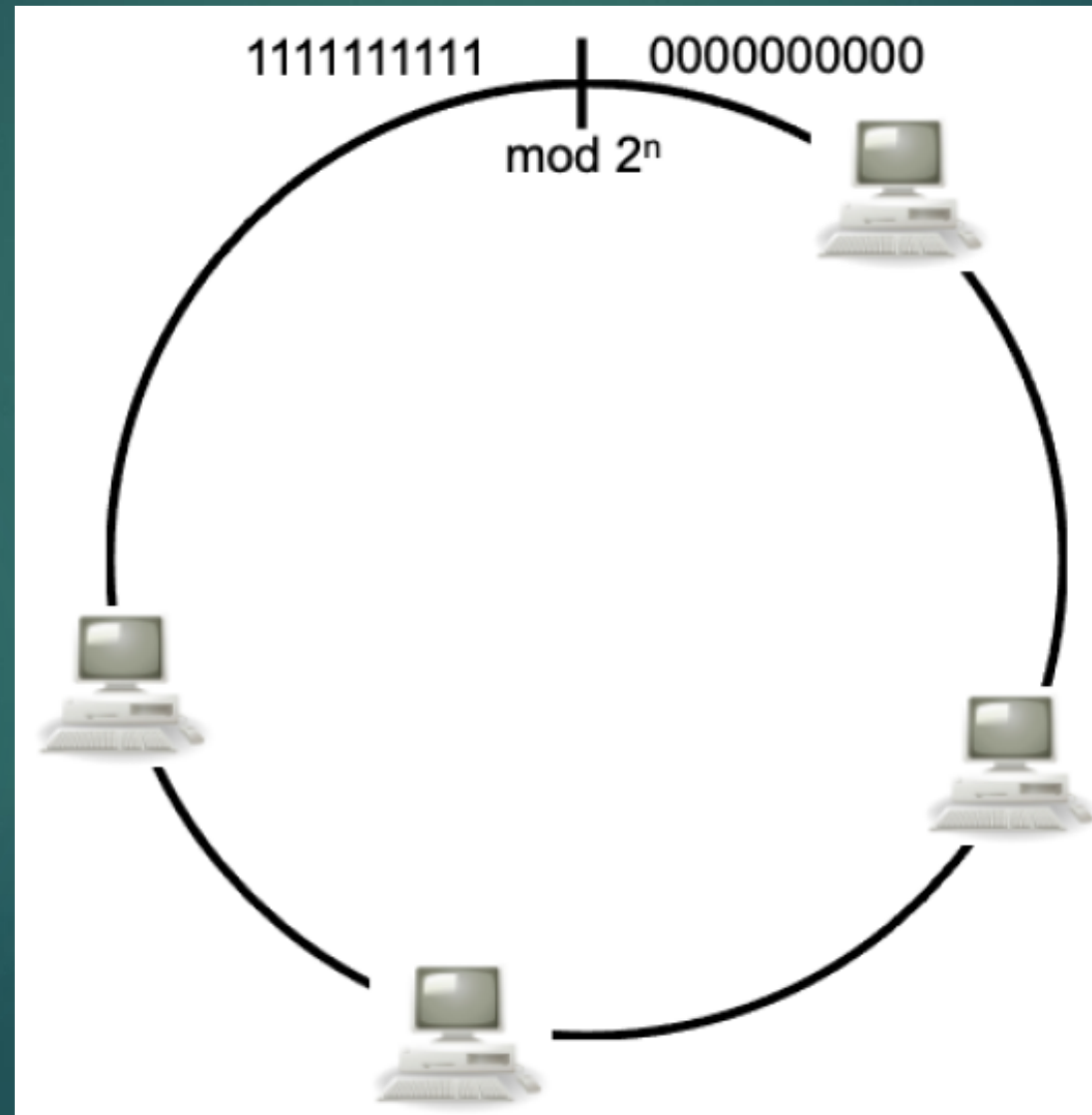
There are several ways that nodes can be connected together. Two main paradigms are peer-to-peer (every node speaks to every other node) and another one is centralized (there is a Coordinator Node that has a special role, and controls the Worker Nodes). In the case of distributed hash tables though, we have a peer to peer network, which is a natural thing to do since we require symmetry and decentralization.

A central aspect of the design of a distributed hash table, and part in particular of the Chord protocol, is that every logical key is hashed to bits that we will call IDs. In the case of Dynamo, the hash is made of 128 bits (7 bytes).

Mathematically, all possible IDs form, quite naturally, a ring structure (consider $\mathbb{Z}/2^{128}\mathbb{Z}$), although in this context we will not need any ring properties in a mathematical sense, but we only need to think of this ring as a circle.

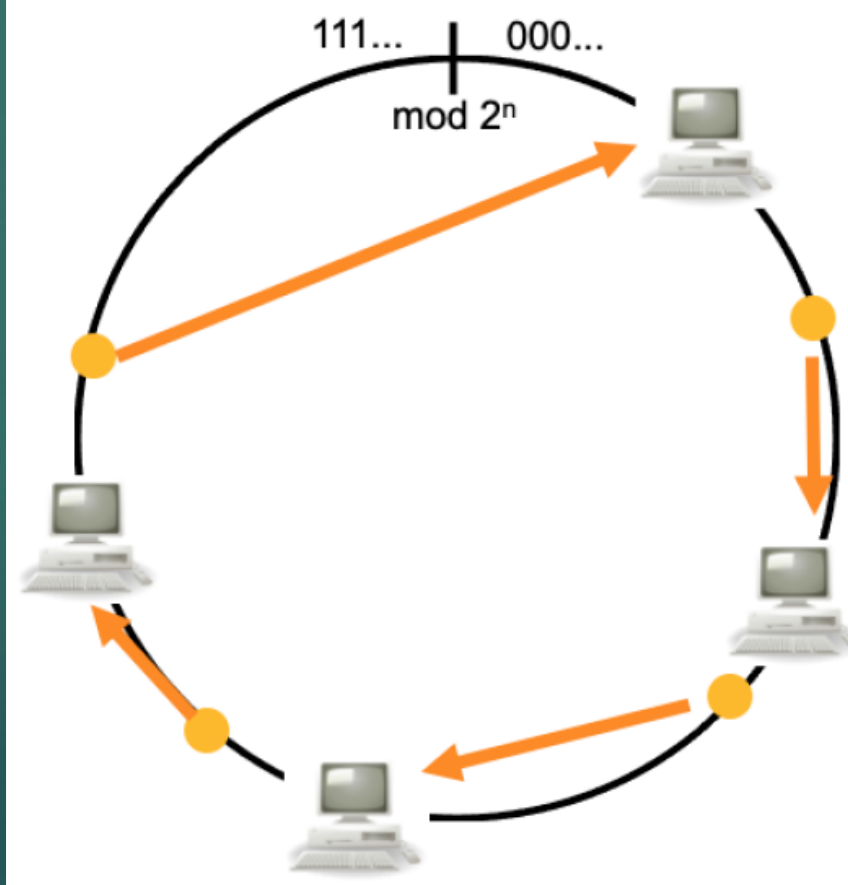


The next conceptual step of the design is that all nodes (remember, this is what we call the machines in the data center) get a random position on that ring, i.e., a random 128-bit number.

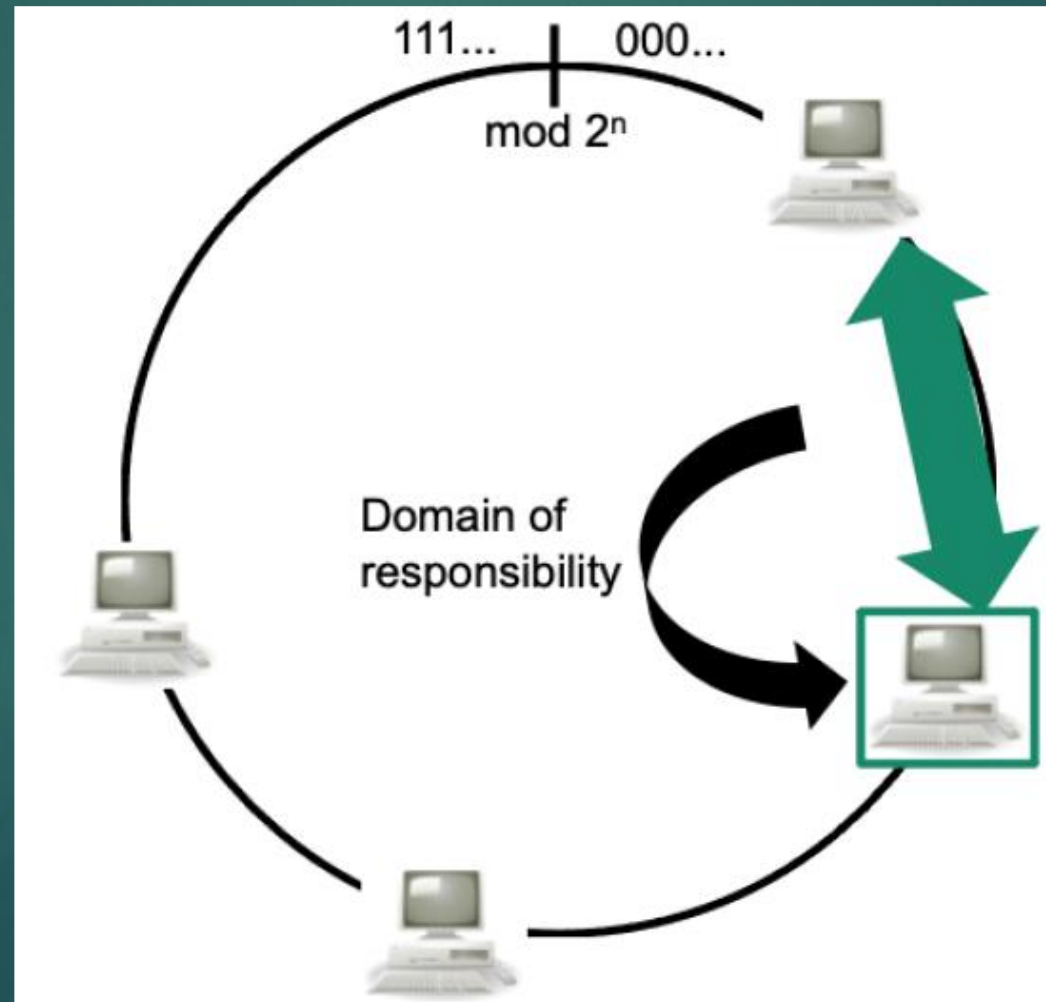


Which node stores what key is then decided based on the ring topology. Given a key k , its 128-bit hash is taken and it corresponds to a position on the ring. From that position on, one follows the ring clockwise, until one reaches one of the nodes (remember, they were logically assigned a position on the ring); this node is responsible for storing key k .

Here are a few examples:



The domain of responsibility of a single node n is thus the ring interval separating it from the next node counterclockwise: all keys whose hash lands in that interval are stored on, and the responsibility of, node n .



Now, what happens when nodes come and go? Well, when a new node n joins the cluster, it gets assigned a position on the ring. This position is in the domain of responsibility of some existing node m . This ring interval is now going to be split: everything between m and n remains within the responsibility of m , but the other half of the domain of responsibility is newly the responsibility of n . The corresponding data needs to be transferred from m to n .

What about when a node leaves? Let us start with the case of a graceful exit, i.e., the node (call it n) informs the rest of the cluster before leaving. Then, it will transfer all its interval of responsibility to the next node clockwise (call it m) on the ring: two intervals merge into a single one. Here too, data needs to be transferred from n to m .

Now, let us look into an abrupt exit: the node crashes with no opportunity to inform the cluster or prepare its exit.

And we have a problem. Because the data for that node is gone.

And this problem is going to be solved with a fundamental principle

that will follow us through the course: data replication. We slightly change the design: data within an interval will not be stored only the node that follow the interval clockwise, but on the next N nodes that follow the interval clockwise. Equivalently explained, a node is responsible not only for the interval that follows it counterclockwise, but for the next N intervals that follow it counterclockwise.

Preference lists

The next step of the design is: given a key (and its 128-bit hash), how do we *physically* find out which node(s) hold the associated value? The explanations we gave so far are logical and based on the ring; but this needs to also work in practice, with the software installed on all nodes. And the challenge is that the system is decentralized, so there is nothing like a central place to look at. The client who wants to access a key needs to know which one of the (say) 1,000 machines it needs to connect to with the hope of getting an answer.

In the chord protocol, a technology called a *finger table* is used. Each node knows the next node clockwise, and the second node, and the 4th node, and the 8th node, etc: going through the powers of two. As you probably guessed, this means that a binary search is used to find the desired node within logarithmic time, hopping from node to node with increasing precision and until one finds The One(s).

Dynamo changes this design to so-called “preference lists”: each node knows, for every key (or key range), which node(s) are responsible (and hold a copy) of it. This is done by associating every key (key range) with a list of nodes, by decreasing priority (going down the ring clockwise). The details of this protocol are left out, but in many systems, sharing the same view between nodes is done either through some peer-to-peer synchronization protocol with all nodes keeping a local copy, or through access to a shared network drive or database (much smaller than the key-value store itself).

The size of every list must be at least N , where N is the parameter discussed earlier (the minimum number of nodes holding a copy of the value for each key). Note that it can get (temporarily) larger than N if there are network partitions or nodes leaving and taking over. The first node in the list (for each key range) is called the coordinator.

Two more parameters come into play: R and W , with

$$R + W > N$$

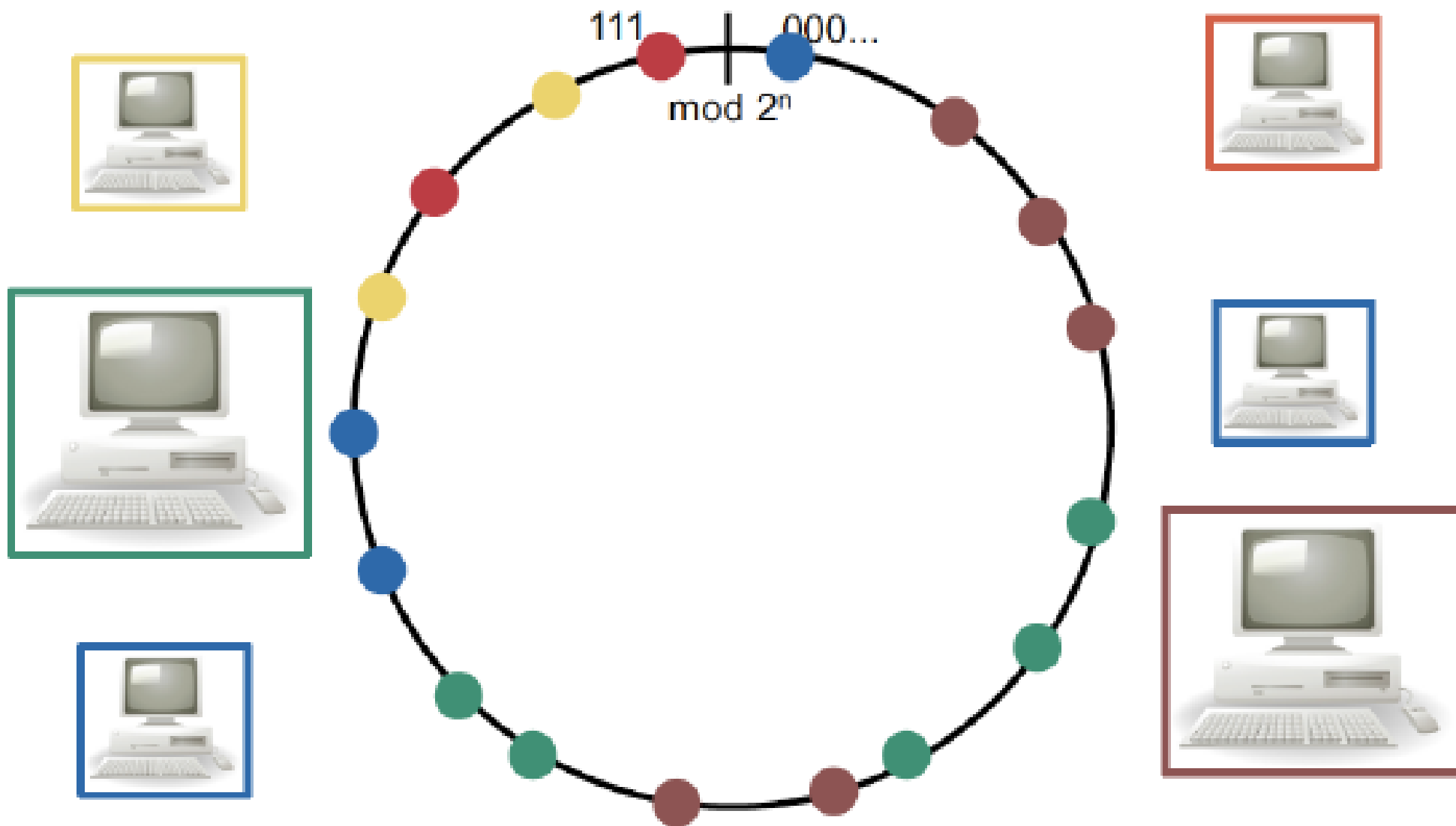
Upon reading a value, R is the minimum number of nodes (among the N nodes responsible for the key) from which a copy of the value must be obtained and compared. Upon writing a new value, W is the minimum number of nodes (among the N nodes responsible for the key) that must have acknowledged receipt and confirmed storage of the value before the put or delete transaction succeeds.

An initial request (put, delete or get) is typically dispatched, thanks to a load balancer, to a random node. This random node then looks at the key in its copy of the preference lists, and redirects the request to the coordinator node (first in the preference list). If this fails, an attempt is made with the next node, etc. Then, the coordinator contacts all other $(N-1)$ nodes in the list, waiting for R (or W) distinct answers from the other nodes, and then the answer to the request is sent back to the client.

An improvement: tokens and virtual nodes

The design described so far has two issues: first, we might be out of luck with the (logical) distribution of the nodes on the ring, with some nodes responsible for large intervals and some others for smaller ones. Second, the nodes may not all have the same hardware resources: some may have more memory or less CPU, etc.

These two issues are addressed by virtualizing the nodes: instead of directly mapping each node to one position on the ring, a larger quantity of (virtual) nodes, called tokens, are positioned on the ring.



For example, there might be 100 nodes in the cluster, and we will position 1000 tokens on the ring. Then, the tokens are assigned to the nodes, in numbers that match their resources: a large node may get, say, 20 tokens, while a small one may get 3. Then the system works as above, each physical node handling the responsibilities assigned with all of its assigned tokens.

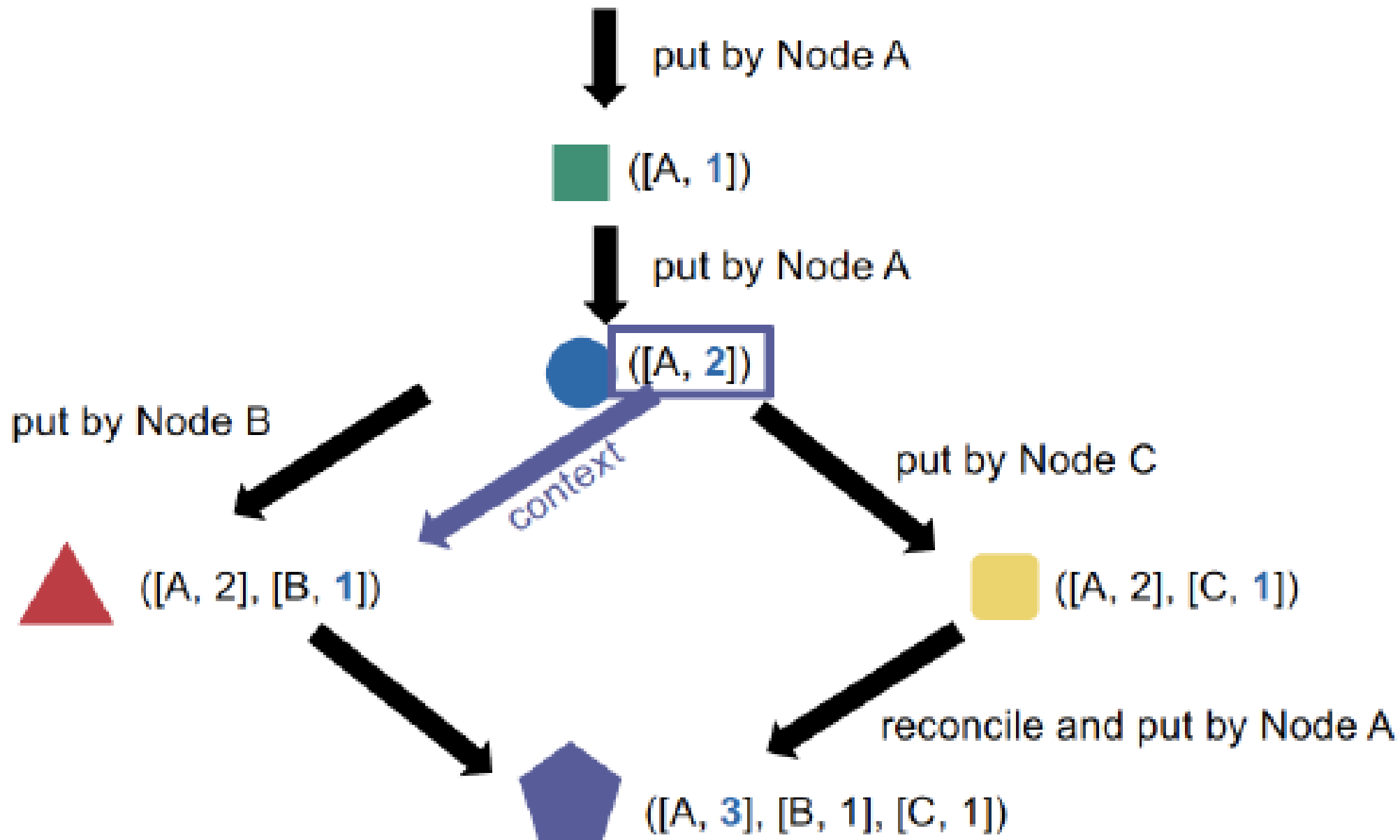
Adding or removing a node is straightforward: all tokens corresponding to a node leaving the network are redistributed to other nodes (not necessarily the same one), and a node joining the network gets assigned tokens taken over from other (not necessarily the same one) nodes.

Partition tolerance and vector clocks

We now analyze how a distributed hash table behaves in the presence of network partitions. As we saw previously, when there is a network partition a system will either crash (CA), become temporarily unavailable (CP), or accept inconsistencies, downgrading the guarantee to eventual consistency (AP). Distributed hash tables, including Dynamo, are typically AP.

A fundamental conceptual tool in AP systems is the use of vector clocks. In a system with no partitions, versions of the data can be seen as linear, and could be identified with integers (in fact, some code repository systems do exactly this).

But in case there are network partitions, the versions may fork, forming a DAG (Directed Acyclic Graph) structure rather than a line. Vector clocks are a way to annotate the versions when they follow a DAG structure.



A vector clock can logically be seen as a map from nodes (machines) to integers, i.e., the version number is incremented per machine rather than globally. A version number increases for a machine when this machine processes the corresponding entry (key-value) and updates it. For machines that have not yet touched a given entry, the corresponding number (implicitly 0) can be left out from the vector clock: it gets added the first time the machine updates the entry.

On the picture above, node A creates the entry and associates it with a vector clock

```
{ "A" : 1 }
```

Then node A (which is probably at the top of the preference list) updates the entry again and it increases:

```
{ "A" : 2 }
```

Then for some reason, the entry was processed, separately and without knowing about each other (network partition!) by nodes B and C. Node B updates the value and associates it with vector clock

```
{ "A" : 2, "B" : 1 }
```

while node C, who does not know about node B, does the same and associates it with vector clock

```
{ "A" : 2, "C" : 1 }
```

We now have two conflicting versions, which will be returned to the user upon getting the value for that key (assuming it sees both node B and C, i.e., the network partition is over), together with the context, which "merges" the conflicting vector clocks with the max integers:

```
{ "A" : 2, "B" : 1, "C" : 1 }
```

This context is passed (via put()) to node A for another update to resolve the conflict, leading to the new vector clock:

```
{ "A" : 3, "B" : 1, "C" : 1 }
```

Note that vector clocks can be compared to each other with a partial order relation $\leq$. A partial order is a relation that is antisymmetric (if the order holds in both directions, then the vector clocks must be equal), reflexive (the order always holds if both sides are the same vector clock), and transitive (if the order holds between vector clocks A and B as well as B and C, then it must hold between vector clocks A and C), but it might not be total in the sense that vector clocks might not be comparable (the order is undefined).

More specifically, a vector clock is smaller or equal to another if for each machine, the two associated integers are also smaller or equal (in the same direction). For example.

$$\{ \text{"A"} : 1 \} \leq \{ \text{"A"} : 1 \}$$

$$\{ \text{"A"} : 1 \} \leq \{ \text{"A"} : 2 \}$$

$$\{ \text{"A"} : 1 \} \leq \{ \text{"A"} : 2, \text{"B"} : 1 \}$$

$$\{ \text{"A"} : 1 \} \leq \{ \text{"A"} : 2, \text{"C"} : 1 \}$$

$$\{ \text{"A"} : 1 \} \leq \{ \text{"A"} : 3, \text{"B"} : 1, \text{"C"} : 1 \}$$

$$\{ \text{"A"} : 2 \} \leq \{ \text{"A"} : 2 \}$$

$$\{ \text{"A"} : 2 \} \leq \{ \text{"A"} : 2, \text{"B"} : 1 \}$$

$$\{ \text{"A"} : 2 \} \leq \{ \text{"A"} : 2, \text{"C"} : 1 \}$$

$$\{ \text{"A"} : 2 \} \leq \{ \text{"A"} : 3, \text{"B"} : 1, \text{"C"} : 1 \}$$

$$\{ \text{"A"} : 2, \text{"B"} : 1 \} \leq \{ \text{"A"} : 3, \text{"B"} : 1, \text{"C"} : 1 \}$$

$$\{ \text{"A"} : 2, \text{"C"} : 1 \} \leq \{ \text{"A"} : 3, \text{"B"} : 1, \text{"C"} : 1 \}$$

A partial order, unlike a total order, may not have a *maximum* (greater than all others), but it may have several *maximal elements* instead (not smaller than any other). When there is a unique maximal element (which is then the maximum because there is a finite number of vector clocks), there is no conflict and only one value is associated with the key throughout the DHT; and conflicts can be recognized because there are several maximal elements (typically one for each former partition), at least until the conflict is resolved.