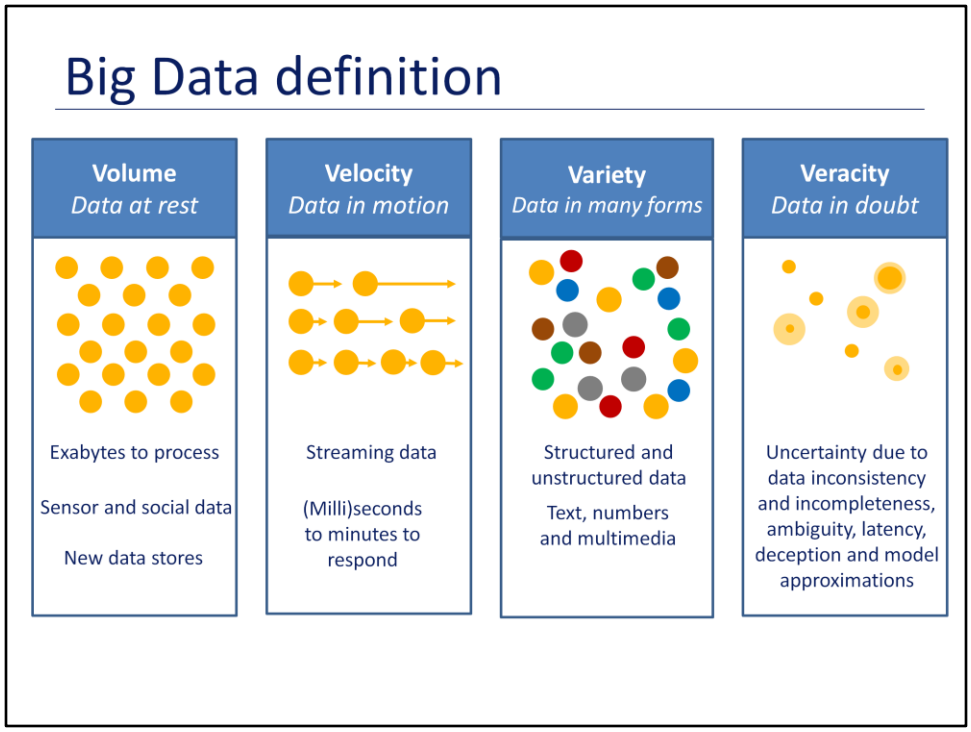
We are living in the era of big data, where exponential growth of phenomena such as web, social networking, smartphones, and so on are producing petabytes of data on a daily basis. Gaining insights from analyzing these very large amounts of data has become a *must-have* competitive advantage for many industries. However, the size and the possibly unstructured nature of these data sources make it impossible to use traditional solutions such as relational databases to store and analyze these datasets.

Storage, processing, and analyzing petabytes of data in a meaningful and timely manner require many compute nodes with thousands of disks and thousands of processors together with the ability to efficiently communicate massive amounts of data among them. Such a scale makes failures such as disk failures, compute node failures, network failures, and so on a common occurrence making fault tolerance a very important aspect of such systems. Other common challenges that arise include the significant cost of resources, handling communication latencies, handling heterogeneous compute resources, synchronization across nodes, and load balancing. As you can infer, developing and maintaining distributed parallel applications to process massive amounts of data while handling all these issues is not an easy task.


1 NEW DEFINITION IS ADDED ON URBAN
1,600+ READS ON Scribd
13,000+ HOURS MUSIC STREAMING ON PANDORA
12,000+ NEW ADS POSTED ON craigslist
370,000+ MINUTES VOICE CALLS ON skype
98,000+ TWEETS
20,000+ NEW POSTS ON tumblr.
320+ NEW twitter ACCOUNTS
13,000+ iPhone APPLICATIONS DOWNLOADED
100+ NEW LinkedIn ACCOUNTS
1 associatedcontent NEW ARTICLES PUBLISHED
IN 60 SECONDS
100+ Answers.com 40+ YAHOO! ANSWERS
6,600+ NEW PICTURES ARE UPLOADED ON flickr
600+ NEW VIDEOS
50+ WORDPRESS DOWNLOADS
70+ DOMAINS REGISTERED
60+ NEW BLOGS
168 MILLION EMAILS ARE SENT
694,445 SEARCH QUERIES
1,700+ Firefox DOWNLOADS
695,000+ facebook STATUS UPDATES
125+ PLUGIN DOWNLOAD
1,500+ BLOG POSTS
79,364 WALL POSTS
510,040 COMMENTS
Google
Google Search



The term “big data” remains difficult to understand because it can mean so many different things to different people. Your understanding will be different if you look at big data through a technology lens, versus a business lens or industry lens.
Essentially, big data refers to two major phenomena:

- The breathtaking speed at which we are now generating new data
- Our improving ability to store, process and analyze that data

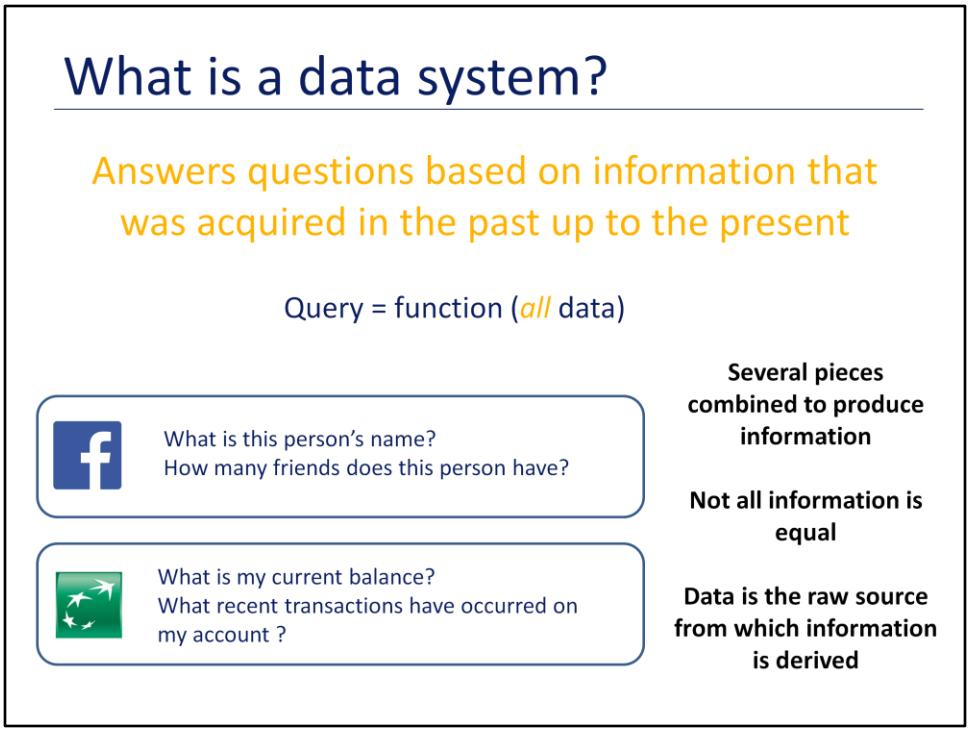
To describe the phenomenon that is big data, people have been using the four Vs: Volume, Velocity, Variety and Veracity.

Volume refers to the vast amount of data generated every second. Just think of all the emails, Twitter messages, photos, video clips and sensor data that we produce and share every second. We are not talking terabytes, but zettabytes or brontobytes of data. On Facebook alone we send 10 billion messages per day, click the like button 4.5 billion times and upload 350 million new pictures each and every day. If we take all the data generated in the world between the beginning of time and the year 2000, it is the same amount we now generate every minute! This increasingly makes data sets too large to store and analyze using traditional database technology. With big data technology we can now store and use these data sets with the help of distributed systems, where parts of the data is stored in different locations, connected by networks and brought together by software.
Velocity refers to the speed at which new data is generated and the speed at which

data moves around. Just think of social media messages going viral in minutes, the speed at which credit card transactions are checked for fraudulent activities or the milliseconds it takes trading systems to analyze social media networks to pick up signals that trigger decisions to buy or sell shares. Big data technology now allows us to analyze the data while it is being generated without ever putting it into databases.

Variety refers to the different types of data we can now use. In the past we focused on structured data that neatly fits into tables or relational databases such as financial data (for example, sales by product or region). In fact, 80 percent of the world's data is now unstructured and therefore can't easily be put into tables or relational databases—think of photos, video sequences or social media updates. With big data technology we can now harness differed types of data including messages, social media conversations, photos, sensor data, video or voice recordings and bring them together with more traditional, structured data.

Veracity refers to the messiness or trustworthiness of the data. With many forms of big data, quality and accuracy are less controllable, for example Twitter posts with hashtags, abbreviations, typos and colloquial speech. Big data and analytics technology now allows us to work with these types of data. The volumes often make up for the lack of quality or accuracy.



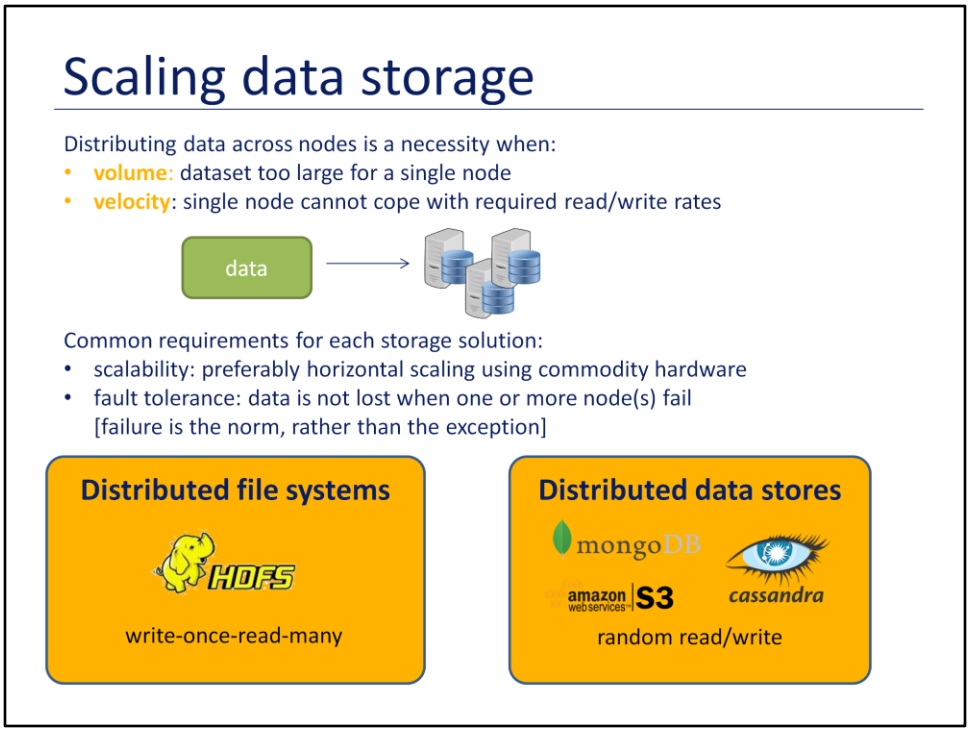
At the most fundamental level, a data system answers questions based on information that was acquired in the past up to the present. So a social network profile answers questions like "What is this person's name?" and "How many friends does this person have?". A bank account web page answers questions like "What is my current balance?" and "What transactions have occurred on my account recently"?

Data systems don't just memorize and regurgitate information. They combine bits and pieces together to produce their answers. A bank account balance, for example, is based on combining the information about all transactions on the account.
Another crucial observation is that not all bits of information are equal. Some information is derived from other pieces of information. A bank account balance is derived from a transaction history. A friend count is derived from a friend list, and a friend list is derived from all the times a user added and removed friends from their profile.
When you keep tracing back where information is derived from, you eventually end up at information that is not derived from anything. This is the rawest information you have: information you hold to be true simply because it exists. We call this information *data*.

STORING BIG DATA

Outline

- Types of distributed storage solutions
  - Distributed file system
    - case study: Hadoop DFS2
  - Distributed data stores
    - key-value; columnar; document; graph
- Reasons for distributing
  - sharding
  - replication



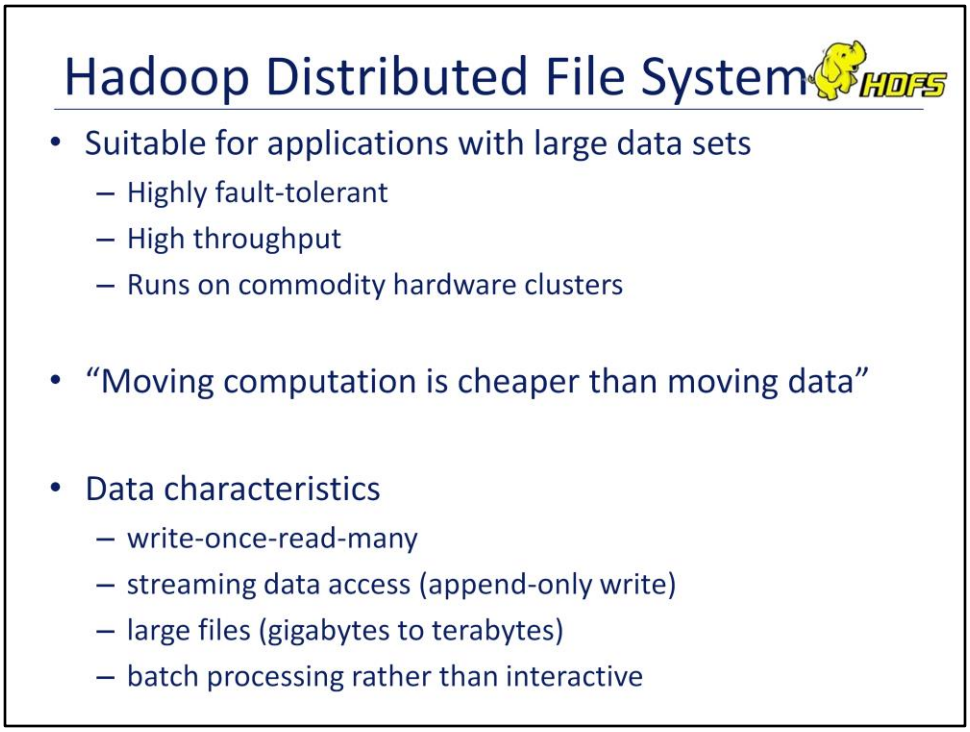
Storage solutions for big data have to be innately distributed. First, as the volume of big data grows, in the near or far feature you will reach the capacity limits of a single node. Second, in some cases a fast processing of big data is required. This means that there can be important constraints on the tolerable latency for a single read or write operation, even under heavy load with many concurrent users.

There is no one-size-fits-all solution for big data. At a high level, we can however distinguish two categories: file systems and data stores.
Distributed file systems, like the Hadoop Distributed File System (HDFS), are a perfect fit when data is written only once (e.g. raw sensor readings, logs, transaction data): the only write operation is to add *new* data (and not to modify existing data). Conversely, you want to perform regular calculations on this data, so reading operations are frequent.

Distributed data stores often prefer availability over consistency (cfr. CAP) and have advanced indexing mechanisms for rapid execution of random read/write messages. These kind of storage solutions are geared to update (overwrite) existing data. Distributed data stores are often grouped under the umbrella "NoSQL", but also relational databases can be distributed. We will discuss this later.

Both distributed file systems and distributed data stores have their unique strengths and compromises and even within a single category each available solution provides

different performance guarantees. It is up to the developer to pick the right tool – and to correctly engineer it. In the next slides, we will cover the major principles that should arm you to choose the best solution for the problem at hand.



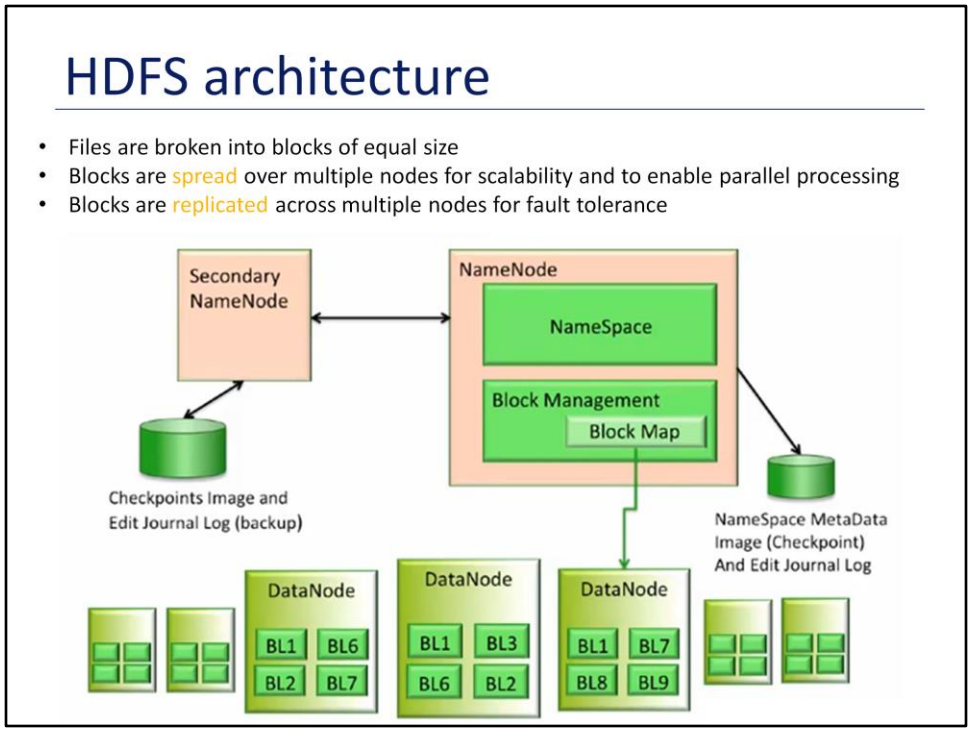
The Hadoop Distributed File System (HDFS) is a distributed and scalable file system that manages how data is stored across a cluster of commodity storage nodes. It has built-in capacities for fault-tolerance and favors high throughput reading operations, even on commodity hardware.

A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.

Applications built on top of HDFS need a write-once-read-many access model for files. A file once create, written and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A Map/Reduce application or a web crawler application fits perfectly with this model.

HDFS provides streaming data access: it is not possible to start reading from a random point in the file. Write operations are limited to appending to the end of an existing file, but this is very complex and should be avoided. If you want to add new data, then simply add an additional file to the filesystem.

The data should be organized in files of sufficient size (GBs per file), otherwise the efficiency of the distributed filesystem will degrade. As we will see later in more detail, application code is brought to the data nodes. Having many small files would mean that the application code has to be deployed on many data nodes, which introduces bookkeeping overhead.



HDFS is deployed across multiple serves, typically called a *cluster* and HDFS manages how data is stored across the cluster.

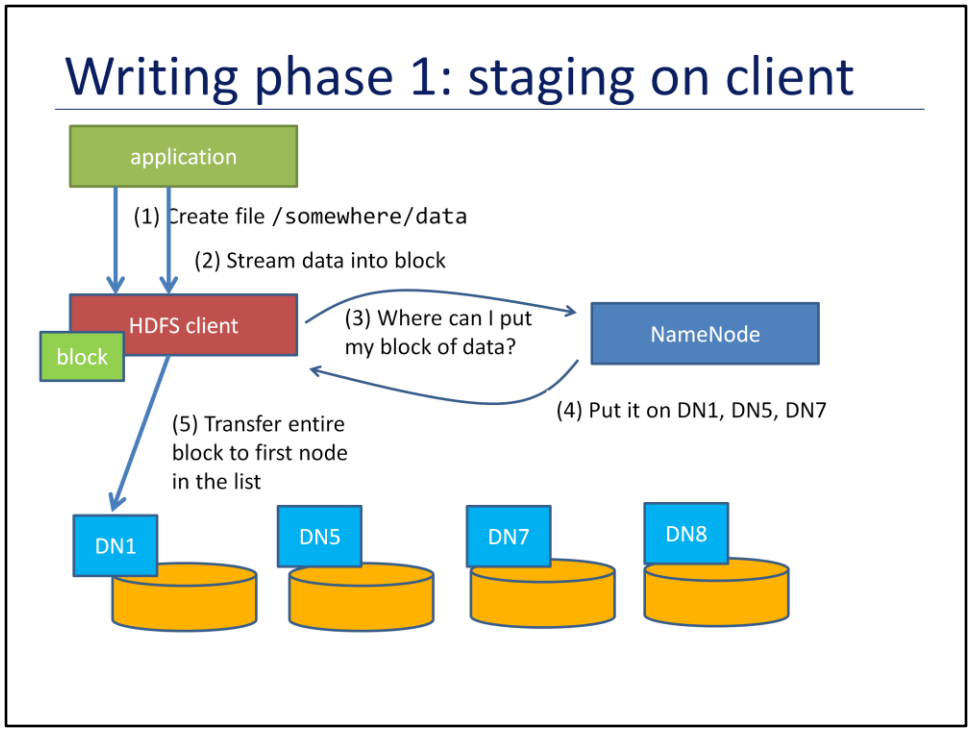
HDFS consists of **NameNode** and **DataNode** services providing the basis for the distributed filesystem. When you upload a file to HDFS, the file is first chuncked into blocks of a fixed size, typically between 64 MB and 256 MB. Each block is then replicated across multiple DataNodes (typically three).

The NameNode stores, manages, and serves the metadata of the filesystem, but does not store any real data blocks. When retrieving data, client applications first contact the NameNode to get the list of locations the requested data resides in and then contact the DataNodes directly to retrieve the actual data.

Hadoop brings in several performance, scalability, and reliability improvements.

- **High Availability** (**HA**) support for the HDFS NameNode: manual and automatic failover capabilities for the HDFS NameNode service, avoiding the NameNode single point of failure weakness
- **HDFS Federation** enables the usage of multiple independent HDFS namespaces in a single HDFS cluster. These namespaces would be managed by independent NameNodes, but share the DataNodes of the cluster to store the data. The HDFS federation feature improves the horizontal scalability of HDFS by allowing us to distribute the workload of NameNodes.

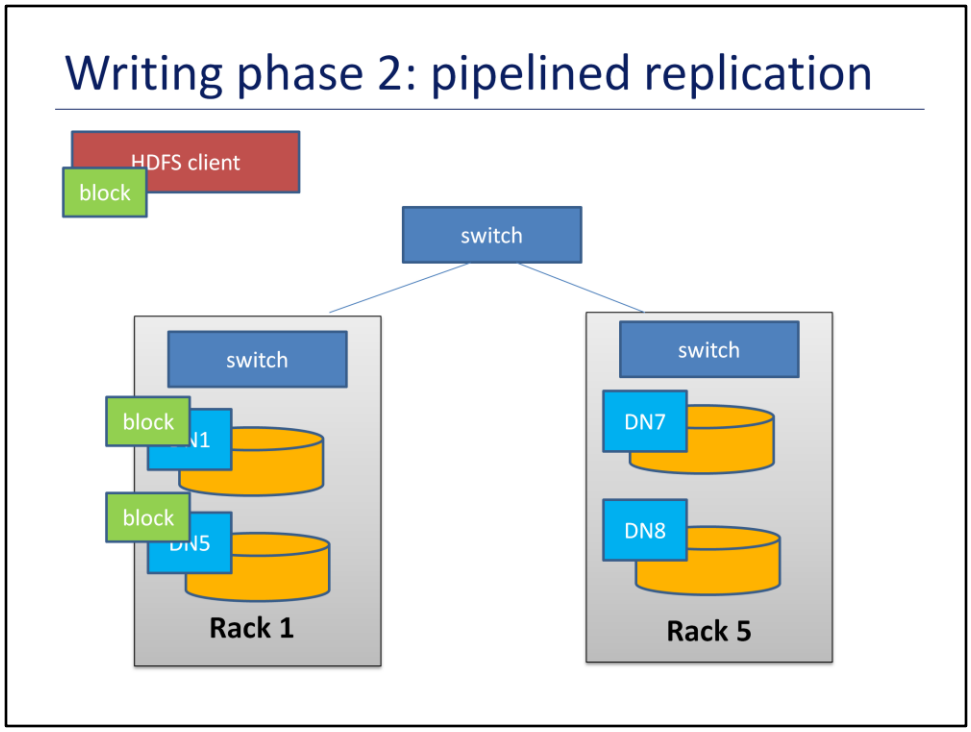
- Other important improvements of include the support for HDFS snapshots, heterogeneous storage hierarchy support and in-memory data caching support



A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it. The NameNode responds to the client request with the identity of the DataNode(s) and the destination data block. Then the client flushes the block of data from the local temporary file to the first of the specified DataNodes by communicating with the DataNode daemon (DNx boxes in the figure).

When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

This design approach has been adopted after careful consideration of target applications that run on HDFS. These applications need streaming writes to files. If a client writes to a remote file directly without any client side buffering, the network speed and the congestion in the network impacts throughput considerably. This approach is not without precedent.



When a client is writing data to an HDFS file, its data is first written to a local file as explained in the previous slide. Suppose the HDFS file has a replication factor of three. When the local file accumulates a full block of user data, the client retrieves a list of DataNodes from the NameNode. This list contains the DataNodes that will host a replica of that block. The client then flushes the data block to the first DataNode. The first DataNode starts receiving the data in small portions, writes each portion to its local repository and transfers that portion to the second DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next.

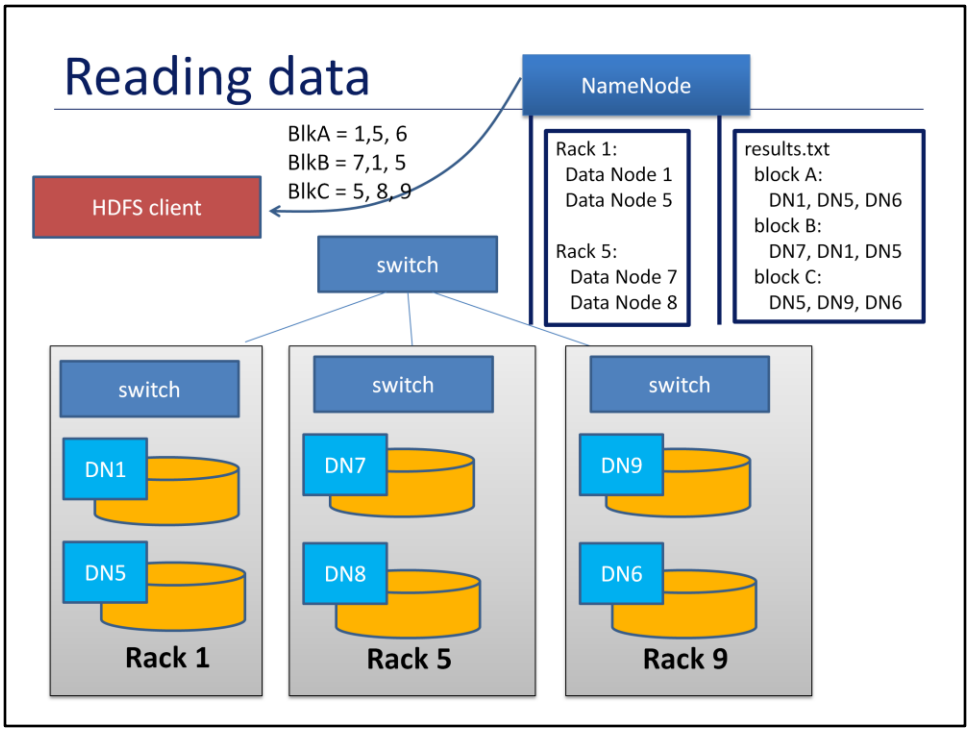
The placement of replicas is critical to HDFS reliability and performance. Optimizing replica placement is a feature that needs lots of tuning and experience. HDFS has the concept of “rack awareness”. The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization. The current default implementation for the replica placement policy is only a first effort and more sophisticated placement strategies may be introduced in the next versions.

Large HDFS instances run on a cluster of computers that commonly spread across

many racks. Communication between two nodes in different racks has to go through switches. In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks. The NameNode knows the rack id each DataNode belongs to.

A simple but non-optimal policy is to place replicas on unique racks. This prevents losing data when an entire rack fails and allows use of bandwidth from multiple racks when reading data. This policy evenly distributes replicas in the cluster which makes it easy to balance load on component failure. However, this policy increases the cost of writes because a write needs to transfer blocks to multiple racks.

For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a different node in the local rack, and the last on a different node in a different rack. This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure; this policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance.

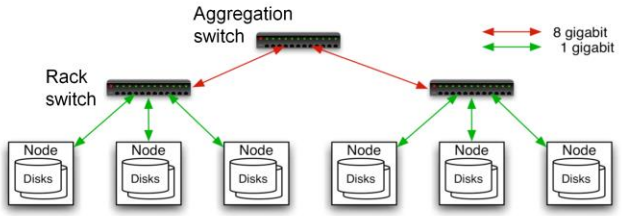


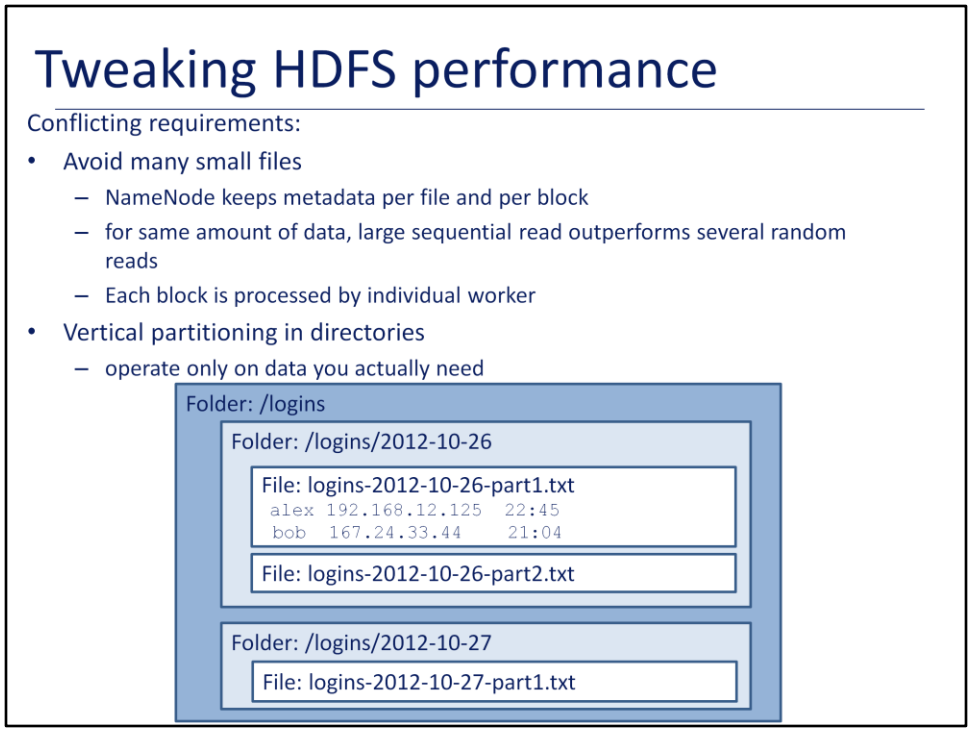
When a Client wants to retrieve a file from HDFS, perhaps the output of a job, it again consults the Name Node and asks for the block locations of the file. The Name Node returns a list of each Data Node holding a block, for each block. The client then picks one of the block location for each block.

To minimize global bandwidth consumption and read latency, the NameNode tries to satisfy a read request from a replica that is closest to the reader. The HDFDS client (the reader) is not always located outside of the cluster. In practice, read requests will often come from applications deployed on datanodes (see later, when we discuss YARN/MapReduce). If there exists a replica on the same rack as the reader node, then that replica is preferred to satisfy the read request. For this reason, the order in which the NameNode returns the list of nodes per block is important: the first node in the list is preferred from the viewpoint of cluster load. Unless the first node has crashed and the namenode has not noticed this yet, (legitimate) clients have no reason for not choosing the first replica in the list.

Data model for reading

- Client operation
  - query NameNode for block location
  - client accesses data directly from DataNode
- Rack-aware replica selection and placement
  - heartbeat: NameNode detects DataNode failures
  - NameNode balances disk usage and communication traffic





Small file problem

HDFS does not work well with lots of small files and instead wants fewer large files. A small file can be defined as any file that is significantly smaller than the Hadoop block size. The Hadoop block size is usually set to 64,128, or 256 MB, trending toward increasingly larger block sizes. However, the small file problem does not just affect small files. If a large number of files in your Hadoop cluster are marginally larger than an increment of your block size you will encounter the same challenges as small files. For example if your block size is 128MB but all of the files you load into Hadoop are 136MB you will have a significant number of small 8MB blocks. The good news is that this can be easily solved by choosing an appropriate (larger) block size.

There are two primary reasons that HDFS has a small file problem:

- **Namenode memory -** Every directory, file, and block in Hadoop is represented as an object in memory on the NameNode. As a rule of thumb, each object requires 150 bytes of memory. If you have a billion files each requiring just one block, this will require 300GB of memory and that is assuming every file is in the same folder! In addition, the NameNode must constantly track and check where every block of data is stored in the cluster. This is done by listening for data nodes to report on all of their blocks of data. The more blocks a data node must report, the more network bandwidth it will consume. Even with high-speed interconnects between the nodes, simple block reporting at this scale could become disruptive.
- **Processing delays –** As we will see later on, HDFS is typically used in conjunction

with batch processing tools like MapReduce. A large number of small files means a large number of random disk IO. Disk IO is often one of the biggest limiting factors in MapReduce performance. One large sequential read will always outperform reading the same amount of data via several random reads. A second processing delay comes from how batch processing frameworks work. Each block will be processed by an individual worker. In the example of MapReduce, each worker ("map task") is run in its own Java Virtual Machine. If you have 10 000 files each containing 10 MB of data, you will have the overhead of spinning up and tearing down just 10 000 Java Virtual Machines. If instead you have 800 files of 128 MB each, you only need 800 workers.

Vertical partitioning

Data is of course stored to be used by applications. You can make your processing application much more efficient if you partition your data so that the application can easily access data relevant to its computation. This process is called *vertical partitioning*, and avoid reading in and filtering out data that you don't need.

Vertically partitioning data on a distributed file system can be done by sorting your data into separate folders. For example, suppose you are storing login information on a distributed file system. Each login contains a username, IP address and timestamp. To vertically partition by day, you can create a separate folder for each day of data. Now if you only want to look at a particular subset of your dataset, you can just look at the files in those particular folders and ignore the other files.

As a file can only reside in one directory, this vertical partitioning means splitting the data in different files. Hence, you should trade-off the benefits of vertical partitioning with the performance penalty of smaller file sizes.

Outline

- Types of distributed storage solutions
  - Distributed file system
    - case study: Hadoop
  - Distributed data stores
    - key-value; columnar; document; graph
- Reasons for distributing
  - sharding
  - replication

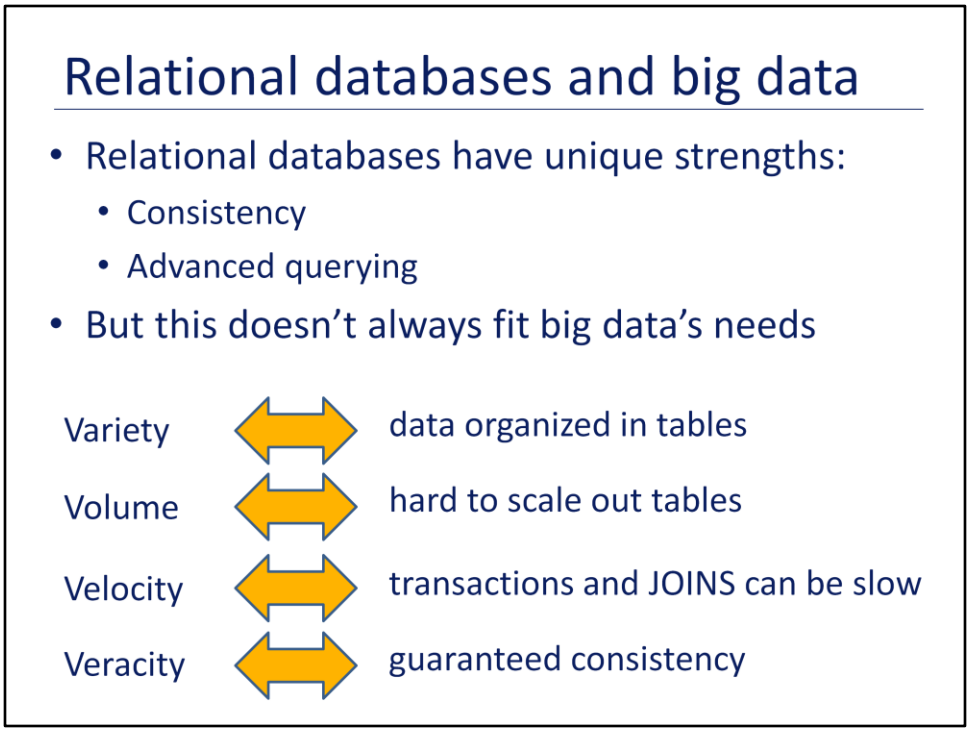
Distributed data stores

- Write-once-read-many does not fit all needs
  - fast random reads
  - real-time updating of data views
- Data stores provide different data model
  - data is indexed
  - data organization allows for fast random writes
- We will discuss the following aspects
  - matching big data with relational databases
  - types of distributed data stores and their trade-offs
  - techniques for distributing indexed data

The write-once-read-many paradigm of distributed file systems does not match all applications needs. Other applications may be better of with random read/write operations.

Databases are efficient in random reads because they index data. Moreover, many databases have advanced ways of organizing data, which favors fast random write operations.

As we will see, relational databases do not match well the requirements of many big data applications. We will study a new breed of data storage solutions – often referred to as "NoSQL". Our primary focus is on how these techniques ensure scalability and fault tolerance. Because these new technologies often lack support for SQL queries, we often refer to them as data *stores* rather than databases.

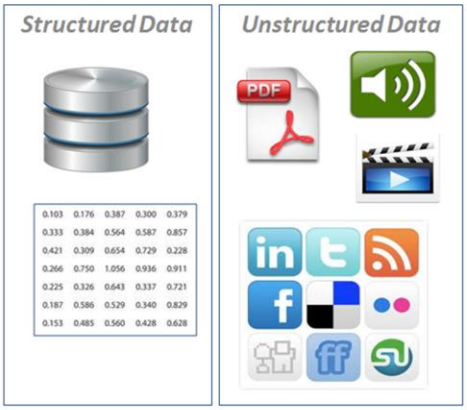


Relational databases provide solid, mature services according to the ACID properties. We get transaction-handling, efficient logging to enable recovery etc. These are core services of relational databases, and the ones that they are good at. They are hard to customize, and might be considered as a bottleneck, especially if you don't need them in a given application (e.g. serving website content with low importance).

Lots of "big data" problems don't require these strict constrains, for example web analytics, web search or processing moving object trajectories, as they already include uncertainty by nature.

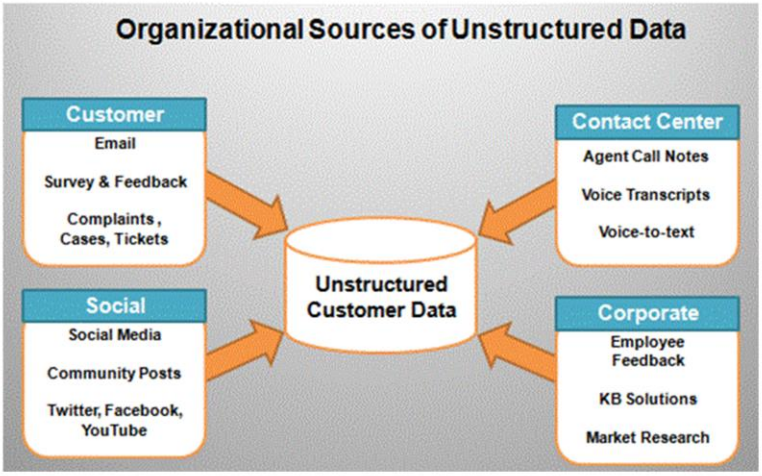
Variety: unstructured data

- big data is semi-structured or unstructured
  - blogs, e-mails, videos, tweets…
  - structure differs between records
  - cannot be mapped to tables (think Excel sheet) in a meaningful way



Big data is typically semi-structured. This means that the individual data records do not conform with a formal data model (as is the case in relational databases), but nonetheless they contain tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. These tags and markers make the data records a self-describing structure.

Variety example: customer data



More insight is gathered via text analysis than via SQL queries

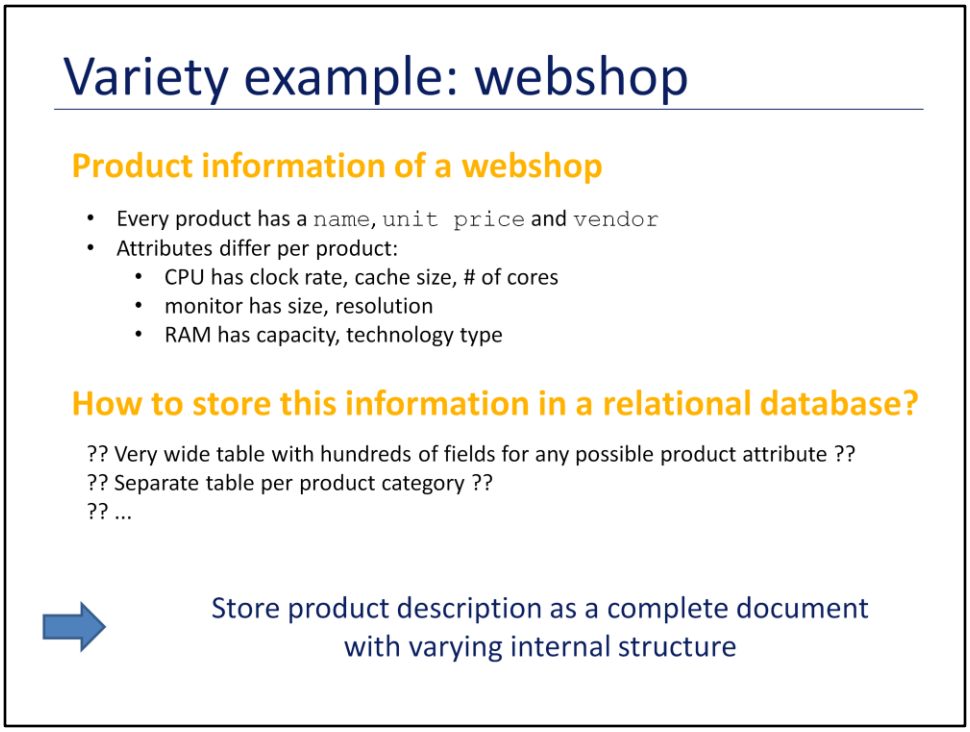
Unstructured data is often text-heavy. The useful knowledge from big data is gathered from applying text analysis on this text (e.g. using tools like MapReduce), rather than from launching advanced SQL queries.

Variety example: storing e-mails

Sender	Date	Body
serge.brin@abc.xyz	12-08-15 01:15 PM	.....
rector@ugent.be	14-08-15 08:05 AM	.....
...	...	...

- E-mails can be saved in semi-structured way
- Only very limited knowledge about customers can be derived using queries
  - What would be a useful query for the body column?

Continuing the example of the previous slide, let's look at the e-mails received. Although all e-mails have a body, it is hard to imagine a useful query for the body column.



Consider the example of a webshop. Each product (category) has many different attributes. It is not straightforward to map this efficiently in the table structure of a relational database.

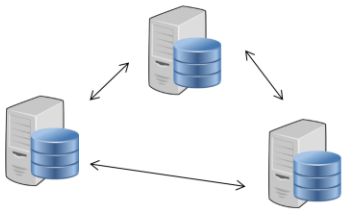
You could create a very wide table with hundreds of field for any possible attribute some product could have, but for most product most of these fields will be NULL. Adding new columns to a relational DB requires the system be shut down and ALTER TABLE commands to be run. When a database is large, this process can impact system availability, costing time and money.

You could have a separate table for each product category, thus introducing a lot of tables and relationships.

Although all options are valid, none of them is really satisfying. The fundamental problem is the rigid database schema structures imposed by relational DBs that do not match our needs.

Volume/Velocity: Distributing relational databases

- JOINs require data from multiple nodes
- ACID guarantees hard to maintain
  - Atomicity/Consistency/Isolation/Durability
  - enforced via transactions
    - intrinsically slower because of locking
    - CAP → system is unavailable from time to time



When the data becomes too large for a single node, it will be split over multiple nodes (shards). At a high level, this can be done in two ways for a relational DB: either you split the rows of a large table, either you distribute on a per-table basis (in practice, both approaches are combined). But in both ways multiple nodes are involved when we perform a JOIN operation: JOINs require the full data of each table being joined.

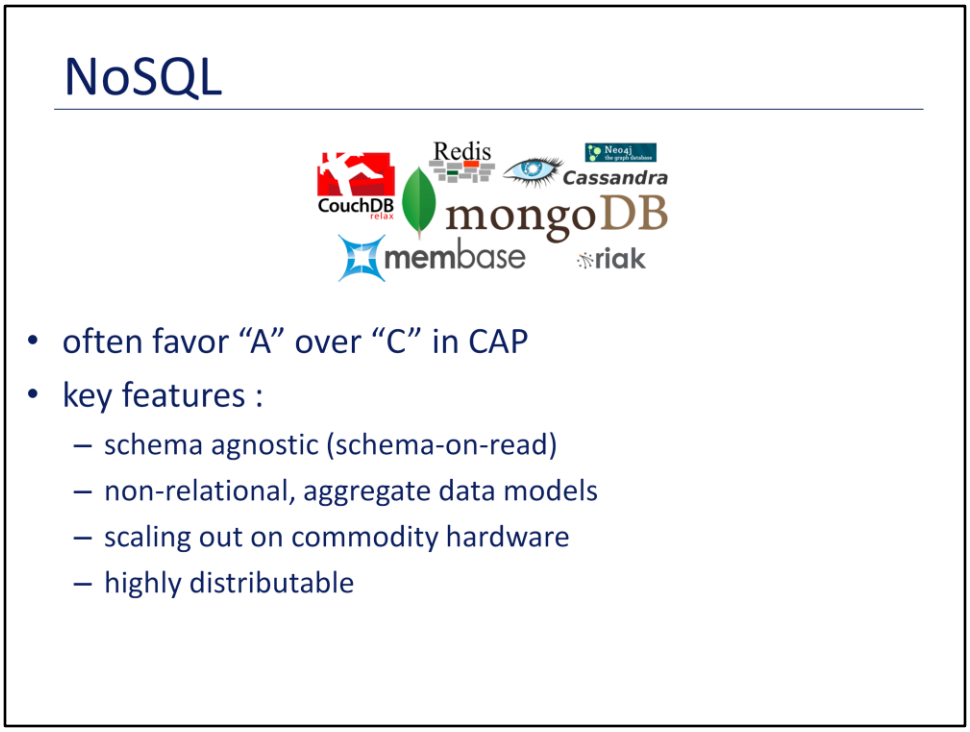
Another strength of relational DBs is their ACID-compliancy. In a relational database, the standards (and most of the existing products) are built around the concept of *transactions*, which are maintained in a definite order by a *transaction log*, which is by its nature a global scope data structure. The transaction log allows a database to guarantee that the current state of the database can be provably reached from some earlier state of the database by the application, in sequence, of a series of transactions. This is the basis of various ACID guarantees. ACID-compliant transaction means the database is designed so it absolutely will not lose data:

✓Each operation moves the database from one valid state to another (*A*tomic).
✓Everyone has the same view of the data at any point in time (*C*onsistent).
✓Operations on the database don't interfere with each other (*I*solation).
✓When a database says it has saved data, you know the data is safe (*D*urable).

Regarding Consistency (the "C" in ACID), real-time consistency is particularly difficult to implement in a partitioned system, because transactions that occur across (or

draw data from) multiple partitions need to be globally ordered ("serializable" in transaction terminology) in order to ensure that operations are drawing from a consistent view of the system.

NoSQL systems that use partitioning tend to solve this by relaxing the Consistency requirement such that it applies at a per-partition level, and not at a system level. This is a very reasonable engineering trade-off for most systems, and removes the greatest barrier for scalability that exists in distributed systems.



Four core features apply to most NoSQL databases.

Schema Agnosticism
A database schema is the description of all possible data and data structures in a relational database. With a NoSQL database, a schema isn’t required, giving you the freedom to store information without doing up-front schema design. You are not required to do a lot of up-front design work before you can storing data in NoSQL databases.

An alternative interpretation of schema agnosticism is *schema on read.* You need to know how the data is stored only when constructing a *query* (a coded question that retrieves information from the database), so for practical purposes, this feature is exactly what it says: You need to know the schema on read.

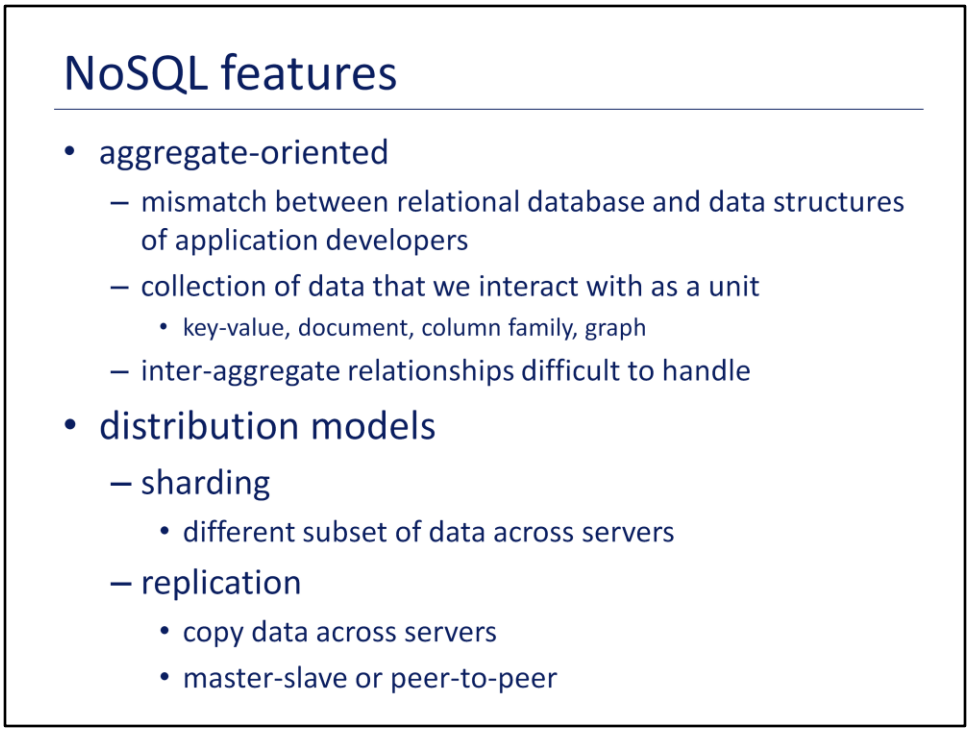
Nonrelational
Relations in a database establish connections between tables of data. For example, a list of transaction details can be connected to a separate list of delivery details. With a NoSQL database, this information is stored as an aggregate — a single record with everything about the transaction, including the delivery address.

Commodity hardware
Some databases are designed to operate best (or only) with specialized storage and

processing hardware. With a NoSQL database, cheap off-the-shelf servers can be used. Adding more of these cheap servers allows NoSQL databases to scale to handle more data.

Highly distributable

Distributed databases can store and process a set of information on more than one device. With a NoSQL database, a cluster of servers can be used to hold a single large database.



Application developers have been frustrated with the impedance mismatch between the relational data structures and the in-memory data structures of the application. Using NoSQL databases allows developers to develop without having to convert in-memory structures to relational structures.

Aggregate Data Models:
Relational database modelling is vastly different than the types of data structures that application developers use. Using the data structures as modelled by the developers to solve different problem domains has given rise to movement away from relational modelling and towards aggregate models. An aggregate is a collection of data that we interact with as a unit. These units of data or aggregates form the boundaries for ACID operations with the database. Key-value, Document, and Column-family data stores (all discussed later) can all be seen as forms of aggregate-oriented database.

Aggregates make it easier for the database to manage data storage over clusters, since the unit of data now could reside on any machine and when retrieved from the database gets all the related data along with it. Aggregate-oriented databases work best when most data interaction is done with the same aggregate, for example when there is need to get an order and all its details, it better to store order as an aggregate object but dealing with these aggregates to get item details on all the orders is not elegant.

Aggregate-oriented databases make inter-aggregate relationships more difficult to handle than intra-aggregate relationships. Aggregate-ignorant databases are better when interactions use data organized in many different formations.

Distribution Models:

Aggregate oriented databases make distribution of data easier, since the distribution mechanism has to move the aggregate and not have to worry about related data, as all the related data is contained in the aggregate. There are two styles of distributing data:

- **Sharding**: Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
- **Replication**: Replication copies data across multiple servers, so each bit of data can be found in multiple places. Replication comes in two forms,
    - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
    - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.

Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single server creating a single point of failure. A system may use either or both techniques.

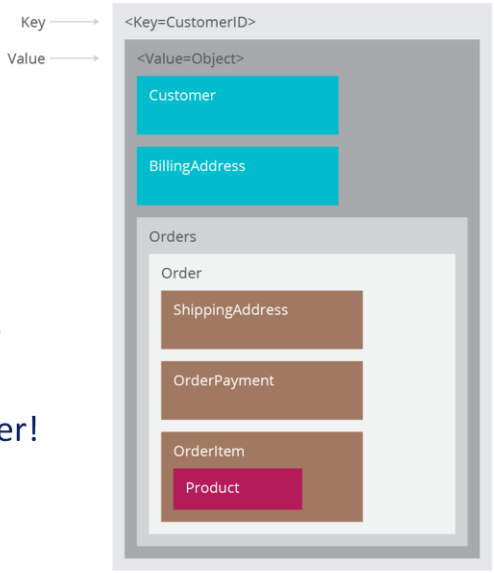
Common NoSQl aggregation types

Type	Description	Typical uses
Key-value	Associate large data file with a simple text string	Dictionary, image/document/file store, query cache, lookup tables
Graph	Store nodes and arcs of a graph	Social network queries, friend-of-friends queries, inference, rules system, pattern matching
Column family	Store a sparse matrix data using a row and column index	Web crawling, large sparsely populated tables, highly-adaptable systems, systems with high variance
Document	Store tree-structured hierarchical information in a single unit	Any data that has a natural container structure: office documents, sales orders, invoices, product descriptions, forms, web pages

One of the challenges for users of NoSQL systems is there are many different architectural patterns from which to choose. The table above lists the significant data architecture patterns associated with the NoSQL movement.

Key-value databases

- very simple API
- value is a “blob” for database
- easy to scale
- examples: Riak, Redis, Memcached, Amazon S3
  - properties may differ!



A *key-value store* is a simple database that when presented with a simple string (the key) returns an arbitrary large BLOB of data (the value). Key-value stores have no query language; they provide a way to add and remove key-value pairs (a combination of key and value where the key is bound to the value until a new value is assigned) into/from a database.

The dictionary is a simple key-value store where word entries represent keys and definitions represent values. Like the dictionary, a key-value store is also indexed by the key; the key points directly to the value, resulting in rapid retrieval, regardless of the number of items in your store.

One of the benefits of not specifying a data type for the value of a key-value store is that you can store any data type that you want in the value. The system will store the information as a BLOB and return the same BLOB when a GET (retrieval) request is made. It’s up to the application to determine what type of data is being used, such as a string, XML file, or binary image. All key-value databases are not the same, there are major differences between these products, for example: Memcached data is not persistent while in Riak it is, these features are important when implementing certain solutions. Let’s consider we need to implement caching of user preferences, implementing them in memcached means when the node goes down all the data is lost and needs to be refreshed from source system, if we store the same data in Riak we may not need to worry about losing data but we must also consider how to

update stale data. Its important to not only choose *a* key-value database based on your requirements, it's also important to choose *which* key-value database.

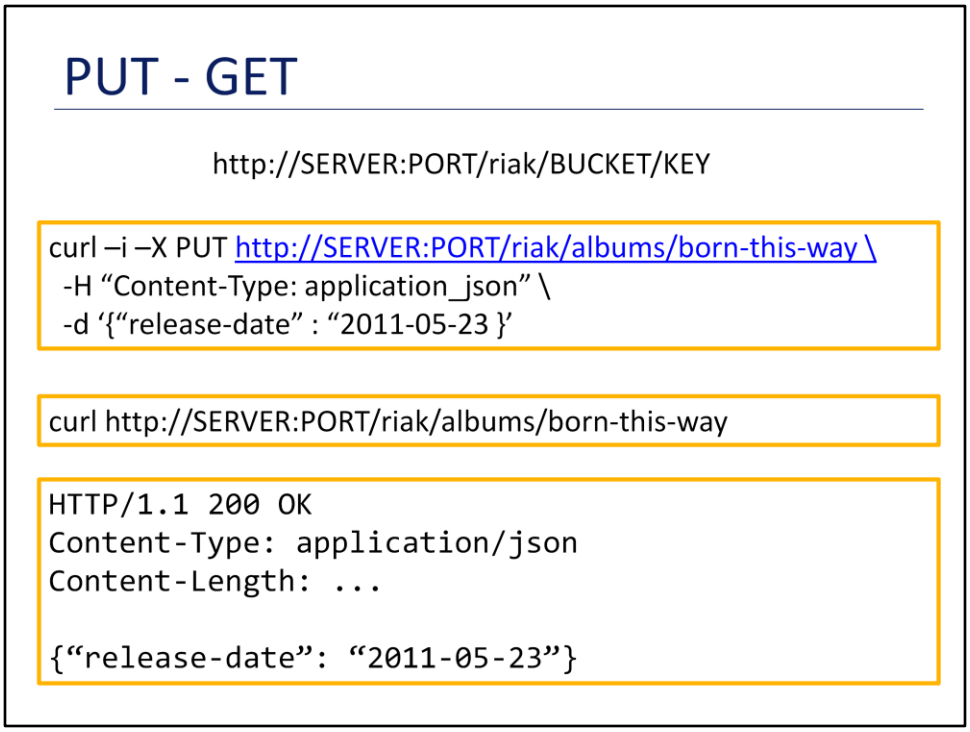
Example key-value: RIAK

- Key features:
  - Resilience
  - Link walking and support for map-reduce
- Keys are organized in buckets

- Values can be anything: XML, JSON, images…

Riak is a distributed key-value database where values can be anything—from plain text, JSON, or XML to images or video clips—all accessible through a simple HTTP interface. Riak is also fault-tolerant. Servers can go up or down at any moment with no single point of failure. But this flexibility has some trade-offs. Riak lacks robust support for ad hoc queries, and key-value stores, by design, have trouble linking values together
(in other words, they have no foreign keys). Riak attacks these problems by providing mechanisms like link walking and map reduce.

Riak breaks up classes of keys into *buckets* to avoid key collisions—for example, a key for java the *language* will not collide with java the *drink*.



Interfacing with Riak happens through HTTP requests. You query via URLs, headers, and verbs, and Riak returns assets and standard HTTP response codes.

In the example on the slide, we *put* a JSON file with metadata with key "born-this-way" in the bucket "albums". If you use a POST request to /riak/albums, then RIAK will generate the key itself. A GET request to the same location will retrieve the value.

RIAK – links to add metadata

One-way, multiple links per key possible

```
curl –i –X PUT http://SERVER:PORT/riak/albums/born-this-way \
 -H “Content-Type: application/json” \
 -H “Link: <riak/songs/judas>; risktag=\”lists_track\” “
 -d ‘{“release-date” : “2011-05-23 }’
```

```
curl –i –X PUT http://SERVER:PORT/riak/songs/judas \
 -H “Content-Type: application/json” \
 -H “Link: <riak/mp3s/62542>; risktag=\”is_audio_file\” “
 -d ‘{“release-date” : “2011-05-23 }’
```

```
curl –i –X PUT http://SERVER:PORT/riak/mp3/62542 \
 -H “Content-Type: audio/mpeg3” \
 -H “Link: <riak/mp3s/62542>; risktag=\”is_audio_file\” “
 -d ...
```

One of the ways that we are able to extend the fairly-limited data model provided by a key/value store is with the notion “links” and a type of query known as “link walking.”

Links are metadata that establish one-way relationships between objects by associating one key to other keys. The basic structure is this:

Link: </riak/bucket/key>; riaktag=\"whatever\"

The key to where this value links is in pointy brackets (<…>), followed by a semicolon and then a tag describing how the link relates to this value (any string). The links are attached by adding a “Link” header in the PUT (or POST) HTTP request. In the example we are attaching a link with tag “lists_track” to the key “Judas” in the bucket “songs”. We are also adding the key “Judas”, which points to a JSON with metadata about the song and has a link “is_audio_file” to the key corresponding with the correct mp3 in the bucket “mp3”. You can of course add multiple links to one object. The object that the link is pointing to should not even be in the database at the moment you register the link.

Links are unidirectional: an object is not aware of the links pointing to it.

RIAK – link walking

bucket, tag, keep

Get all objects in the bucket “songs” that are linked with tag “lists_track” on the album “born-this-way”:

```
curl http://SERVER:PORT/riak/albums/born-this-way/songs,lists_track,1
```

Get all objects in any bucket that are linked with tag “lists_track” on the album “born-this-way”:

```
curl http://SERVER:PORT/riak/albums/born-this-way/_,lists_track,1
```

Get all mp3s that are on the album “born-this-way”:

```
curl http://SERVER:PORT/riak/albums/born-this-way/
      _,lists_track,_/mp3,is_audio_file,1
```

Once you have tagged objects in Riak with links, you can then traverse them with an operation called “Link Walking.” With links, you create lightweight pointers between your data, for example, from 'projects' to 'milestones' to 'tasks', and then select data along that hierarchy using simple client API commands. This can substitute as a lightweight graph database, as long as the number of links attached to a given key are kept reasonably low. Links are an incredibly powerful feature of Riak.

Getting the linked data is achieved by appending a *link spec* to the end of the URL that is structured like this: /bucket,tag,keep

- Bucket – a bucket name to limit the links to
- Tag – the “riaktag” to limit the links
- Keep – 0 or 1, whether to return results from this step or phase

Each of these three elements can be replaced by an underscore (_) representing wildcards.

You can walk any number of links with one request by chaining multiple *link specs*. In the example above, we request from the database all objects in the bucket “mp3” that are pointed to by a link tagged “is_audio_file” from any object (in any bucket) that is inked with a tag “lists_track” from the object with key “born-this-way” in the bucket “albums”. By default, Riak will only include the objects found by the last step. If we would have put a “1” in the first link spec, then Riak would also return the intermediate objects in the GET response.

Document databases

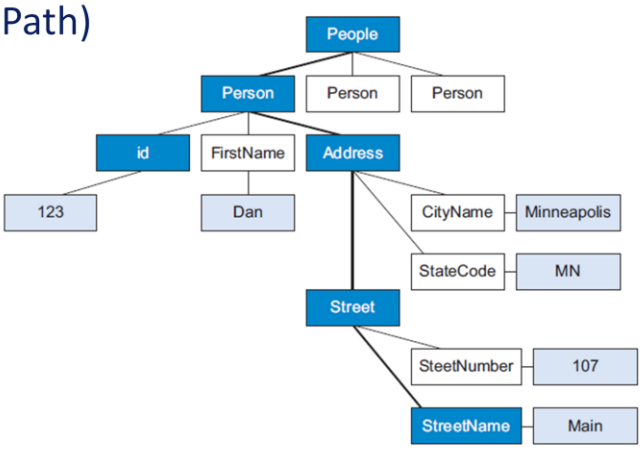
- Documents encapsulate and encode data in some standard format (XML, JSON...)
- Document is automatically indexed into tree-like structure
- Rich query language
- Example: MongoDB, CouchDB

Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. **The documents stored are similar to each other but do not have to be exactly the same**. Compared to relational databases, for example, collections could be considered analogous to tables and documents analogous to records. But they are different: every record in a table has the same sequence of fields, while documents in a collection may have fields that are completely different.

Document databases store documents in the value part of the key-value store. While key-value stores, when presented with a key, return a blob of data that lacks a formal structure and is not indexed or searchable, document stores work in the opposite manner: the key may be a simple ID which is never used or seen.
But you can get almost any item out of a document store by querying any value or content within the document. Document databases such as MongoDB provide a rich query language and constructs such as database, indexes etc allowing for easier transition from relational databases.

Document DBs: tree structure

Retrieve documents based on their content (cfr. XPath)



People/Person[id='123']/Address/Street/StreetName/text()

Think of a document store as a tree-like structure. Document trees have a single root element (or sometimes multiple root elements). Beneath the root element there is a sequence of branches, sub-branches, and values. Each branch has a related path expression that shows you how to navigate from the root of the tree to any given branch, sub-branch, or value. Each branch may have a value associated with that branch. Sometimes the existence of a branch in the tree has specific meaning, and sometimes a branch must have a given value to be interpreted correctly.

Each document store has an API or query language that specifies the path or path expression to any node or group of nodes in the tree. With the query on the slide you begin by selecting a subset of all people records that have the identifier 123. Often this points to a single person. Next you look in the Address section of the record and select the text from the Address street name. The full path name to the street name is the following: People/Person[id='123']/Address/Street/ StreetName/text().

Example document store: Mongo

- document-oriented
  - schemaless
  - JSON (BSON) format
- advanced server-side JavaScript queryability
  - ad hoc queries by field, range, regular expression
  - queries can include user-defined JS functions
  - map-reduce
  - indexing
- scalability and redundancy
  - load balancing
  - replication

Mongo is designed as a scalable database—the name Mongo comes from "hu*mongo*us"—with performance and easy data access as core design goals. Mongo hits a sweet spot between the powerful **queryability** of a relational database and the **distributed nature** of other datastores like Riak or HBase. Mongo is a JSON document database (though technically data is stored in a binary form of JSON known as BSON). A Mongo document can be likened to a relational table row without a schema, whose values can nest to an arbitrary depth. It enforces however no schema (similar to Riak), so documents can optionally contain fields or types that no other document in the collection contains.

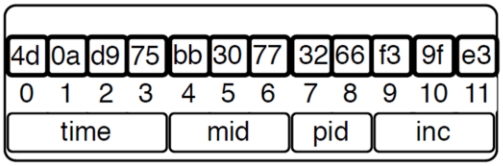
Despite being schemaless, Mongo offers extensive options for querying, including querying by user-defined functions, map-reduce functionality and indexing. These queries are written in JavaScript and executed at the server: the client only receives the result of the query (like with traditional SQL queries). We will discuss some of these features in the next slides.

Mongo's other strength lies in its ability to handle huge amounts of data (and huge amounts of requests) by replication and horizontal scaling.

Mongo: collections

- Interaction through JavaScript commands in Mongo shell
- Documents are inserted into collections (~ buckets in Riak)
- Each document is automatically given an ID
  - Different nodes in cluster generate non-colliding IDs

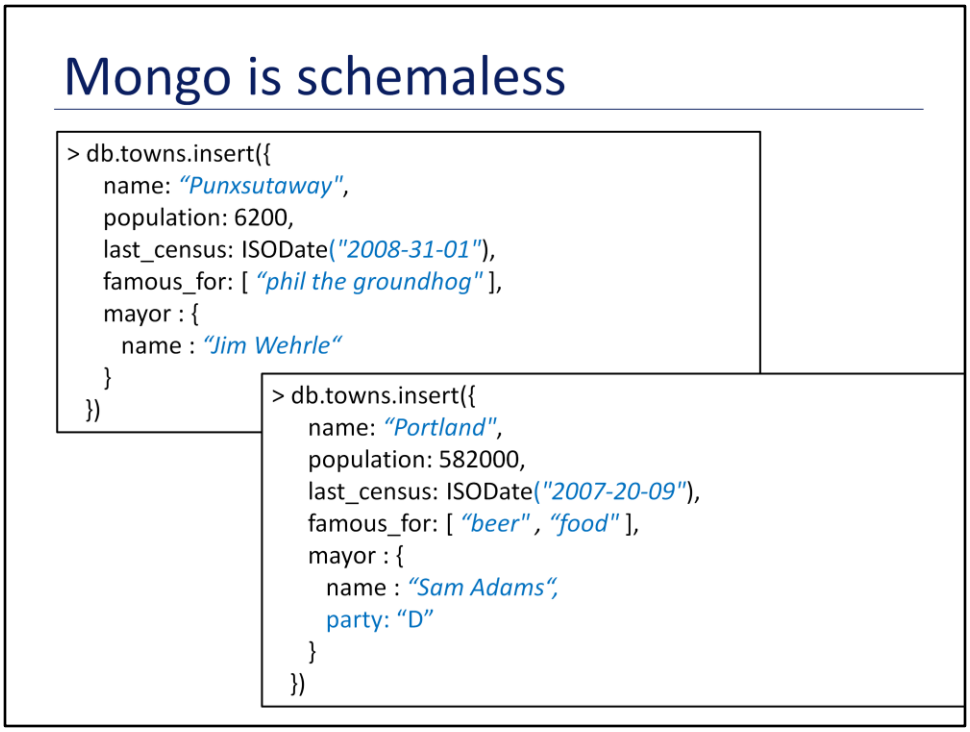
```
> db.towns.insert({
    name: "New York",
    population: 22200000,
    last_census: ISODate("2009-07-31"),
    famous_for: [ "statue of liberty", "food" ],
    mayor : {
      name : "Michael Bloo
      party : "I"
    }
  })
```



A mongo database comprises multiple *collections*. The concept of collections is similar to a bucket in Riak nomenclature. Interaction happens via JavaScript in a mongo shell. Once you have opened a mongo shell, the variable “db” is a JavaScript object that contains information about the current database. “db.x”is a JavaScript object representing a collection (named x). Commands on these functions are just JavaScript functions.

The command above shows how a new document is added to the collection “towns”. Since Mongo is schemaless, there is no need to define anything up front; merely using it is enough. In the example above, the collection towns didn’t exist before the first document was added to it.

Mongo automatically adds an _id field of type ObjectId to each JSON document stored. This is akin to a numeric key. The ObjectId is always 12 bytes, composed of a timestamp, client machine ID, client process ID, and a 3-byte incremented counter. The crux about this autonumbering scheme is that each process on every machine can handle its own ID generation without colliding. This design choice gives a hint of Mongo’s distributed nature.



The documents added in a single bucket can have varying structure. Not all documents need to have all fields, fields can be nested with arbitratry depth, etc...
In the example above, the political party of the mayor of Punxsutaway is unknown. In a relational database, we should enter NULL in the corresponding column. In a schemaless document store, we simply omit that value.
While the overlap in the current example is still relatively high; this needs not be the case: data items in the same collection can have a completely different structure.

Mongo: querying

```
> db.towns.find(
    { _id : ObjectId("4d0ada1fbb30773266f39fe4")}, { name : 1 })


{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "name" : "Punxsutawney"
}
```

```
> db.towns.find(
   { _id : ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 0 })

{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "population" : 6200,
  "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800 (PST)",
  "famous_for" : [ "phil the groundhog" ]
}
```

To access a specific document, you call the `find()` function. The first argument is a the `_id` of the document you want to retrieve. The `find()` function also accepts an optional second parameter: a fields object you can use to filter which fields are retrieved. If you want only the town name (along with `_id`), pass in "name" with a value resolving to 1 (or true).

Mongo: advanced querying

Use of regex and range operators:

```
> db.towns.find(
        { name : /^P/, population : { $lt : 10000 } },
        { _id : 0, name : 1, population : 1 }
  )

{ "name" : "Punxsutawney", "population" : 6200 }
```

Query by matching nested array data:

```
> db.towns.find(
        { famous_for : /statue/ },
        { _id : 0, name : 1, famous_for : 1 }
  )

{ "name" : "New York", "famous_for" : ["statue of liberty", "food" ]}
```

In Mongo you can construct ad hoc queries by field values, ranges, or a combination of criteria. The first example on the slide uses a regular expression and a range operator to find all towns that begin with the letter P and have a population less than 10 000. The second example demonstrates the ability to match nested array data.

Mongo: matching subdocuments

```
db.towns.find(
   { 'mayor.party' : 'I' },
   { _id : 0, name : 1, mayor : 1 }
)

{
  "name" : "New York",
  "mayor" : {
    "name" : "Michael Bloomberg",
    "party" : "I"
}
```

```
> db.towns.find(
 { 'mayor.party' : { $exists : false } },
 { _id : 0, name : 1, mayor : 1 }
)

{ "name" : "Punxsutawney", "mayor" : { "name" : "Jim Wehrle" } }
```

The true power of Mongo stems from its ability to dig down into a document and return the results of deeply nested subdocuments. To query a subdocument, your field name is a string separating nested layers with a dot.

The examples on the slide shows how to find all towns with independent mayors, or those with mayors who don't have a party.

Mongo: indexing

Collection of phone number documents:

```
> db.phones.find().limit(2)
{ "_id" : 18005550000, "components" : { "country" : 1, "area" : 800,
  "prefix" : 555, "number" : 5550000 }, "display" : "+1 800-5550000"
}
{ "_id" : 88005550001, "components" : { "country" : 8, "area" : 800,
  "prefix" : 555, "number" : 5550001 }, "display" : "+8 800-5550001"
}
```

Create your own index on the `display` field

```
> db.phones.ensureIndex(
   { display : 1 },
   { unique : true, dropDups : true }
  )
```

One of Mongo's useful built-in features is indexing to increase query performance — something that's not available on all NoSQL databases. MongoDB provides several of data structures for indexing, such as the classic B-tree, and other additions such as two-dimensional and spherical GeoSpatial indexes.

To demonstrate the feature, we are going to do a little experiment of MongoDB's B-tree index on a collection with a series of phone number documents. A B-tree is a generalization of a binary search tree in that a node can have more than two children.

Whenever a new collection is created, Mongo automatically creates an index by the `_id` field. Most queries will include more fields than just the `_id`, so we need to make indexes on those fields. In our example, we create a B-tree index on the `display` field by calling `ensureIndex(fields, options)`. The fields parameter is an object containing the fields to be indexed against. The options parameter describes the type of index to make. In this case, we are building a unique index on display that should just drop duplicate entries.

Speed of indexing

Before indexing:

```
> db.phones.find({display: "+1 800-5650001"}).explain()
{ "cursor" : "BasicCursor",
   "nscanned" : 109999,
   "nscannedObjects" : 109999,
   "n" : 1,
   "millis" : 52,
   "indexBounds" : { }
}
```

After indexing:

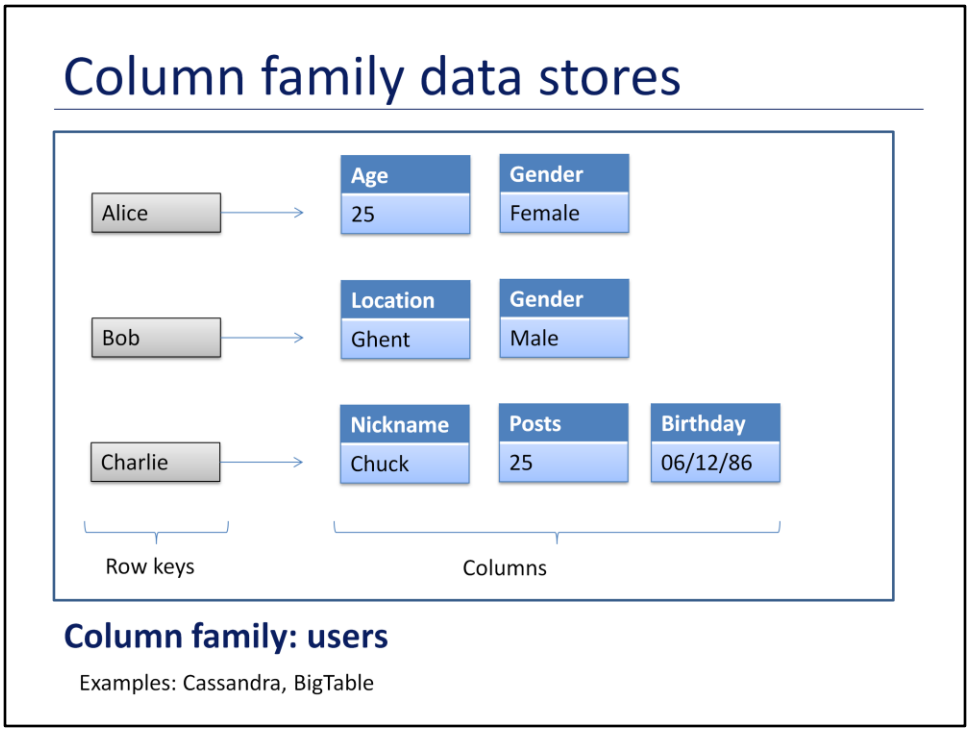
```
> db.phones.find({display: "+1 800-5650001"}).explain()
{ "cursor" : "BtreeCursor display_1",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 3,
  "indexBounds" : { "display" : [ [ "+1 800-5650001", "+1 800-5650001" ] ] }
}
```

The explain() method can be used to output details of a given operation. In the output, the millies field shows the time needed to complete the query.
When we rerun explain after indexing, the millies values will be dropped by at least an order of magnitude. Note how the cursor has changed from a Basic to a B-tree cursor. Mongo is no longer doing a full collection scan but instead walking the B-tree to retreive the value. Importantly, the number of scanned objects dropped from 109 999 to 1 since it has become just a unique lookup (based on the index).

Just like queries can be nested, you can buiild your index on nested values. If you wanted to index on all area codes, use the dot-notated field representation. In production, you should always build indexes in the background. This gives the following command:

```
> db.phones.ensureIndex({ "components.area": 1 }, { background : 1 })
```

Creating an index on a large collection can be slow and resource-intensive. You should always consider these impacts when building an index by creating indexes off-peak times, running index creation in the background, and running them manually rather than using automated index creation.

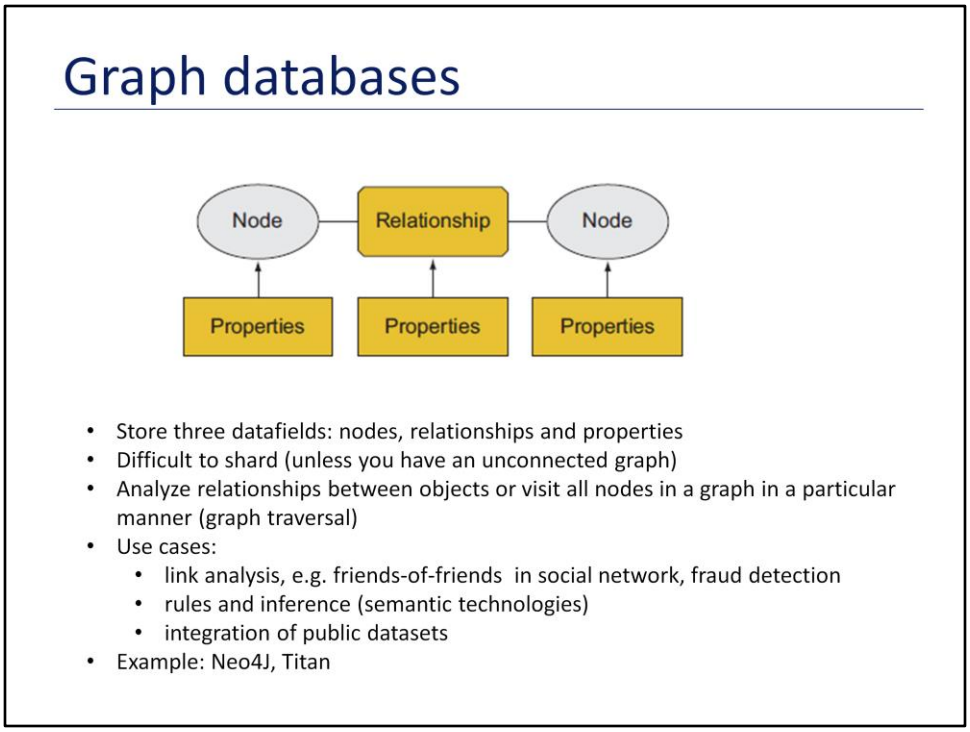


Column-family databases store data in column families as rows that have many columns associated with a row key. The row and column identifiers are used as general purpose keys for data lookup. Various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows.

This type is often referred to as *data stores* rather than *databases*, since this kind of storage solutions lacks features you may expect to find in traditional databases. For example, they lack typed columns, secondary indexes, triggers, and query languages.

An alternative view on the datamodel is to consider it as a map with sorted maps as values (or optionally, a sorted map with sorted maps as values). To expand upon the terms:

- Column families – column families are analogous to tables in relational databases, and each column family stores a completely independent set of information.
- Keys – if you consider a column family as a giant map, keys are the top-level entries in the map. Keys are used to partition a column family across a cluster
- Columns – each key points to another map of name/value pairs called columns. All columns for a key are physically stored together, making it inexpensive to access ranges of columns. Different keys can have different sets of columns, and it's possible to have millions of columns for a given key.



A *graph store* is a system that contains a sequence of nodes and relationships that, when combined, create a graph. A graph store has three data fields: *nodes*, *relationships*, and *properties*. Some types of graph stores are referred to as *triple stores* because of their node-relationship-node structure.

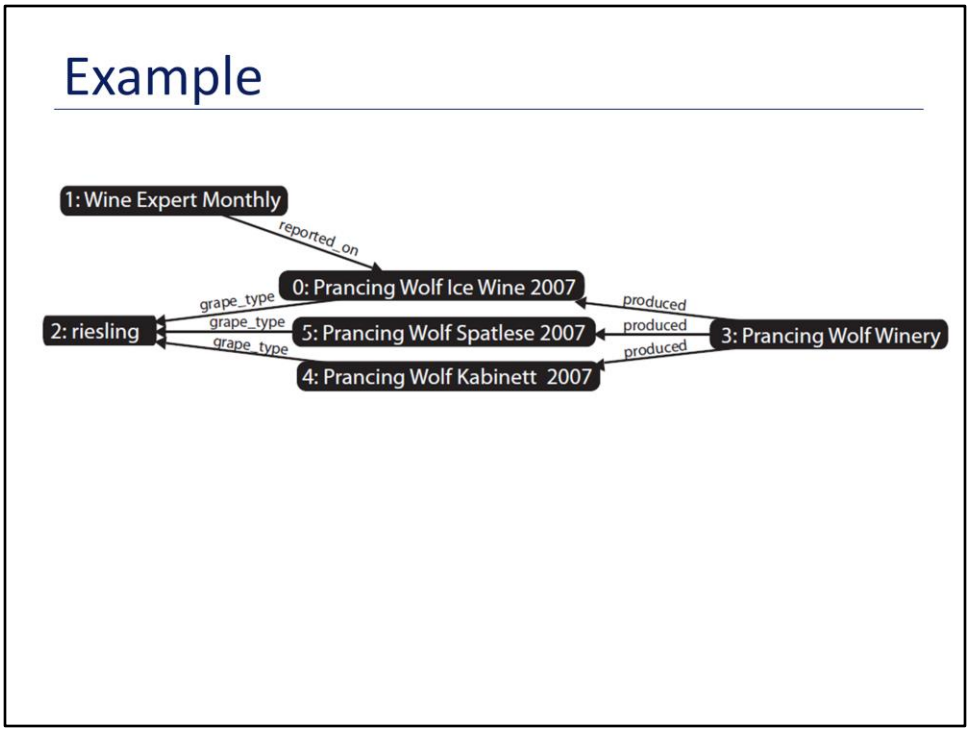
Unlike other NoSQL patterns, graph stores are difficult to scale out on multiple servers due to the close connectedness of each node in a graph. Data can be replicated on multiple servers to enhance read and query performance, but writes to multiple servers and graph queries that span multiple nodes can be complex to implement.

We list three uses cases where a graph store can be used to effectively solve a particular business problem:

- *link analysis:* for business problems that are solved by traversing graph data to search and look for patterns and relationships. The canonical example is to find friends of your friends in a social network, but graph stores are appropriate for identifying distinct patterns of connections between nodes. For example, creating a graph of all incoming and outgoing phone calls between people in a prison might show a concentration of calls (patterns) associated with organized crime. Analyzing the movement of funds between bank accounts might show patterns of money laundering or credit card fraud.
- *rules and inference:* relationships form the basis of the *semantic web stack* (e.g.

Using ontologies and querying languages like SPARQL).

- *dataset integration*: graph stores can be used to automatically join datasets that were created by different organizations (e.g. linked open data). This requires that nodes in the individual datasets can be unambigously correlated, so that they appear as one node in the merged graph.



This is an example of how data is modelled in graph data stores like Neo4J. The black blobs indicate nodes. The upper left node has a single property ("name" – not shown) with the value "Wine Expert Monthly". This magazine reviewed the wine "Prancing Wolf Ice Wine 2007", which is represented as a node with one property ("name"). The edge between these two nodes is annotated with the type of relationship: "reported_on".

This particular ice wine is created from the *riesling* grape. We could add this as a property directly to the wine node, but riesling is a general category that could apply to other wines. For this reason, we better create a new node and set its property to [name: "riesling"]. We also introduce a new relationship as grape_type and give it the property [style: "ice wine"].
Lastly, we do similar operations for the wine manufacturer that produces various wines.

If you want to see some more examples using Neo4J, you can refer to this slide deck: http://www.slideshare.net/peterneubauer/neo4j-5-cool-graph-examples-4473985

NoSQL = end of SQL?

NO ☺

- NoSQL is not a panacea/silver bullet
  - it answers fundamentally different data problems than relational DBs
  - it has different trade-offs (CAP, ACID)
- "SQL doesn't scale" is a myth!
  - Facebook does it...
  - but requires good engineering
- NewSQL databases: Google Spanner, ...
  - borrow some ideas from NoSQL
  - retain support for SQL queries and/or ACID

Outline

- Types of distributed storage solutions
  - Distributed file system
    - case study: Hadoop
  - Distributed data stores
    - key-value; columnar; document; graph
- Reasons for distributing
  - replication
  - sharding

The need for distributing data

You **will** run against the limitations of a single node

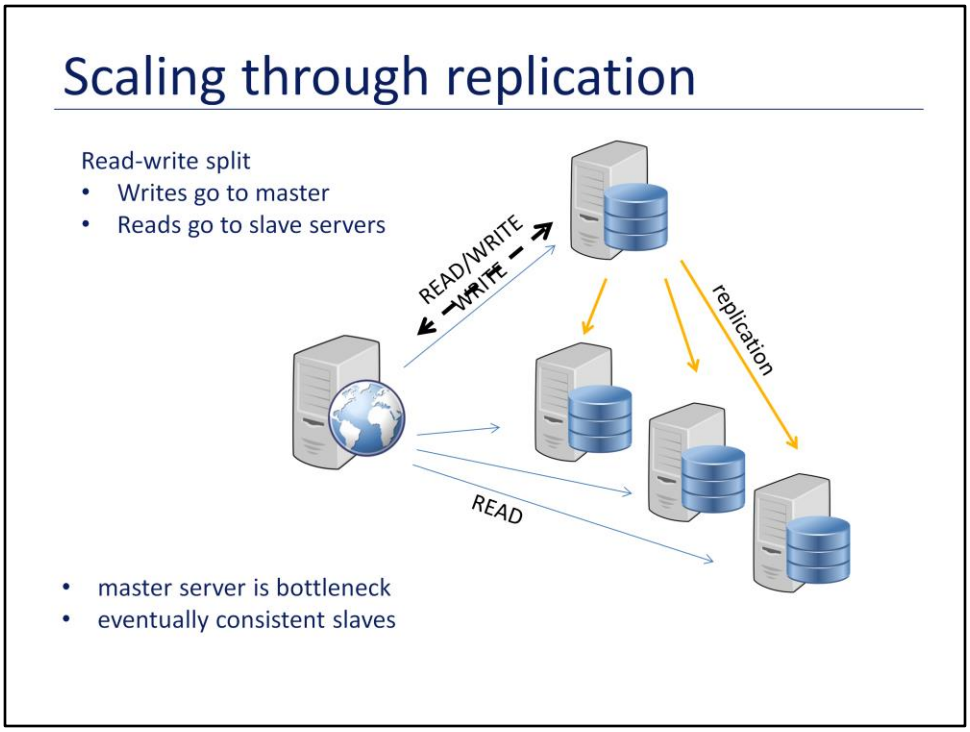
- CPU, memory, disk speed, data size, network bandwidth

This is even more likely in the cloud

- mostly commodity hardware
- multi-tenancy causes resource contention

Database systems with large data sets and high throughput applications can challenge the capacity of a single server. High query rates can exhaust the CPU capacity of the server. Larger data sets exceed the storage capacity of a single machine. Finally, working set sizes larger than the system's RAM stress the I/O capacity of disk drives. Also the network bandwidth from/to the database node can be bottleneck.

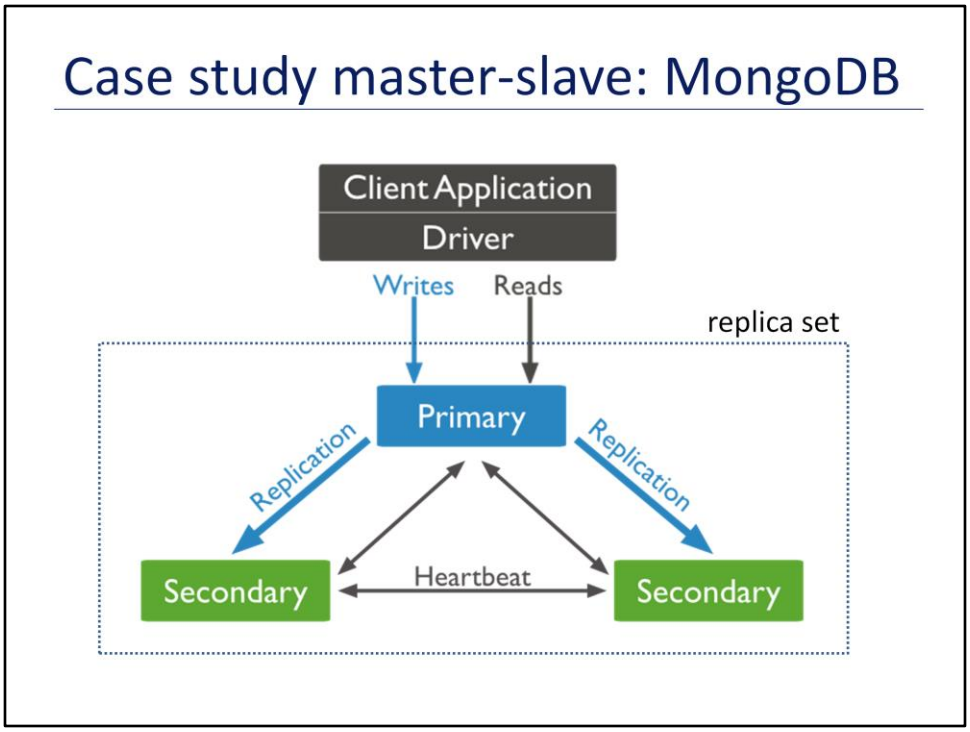
This limited capacity is exacerbated with cloud database services because the database is running on commodity hardware and the database server is multitenant.



Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup. In some cases, you can use replication to increase read capacity. Clients have the ability to send read and write operations to different servers. You can also maintain copies in different data centers to increase the locality and availability of data for distributed applications.

Master-Slave partitioning is the simplest option, with a single Master server for all write (Create Update or Delete) operations, and one or many additional Slave servers that provide read-only operations. The Master uses standard, near-real-time database replication to each of the Slave servers. The Master/Slave model can speed overall performance to a point, allowing read-intensive processing to be offloaded to the Slave servers, but there are several limitations with this approach:

- The single Master server for writes is a clear limit to scalability, and can quickly create a bottleneck.
- Slaves are *eventually consistent*, meaning that the Slave servers are not guaranteed to have a current picture of the data that is in the Master (but they will later). While this is fine for some applications, if your applications always require an up-to-date view, this approach is unacceptable.



A *replica set* in MongoDB is a group of `mongod` processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments. Mongod is the primary daemon process for the MongoDB system. It handles data requests, manages data access, and performs background management operations.

One mongod, the **primary**, receives all writes operations. A replica set can have only one primary. To support replication, the primary records all changes to its data sets in its log (MongoDB calls this the oplog). All other instances, **secondaries**, replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set.

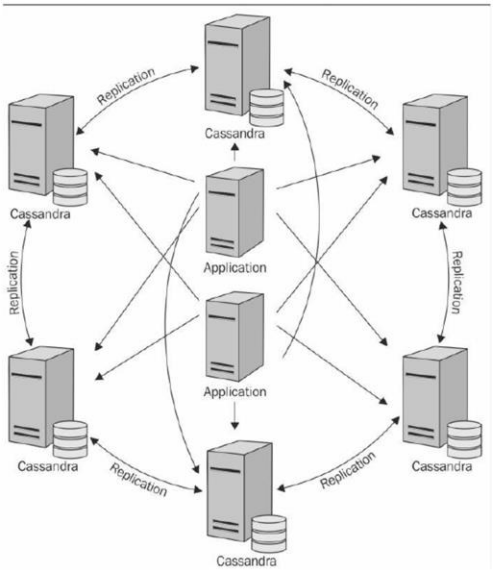
When a primary does not communicate with the other members of the set for more than 10 seconds, the replica set will attempt to select another member to become the new primary. The first secondary that receives a majority of the votes becomes primary.

By default, an application directs its read operations to the primary member in the replica set. Because write operations are issued to the primary, reading from the primary always returns the latest version of a document. Although it is possible to send read requests to secondary nodes, this is in general not recommended, because:

- All members of a replica have roughly equivalent write traffic; as a result, secondaries will service reads at roughly the same rate as the primary.
- Replication is asynchronous and there is some amount of delay between a successful write operation and its replication to secondaries; reading from a secondary can return out-of-date data.
- Distributing read operations to secondaries can compromise availability if *any* members of the set become unavailable because the remaining members of the set will need to be able to handle all application requests.
- Mongo runs a balancer in the background, who shifts chunks of data between nodes to balance node usage. For clusters with the balancer active, secondaries may return stale results with missing or duplicated data because of incomplete or terminated chunk migrations.

Sharding increases read and write capacity by distributing read and write operations across a group of machines, and is often a better strategy for adding capacity.

Case study ring: Cassandra



no special nodes

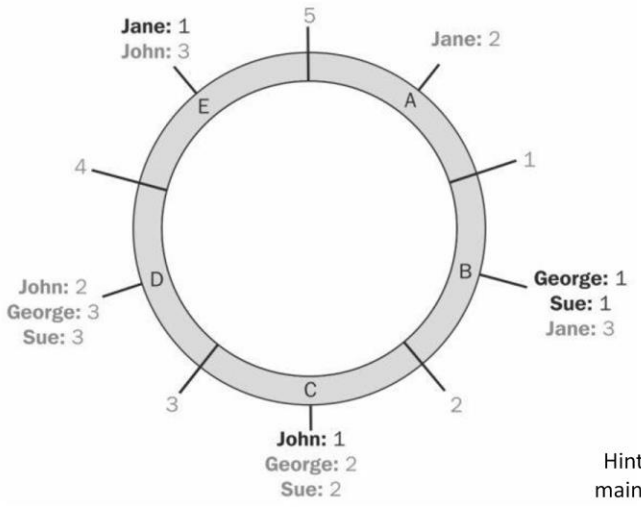
applications can query any node

supports rack- and datacenter-awareness

Unlike either monolithic or master-slave designs, Cassandra makes use of an entirely peer-to-peer architecture. All nodes in a Cassandra cluster can accept reads and writes, no matter where the data being written or requested actually belongs in the cluster. Internode communication takes place by means of a gossip protocol, which allows all nodes to quickly receive updates without the need for a master coordinator. Note that in contrast to the monolithic and master-slave architectures, there are no special nodes. In fact, all nodes are essentially identical, and as a result Cassandra has no single point of failure.

Cassandra employs a sophisticated replication system that allows fine-grained control over replica placement and consistency guarantees. There are two strategies available, one for single data center deployments and one when you have multiple datacenters.

Cassandra: replication in single data center



Hinted handoffs allow to maintain replication factor even when a replica node fails

We will only discuss the replication strategy in a single data center, assuming the default replication factor of three. As we will discuss later in more detail, data is assigned to the node in the cluster via a hash algorithm. Each node owns its own range of the hashed key space. The primary replica for each key is assigned to the node owning the hashed key value. Cassandra will then walk the ring in a clockwise direction to place each additional replica.

In the example on the slide, we have a cluster of 5 nodes (A-E), each covering some part of the hash keyspace. In the diagram, the keys in bold represent the primary replicas. Taking the hash of "Jane", we find that the first replica must be placed on node E. Then Cassandra takes the next two nodes (A and B) to place a replica.

Maintaining the replication factor when a node fails

You can specify a consistency level when you write to a node (see next slide). One key way in which Cassandra maintains fault tolerance even during node failure is through a mechanism called *hinted handoff*. If one of the replica nodes is unreachable during a write, then the system will store a hint on the coordinator node (the node that receives the write). This hint contains the data itself along with information about where it belongs in the cluster. Hints are replayed to the replica node once the coordinator learns via gossip that the replica node is back online.

Cassandra: tunable consistency

Consistency Level	Reads	Writes
ANY	Not supported	Data must be written to at least one node. Hinted handoffs tolerated.
ONE	Replica from closest node	Idem ANY; but no hinted handoffs tolerated
TWO	Replicas from two closest node	Idem ONE, but two replicas must be written
QUORUM	Replicas from a quorum will be compared and replica with latest timestamp is returned	Data must be writtento a quorum of replica nodes.
ALL	Idem QUORUM, but for all nodes	Data must be written to all replica nodes.

Closely related to replication is the idea of consistency between replicas. Cassandra is often described as an eventually consistent system, but it is more accurate to describe it as having tunable consistency. The precise degree of consistency guarantee can be specified on a per-statement level. This gives the application developer strong control over the trade-offs between consistency, availability, and performance at call level – rather than forcing a one-size-fits-all strategy.

On every read and write operation, the caller must specify a consistency level, which lets Cassandra know what level of consistency to guarantee for that one call. For any operation, it is possible to achieve either strong consistency or eventual consistency. In the former case, we can know for certain that the copy of the data that Cassandra returns will be the latest. In the case of eventual consistency, the data returned may or may not be the latest, or there may be no data returned at all if the node is unaware of newly inserted data. Under eventual consistency, it is also possible to see deleted data if the node you're reading from has not yet received the delete request.

There are numerous combinations of read and write consistency levels, all with different consistency guarantees. To illustrate this point, let's assume that you would like to guarantee absolute consistency for all read operations. On the surface, it might seem as if you would have to read with a consistency level of ALL, thus sacrificing availability in the case of node failure. But there are alternatives depending on your use case.

There are actually two additional ways to achieve strong read consistency:

Write with consistency level of ALL:
This has the advantage of allowing the read operation to be performed using ONE, which lowers the latency for that operation. On the other hand, it means the write operation will result in UnavailableException if one of the replica nodes goes offline.

Read and write with QUORUM or LOCAL_QUORUM:
Since QUORUM requires a majority of nodes, using this level for both the write and the read will result in a full consistency guarantee, while still maintaining availability during a node failure.

You should carefully consider each use case to determine what guarantees you actually require. For example, there might be cases where a lost write is acceptable, or occasions where a read need not be absolutely current. At times, it might be sufficient to write with a level of QUORUM, then read with ONE to achieve maximum read performance, knowing you might occasionally and temporarily return stale data. Cassandra gives you this flexibility, but it's up to you to determine how to best employ it for your specific data requirements.

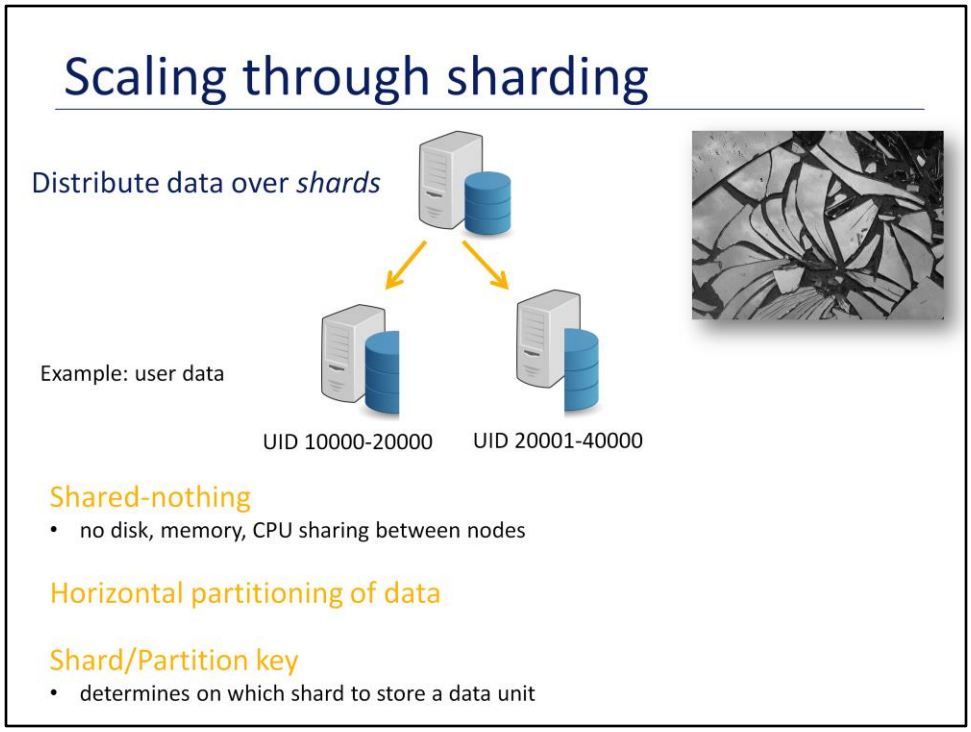
Balancing replication factor with consistency

Assume a cluster of **10 nodes**, with various replication factors (RF) and consistency levels (CL)

RF	Write CL	Read CL	Consistency	Availability
1	ONE QUORUM ALL	ONE QUORUM ALL	Consistent	No replica loss tolerated
2	ONE	ONE	Eventual	Tolerates loss of one replica
2	ONE	QUORUM ALL	Consistent	Tolerates loss of one replica on writes, but none on reads
3	ONE	ONE	Eventual	Tolerates loss of two replicas
3	ONE	QUORUM	Eventual	Tolerates loss of two replicas on write and one on reads

Achieving the desired availability, consistency, and performance targets requires coordinating your replication factor with your application's consistency level configurations.

In the table above, we consider a single data center cluster of 10 nodes and examine the impact of various configuration combinations on consistency and availability. We leave it as an exercise to the reader to study other combinations.



Database sharding provides a method for scalability across independent servers, each with their own CPU, memory and disk. The basic concept of database sharding is very straightforward: take a large database, and break it into a number of smaller databases across servers.

Sharding is a *horizontal* scaling strategy in which resources from each shard (or node) contribute to the overall capacity of the sharded database. Database shards are said to implement a *shared nothing* architecture that simply means that nodes do not share with other nodes; they do not share disk, memory, or other resources. Sharding thus supports high throughput and large data sets:

- Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows. As a result, a cluster can increase capacity and throughput *horizontally*. For example, to insert data, the application only needs to access the shard responsible for that record.
- Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows. For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data.

A specific database column designated as the *shard key* determines which shard node stores any particular database row. The shard key is needed to access data. As a naïve but easily understood example, the shard key is the *username* column and the first

letter is used to determine the shard. Any usernames starting with A-J are in the first shard, and K-Z in the second shard. When your customer logs in with their username, you can immediately access their data because you have a valid shard key.

How to shard?

There is no single secret sauce

- Some basic building blocks
- More about what *not* to do rather than a specific recipe
- Optimal scheme is highly application specific

It is important to note that Database Sharding is effective because it offers an application specific technique for massive scalability and performance improvements. The degree of effectiveness is directly related to how well the sharding algorithms themselves are tailored to the application problem at hand. There are numerous methods for deciding how to shard your data, and its important to understand your transaction rates, table volumes, key distribution, and other characteristics of your application.

There are multiple shard schemes possible, each designed to address a specific type of application problem. Each scheme has inherent performance and/or application characteristics and advantages when applied to a specific problem domain. In fact, using the wrong shard scheme can actually inhibit performance and the very results you are trying to obtain. It is also not uncommon for a single application to use more than one shard scheme, each applied to a specific portion of the application to achieve optimum.

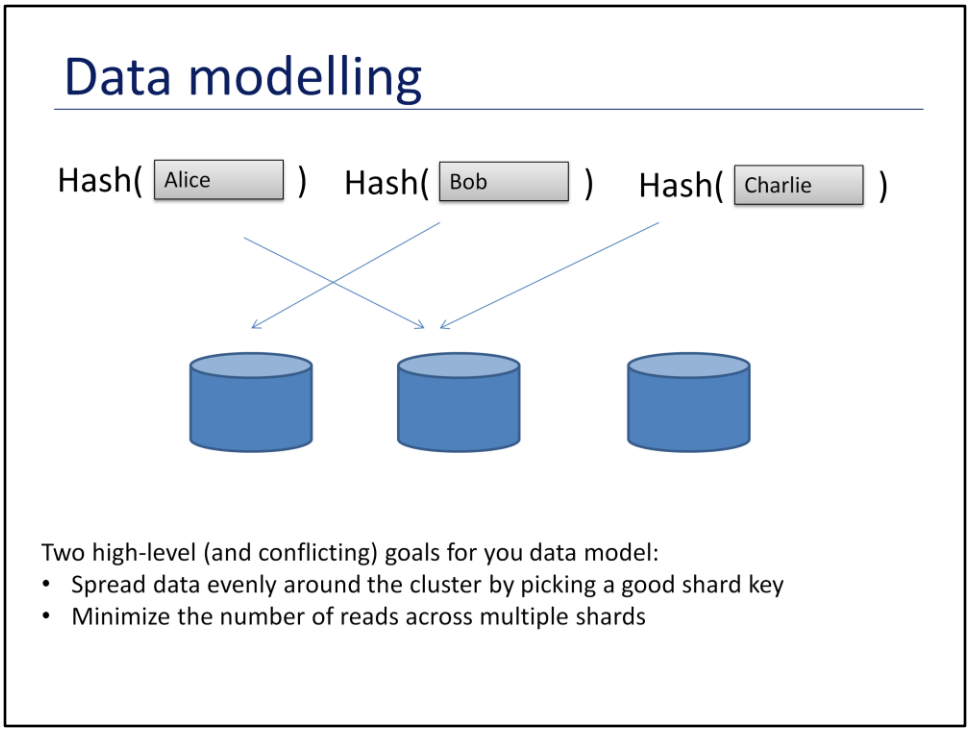
General rule: avoid hotspots

Avoid uneven distribution of data
and/or operations across the shards

Examples of cross-shard operations: sorting, counting, joining

Although data tier frameworks may allow this kind of operations, the response time of such operations will dramatically impact your application

You should organize your shards according to data access patterns. So it is important to pick the right sharding key.



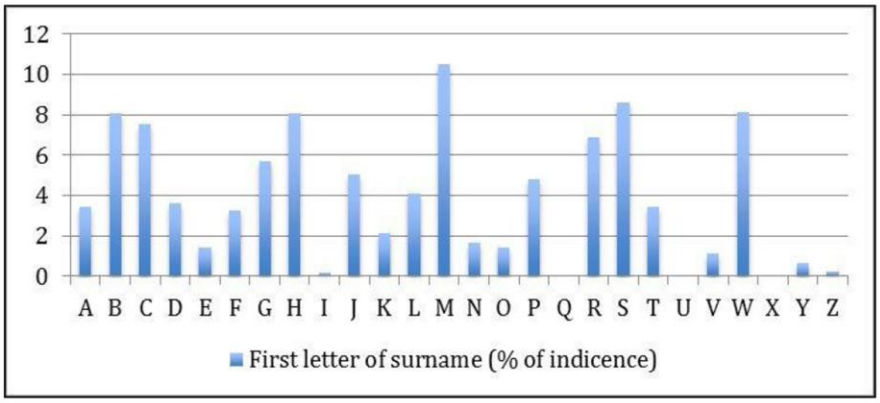
These are the two high-level goals for your data model:

1) Spread data evenly around the cluster
2) Minimize the number of partitions read

Data items (rows, documents…) are spread around the cluster bashed on a hash of the shard key. When you issue a read query, you want to read from as few partitions as possible, because each shard may reside on a different node. The node that receives a read request will generally need to issue separate commands to separate nodes for each partition you request. This adds a lot of overhead and increases the variation in latency. Furthermore, even on a single node, it's more expensive to read from multiple partitions than from a single one due to the way rows are stored.

These two goals often conflict and you need to balance these. We discuss this topic in more detail in the next slides.

Hotspots: uneven distribution of data

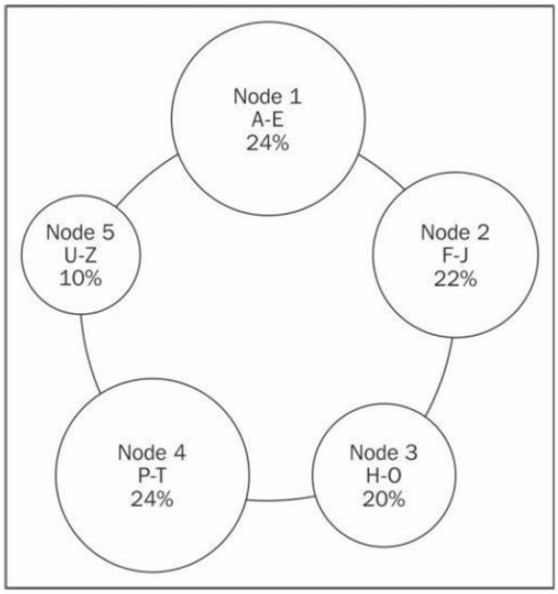


US Census Data, 2000

Let's assume, for example, that you're storing an address book, where the keys represent the last name of the contact. You use the last name of the contact to represent the keys. The graph above shows the distribution among the 26 letters, using 2000 United States Census Data.

As one would expect, last names in the United States are not evenly distributed by the first letter. In fact, the distribution is quite uneven. If we presume that each node owns a subset of the keys alphabetically, the result will resemble the diagram on the following slide.

Hotspots: uneven distribution of data



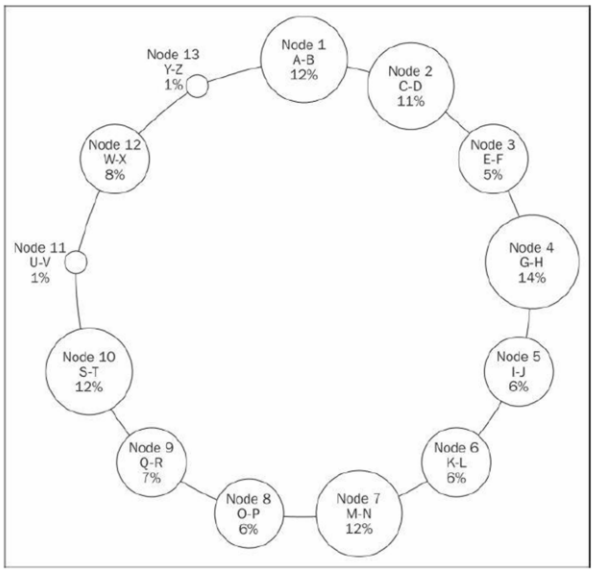
Imbalance on data placement

If queries follow popularity of data, then this also creates an impact on read/write operations

Using the last name as the shard key is likely to result in uneven distribution. Using the data from the previous slide, we see that we have created hotspots in node 1 and node 4, while node 5 is significantly underutilized.

One perhaps less obvious side effect of this imbalance is the impact on reads and writes. If we presume that both reads and writes follow the same distribution as the data itself (which is a logical assumption in this case), the heavier data nodes will also be required to handle more operations than the lighter data nodes.

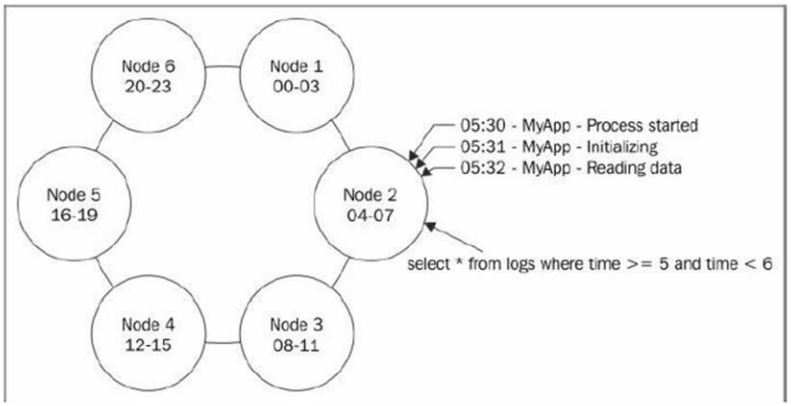
Hotspots: uneven distribution of data



Problem becomes worse in larger clusters

As is often the case in large systems, scaling out does not help to address this problem. In fact, the imbalance only gets worse when nodes are added. Still using the same data distribution from the previous example, the slide now shows the imbalance in a cluster of size 13. While in the five-node cluster, only one node was significantly underutilized, the larger cluster has eight out of 13 nodes doing half or less than half of the work as compared to the other nodes. In fact, two of the nodes own almost no data at all.

Hotspots: imbalanced queries



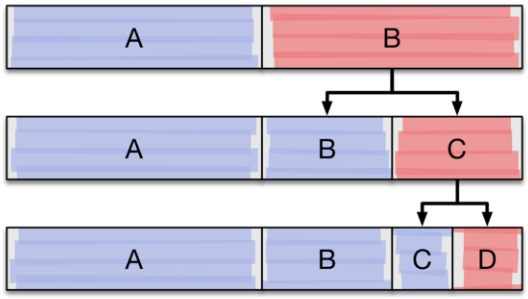
- Data placementis more balanced (assuming application is equally busy each hour)
- Read/write operations are focused on a single node

A common use case for big data storage systems is storing time-series data. Let's assume our use case involves writing log-style data, where we are always writing current timestamps and reading from relatively recent ranges of time.

Let's assume we have a six node cluster where the shard key corresponds to the time of day. If you are always writing current time, your writes will always go to a single node. Even worse, presuming you are reading recent ranges, your reads will also go to that same node.
Using the timestamp as shard key, time-series reads and writes will concentrate on a small subset of nodes. In the figure on the slide, node 2 is the only node doing any work. Each time the hour shifts, the workload will move to the next node in the ring. While the distribution of data in this model might be balanced (or it might not, depending on whether the application is busier at certain times), the workload will always experience hotspots.

Note: please do not conclude that using the timestamp as shard key is always a bad choice. This really depends on your query pattern. For example, if you calculate queries over the complete day, then data is equally retrieved from all nodes in the cluster.

Hotspots: imbalanced queries



Only last range will receive inserts!

This slide provides an alternative view on the uneven distribution of data write operations when using timestamps as shard key. Only the last shard will receive all write operations. Adding additional shards won't work, all requests keep going to the last shard.

A better approach is to use a combined partitioning (or shard) key. We will discuss this further in the Cassandra case study.

Bad sharding: “by application”

- Each service get its own node
- Result:
  - Data distribution is non-uniform, massive hot spots
  - Every data access pattern is unique
  - Very little efficiency of scale

Each service might have different access patterns to the data it needs. If we put all these databases on different nodes, then we organize a non-uniform access pattern across the nodes. Some nodes might be overwhelmed, while others might have spare capacity.

Sharding approaches

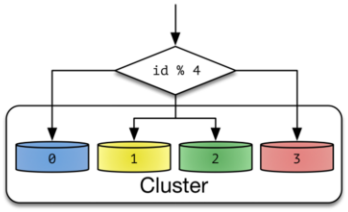
- NoSQL (key-value operations)
  - algorithmic sharding
  - dynamic sharding
    - case study: MongoDB
- SQL
  - entity grouping

This slide provides an overview of the different approaches to sharding we will study in the next slides.

NoSQL databases are often based on key-value operations. Mongo, Cassandra, RIAK all have some sort of key (or indexing). For this kind of databases, we have two typical approaches to sharding. 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.

Many (SQL) databases have more expressive querying and manipulation capabilities. They provide features such as joins, indexes and transactions that reduce complexity for an application. For this style of databases, we apply entity grouping: we store related entities in the same partition to provide additional capabilities within a single partition.

Algorithmic sharding



Sharding function maps partition key to node ID

Can be calculated **by application itself** (and provide as argument to query)

Data distribution does not consider payload size or space utilization

(over-simplistic) example:
user_ID % NUM_NODES

- Suitable for key-value databases with homogeneous values
- Resharding data is challenging:
  - Update sharding function (possibly in applications)
  - Move data around the cluster
- Example: memcached

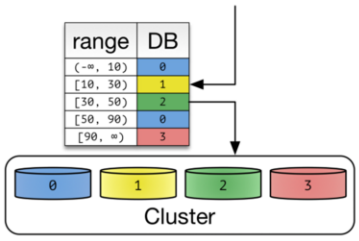
Algorithmically sharded databases use a sharding function *(partition_key) -> database_id* to locate data. A simple sharding function may be "*hash(key) % NUM_DB*".

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 haching (discussed later) 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.

Dynamic sharding



External *locator service* determines location of entries

Locators can be created, split and reassigned to redistribute data

To read and write data, clients need to consult the locator service first

- More resilient to non-uniform distribution of data and/load
- Locator service becomes single point of contention and failure
  - not simple to cache or replicate locators
- Auto-(re)sharding is possible, although challenging
- Used in many popular storage solutions
  - HDFS: Name Node
  - MongoDB: ConfigServer

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.

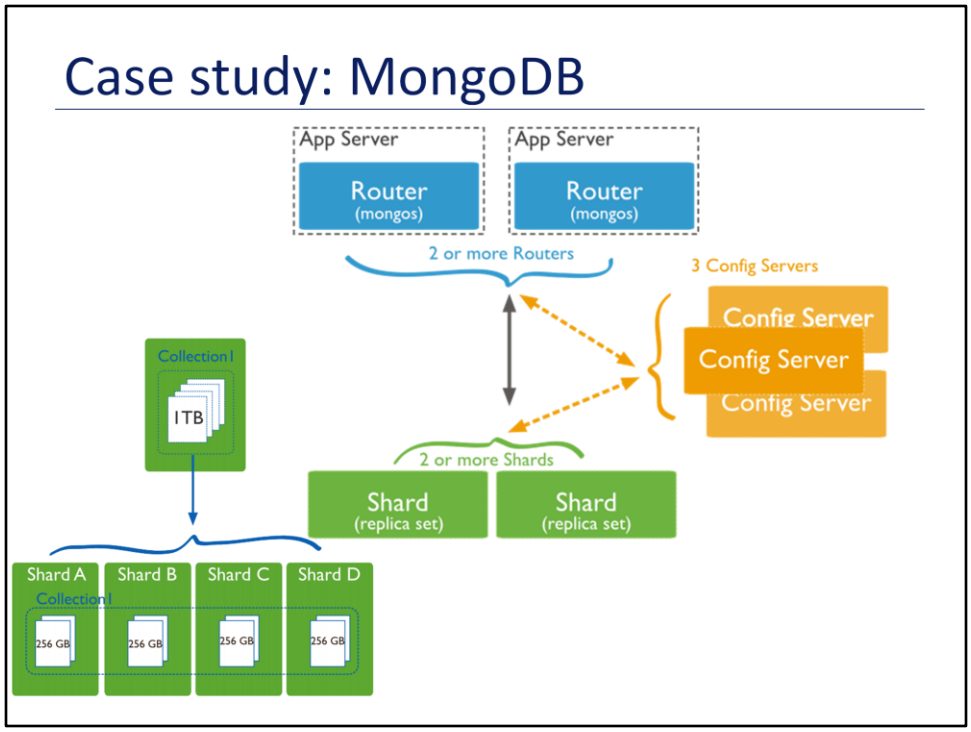
To read and write data, clients need to consult the locator service first. Operation by primary key becomes fairly trivial. Other queries also become efficient depending on the structure of locators. In the example of range-based partition keys, range queries are efficient because the locator service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

Dynamic sharding is more resilient to non-uniform 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 NameNode to store filesystem metadata. In **MongoDB**, the ConfigServer 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. (We'll explain all these Mongo terms in more detal later).



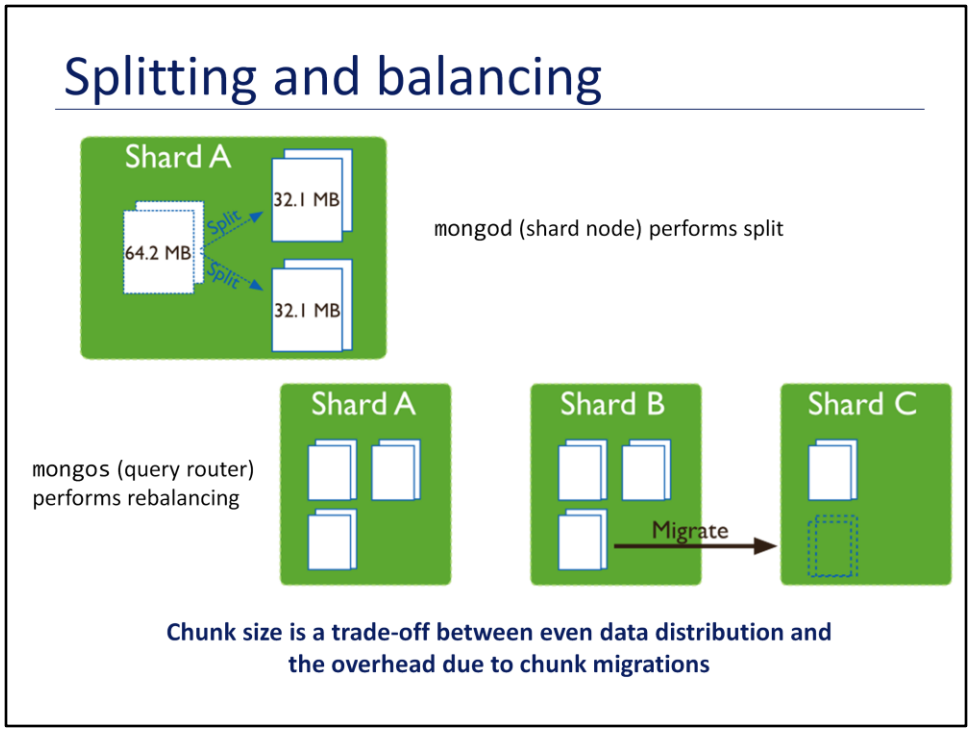
Sharding is an important part of how MongoDB achieves its scalability. MongoDB supports sharding through the configuration of *sharded clusters*.

A sharded cluster has three component types: shards, query routers and config servers.

Shards store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a replica set.

Query Routers, or `mongos` instances (mongos = mongo server daemon), interface with client applications and direct operations to the appropriate shard or shards. The query router processes and targets operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Most sharded clusters have many query routers.

Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. Production sharded clusters have exactly 3 config servers.



In MongoDB, data is split between shards using a shard key. The shard key is either an indexed field or an indexed compound field that exists in every document in the collection. MongoDB partitions data in the collection using ranges of shard key values. Each range, or **chunk**, defines a non-overlapping range of shard key values. MongoDB distributes the chunks, and their documents, evenly among the shards in the cluster.

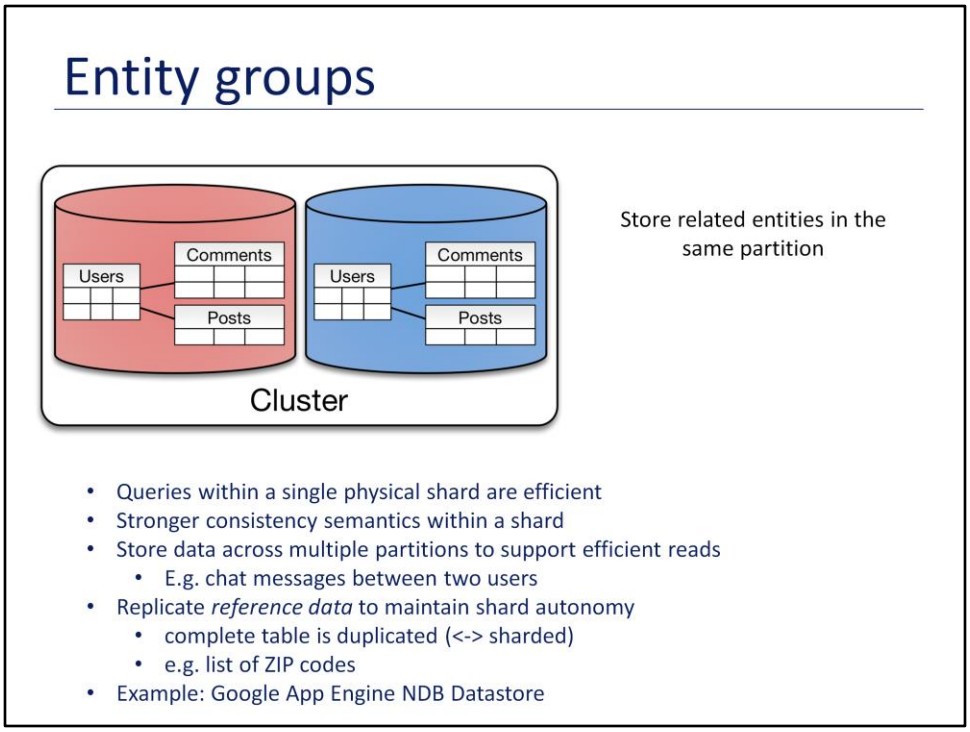
The addition of new data or the addition of new servers can result in data distribution imbalances within the cluster, such as a particular shard containing significantly more chunks than another shard or a size of a chunk is significantly greater than other chunk sizes. MongoDB ensures a balanced cluster using two background processes: splitting and the balancer.

Splitting is a background process that keeps chunks from growing too large. When a chunk grows beyond a specified chunk size, MongoDB splits the chunk in half. Inserts and updates of data trigger splits. Splits are an efficient meta-data change. To create splits, MongoDB does not migrate any data or affect the shards.

The balancer is a background process that manages chunk migrations. The balancer can run from any of the query routers in a cluster. When the distribution of a sharded collection in a cluster is uneven, the balancer process migrates chunks from the shard that has the largest number of chunks to the shard with the least number of chunks

until the collection balances. For example: if collection *users* has 100 chunks on shard 1 and 50 chunks on shard 2, the balancer will migrate chunks from shard 1 to shard 2 until the collection achieves balance. The shards manage chunk migrations as a background operation between an origin shard and a destination shard. During a chunk migration, the destination shard is sent all the current documents in the chunk from the origin shard. Next, the destination shard captures and applies all changes made to the data during the migration process. Finally, the metadata regarding the location of the chunk on config server is updated.

The default chunk size in MongoDB is 64 megabytes. Small chunks lead to a more even distribution of data at the expense of more frequent migrations. This creates expense at the query routing (mongos) layer. Large chunks lead to fewer migrations. This is more efficient both from the networking perspective *and* in terms of internal overhead at the query routing layer. But, these efficiencies come at the expense of a potentially more uneven distribution of data. For many deployments, it makes sense to avoid frequent and potentially spurious migrations at the expense of a slightly less evenly distributed data set.



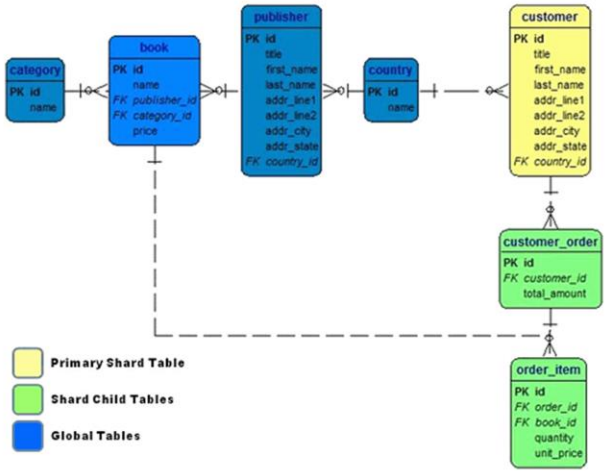
The previous discussion on algorithmic and dynamic sharding was geared towards key-value operations. But sharding can also be realized in relational databases through the concept of entity grouping.

The concept is very simple: store related entities in the same partition to provide additional capabilities within a single partition. These additional capabilities are:

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

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. Another technique is the replication of so-called global tables: the relatively static lookup tables that are common utilized when joining to much larger primary tables. Tables containing values as status codes, countries, types, and even products fall into this category.

Example: bookstore



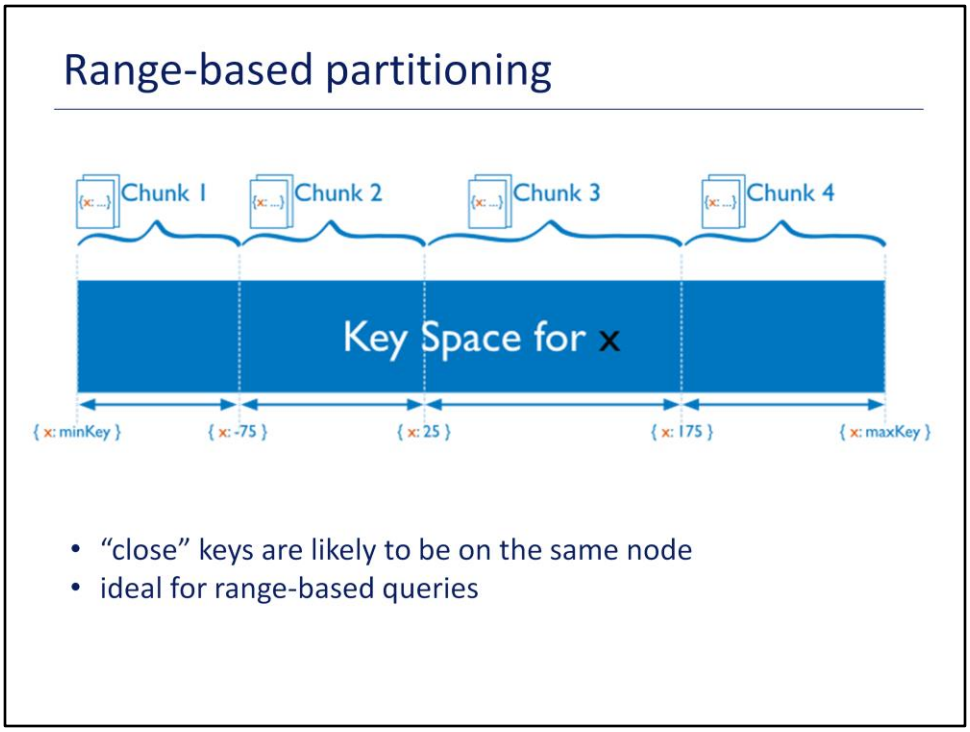
- Data shared by customer.id attribute
  - All related rows in two child tables are sharded as well
- Global Tables: common lookup tables, relatively low activity, replicated to all shards to avoid cross-shard joins

The example shows a simple schema for a Bookstore. The primary shard table that is used to shard the data is the 'customer' entity. The 'customer' table is the parent of the shard hierarchy, with the 'customer_order' and 'order_item' entities as child tables. The data is sharded by the 'customer.id' attribute, and all related rows in the child tables associated with a given 'customer.id' are sharded as well.

The global tables are the common lookup tables, which have relatively low activity, and these tables are replicated to all shards to avoid cross-shard joins. While this example is very basic, it does provide the basic considerations for determining how to shard a given database application.

Picking the right partition key

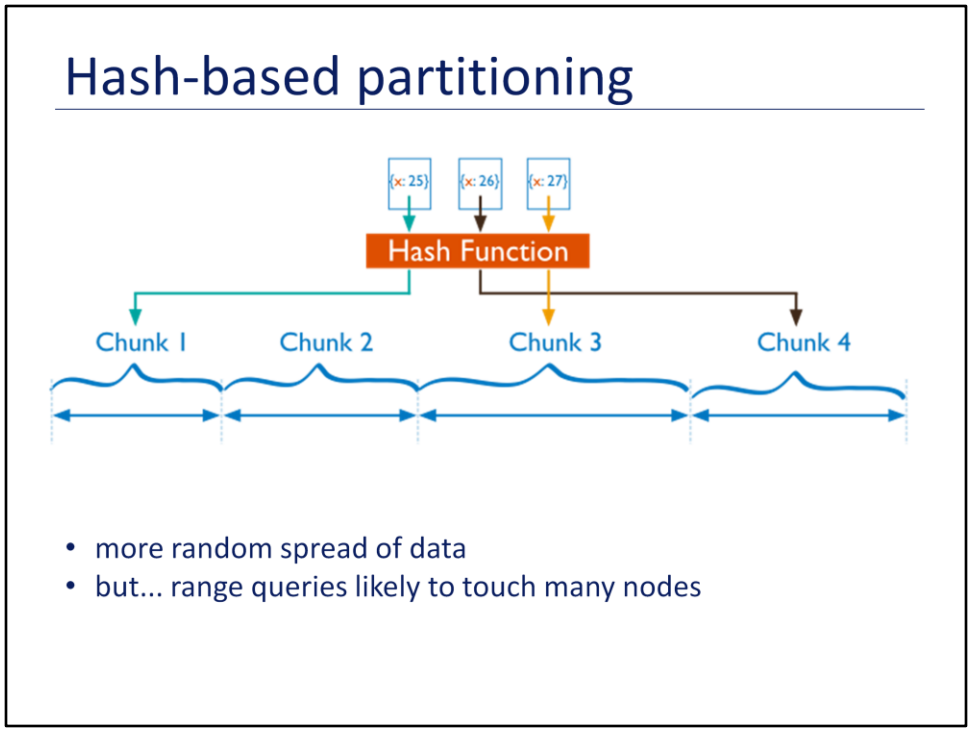
- Range-based partitioning
- Hash-based partitioning
  - Consistent hashing
- Combined keys
  - Case study: time-series in Cassandra



For *range-based sharding*, you divide the data set into ranges determined by the shard key values to provide **range based partitioning**. Consider a numeric shard key: If you visualize a number line that goes from negative infinity to positive infinity, each value of the shard key falls at some point on that line. The key space is partitioned into smaller, non-overlapping ranges. In MongoDB, these rangers are called **chunks** where a chunk is range of values from some minimum value to some maximum value.

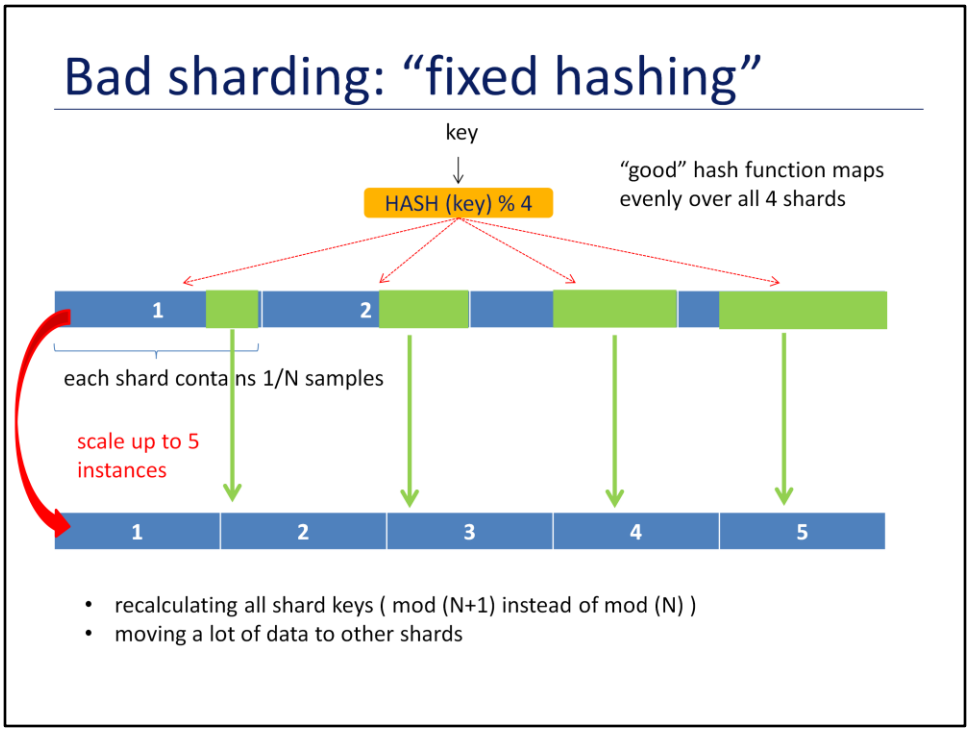
Given a range based partitioning system, data units with "close" shard key values are likely to be in the same chunk, and therefore on the same shard. Range based partitioning supports more efficient range queries. Given a range query on the shard key, the query router can easily determine which chunks overlap that range and route the query to only those shards that contain these chunks.

However, range based partitioning can result in an uneven distribution of data, which may negate some of the benefits of sharding. For example, if the shard key is a linearly increasing field, such as time, then all requests for a given time range will map to the same chunk, and thus the same shard. In this situation, a small set of shards may receive the majority of requests and the system would not scale very well.



For *hash based partitioning*, you compute a hash of a field's value, and then use these hashes to create chunks. With hash based partitioning, two documents with "close" shard key values are *unlikely* to be part of the same chunk.

Hash based partitioning ensures a more random (and thus even) distribution of data at the expense of efficient range queries. Hashed key values results in random distribution of data across chunks and therefore shards. But random distribution makes it more likely that a range query on the shard key will not be able to target a few shards but would more likely query every shard in order to return a result.

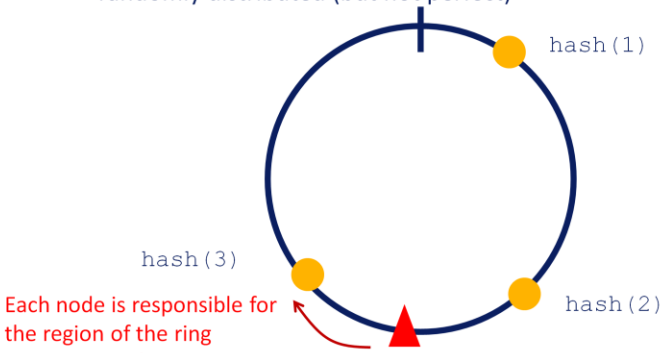


Sharding key schemes should also be robust to scaling up and/or node failures. Since good hashing functions generate output uniformly over their key space, modding this hash output over the number of nodes uniformly spreads the data units over all shards. Otherwise stated: the remainder of the division of the hash of the original key by the number of nodes is used to determine on which node the corresponding value is stored.

However, such a fixed hashing scheme performs badly when we scale horizontally, or when a node fails. Since the number of nodes has changed, we must recalculate for all data items the remainder of the division of the keys by the new number of nodes. For many data items, this means that they have to be shifted to another node in the cluster.

Remedy: Consistent hashing

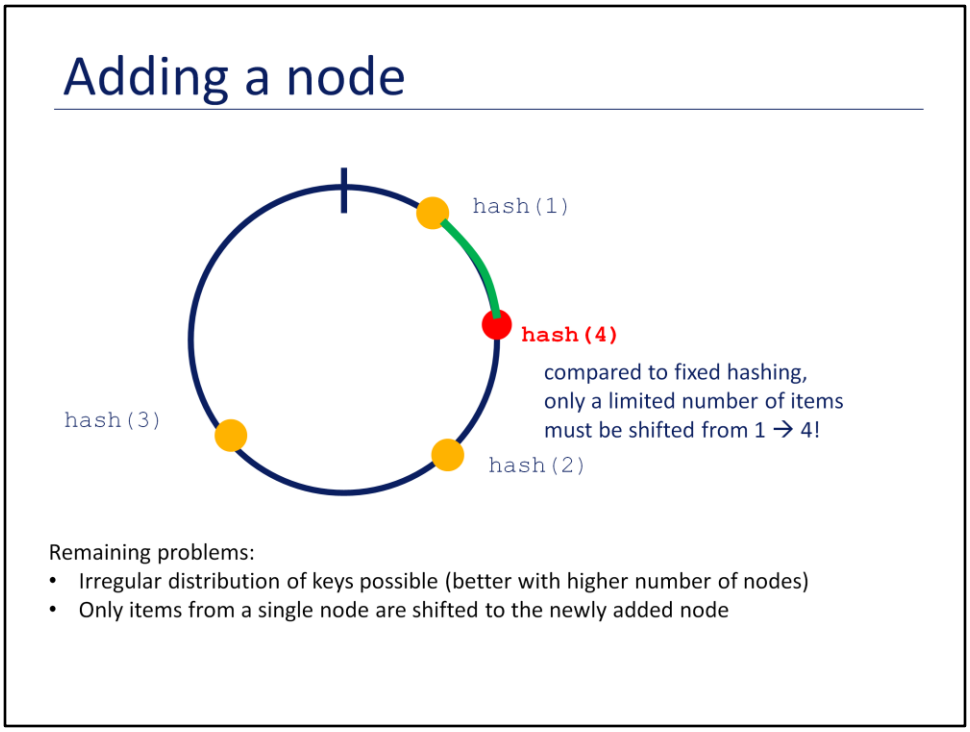
- Allow to determine the location of an object in spite of constant shifting of nodes in and out of the cluster
- predefined range of hash keys
  - take hash of machine node number
  - randomly distributed (but not perfect)



Consistent hashing provides a remedy for the issue presented on the previous slide.

The circle represents a predefined range of hash keys, organized in a ring. Keys are then hashed to produce a value that lies somewhere along the ring. If we take the hash of machine node numbers, than the resulting hashes will be more or less spread uniformly over the key space. We then state that each node is responsible for the region of the ring between itself and its predecessor. If we then calculate the hash of a data unit (the triangle on the slide), then we look at the first machine number hash that is strictly larger than the hash of the data unit to determine where to store the data unit.

(note: the order of hashes is not necessarily identical to the order of the elements hashed: hash(N+1) is not necessarily larger than hash(N))



If we add a fourth node, then only a limited number of items must be shifted compared to fixed hashing.

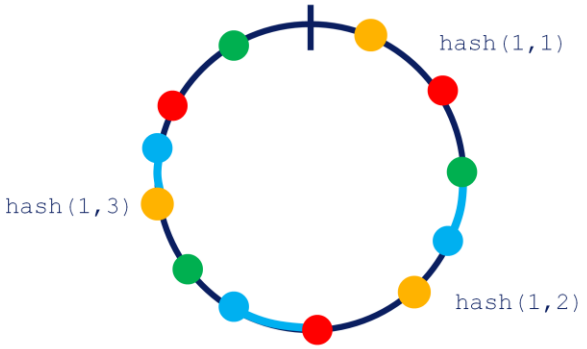
However, two problems remain:

- With a limited number of nodes, the distribution of the key range among the nodes might be irregular
- Only items of a single node are shifted to the newly added node, meaning that one node will be involved in an intensive process of copying data to the new node, while the others are left untouched

Adding a node

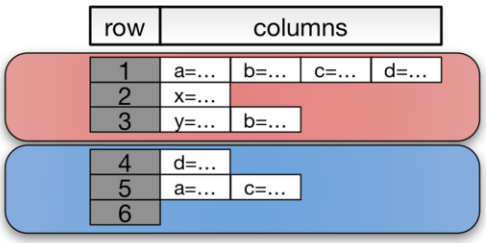
Map multiple "replicas" per machine

- More nodes on the circle = more uniform spreading
- Smaller number of nodes are redistributed to new machine, from almost each of the other machines in the cluster



A better solution is to map multiple replicas per machine. In the example above, each machine is mapped to three nodes on the "key circle". If we add an additional node (e.g. the red dots), then the circle will be further divided. As you can see, items will be shifted from almost each of the other machines.

Combined keys in Cassandra



- Model column families around query patterns
  - what queries do I want to support?
- De-normalize and duplicate for read performance
  - no JOINs in Cassandra
  - +/- one table (column family) per query pattern
  - repeating data is not a shame...

Picking the right partition (or shard) key is not an easy task. Most often, using a single field in your data will not be sufficient. Instead, you will have to resort to combined (or compound) keys.

We will study this for the Cassandra database that was discussed earlier. To make the most out of Cassandra, you need to follow two rules:

- Cassandra is optimized for high write throughput, so write operations are relatively cheap. Read operations tend to be more expensive and are much more difficult to tune. So, if you can perform extra writes to improve the efficiency of your read queries, it's almost always a good tradeoff. For this, you must carefully study which queries are typically performed on your data.
- Disk space is generally the cheapest resource (compared to CPU, memory, disk IO or network), and Cassandra is architected around that fact. In order to get the most efficient reads, you often need to de-normalize and duplicate data. In the relational world, the pros of normalization are well understood: less data duplication, fewer data modification anomalies, conceptually cleaner, easier to maintain, and so on. The cons are also understood: that queries might perform slowly if many tables are joined, etc. The same holds true in Cassandra, but the cons are magnified since it's distributed and Cassandra supports no joins.

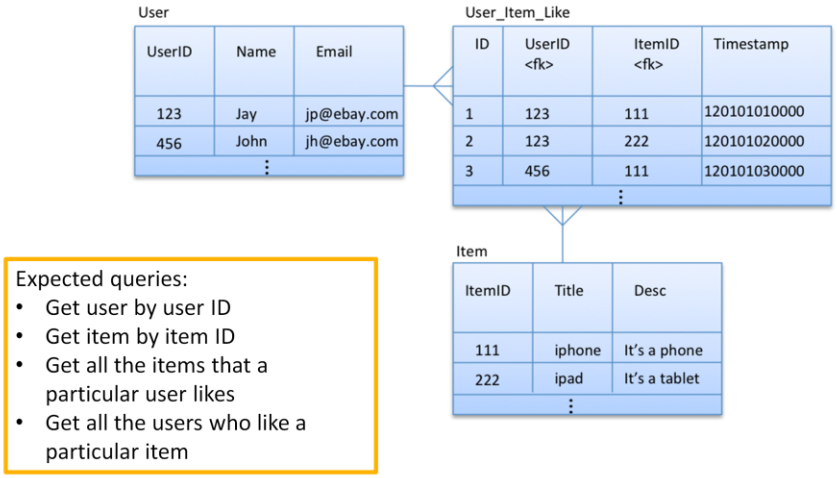
How to do this?

First, try to determine *exactly* what queries you need to support. This can include a lot of considerations that you may not think of at first. For example, think about grouping by an attribute, ordering by an attribute, filtering, enforcing uniqueness, etc... Changes to just one of these query requirements will frequently warrant a data model change for maximum efficiency.

Second, try to create a table (= column family) where you can satisfy your query by reading (roughly) one partition. In practice, this means you will use roughly one table per query pattern. If you need to support multiple query patterns, you usually need more than one table. To put this another way, each table should pre-build the "answer" to a high-level query that you need to support. If you need different types of answers, you usually need different tables. This is how you optimize for reads. Many of your tables may (/will) repeat the same data.

Example: “Like” relationship

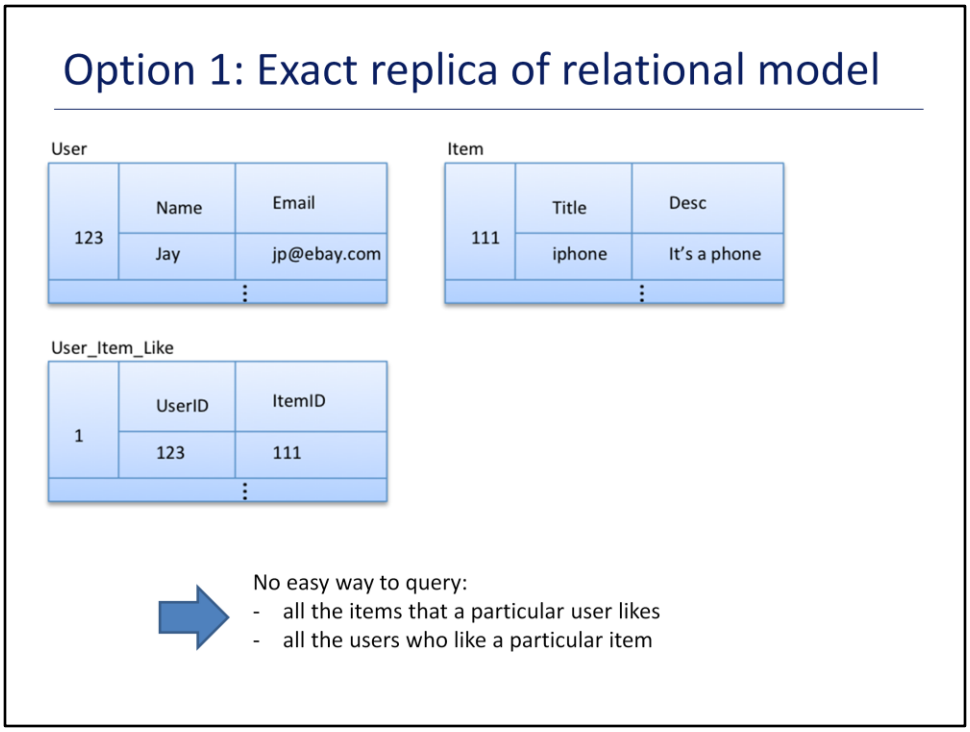
Relational view:



Expected queries:
- Get user by user ID
- Get item by item ID
- Get all the items that a particular user likes
- Get all the users who like a particular item

The example concerns the functionality of an e-commerce system where users can like one or more items. One user can like multiple items and one item can be liked by multiple users, leading to a many-to-many relationship as shown in the relational model on the slide.

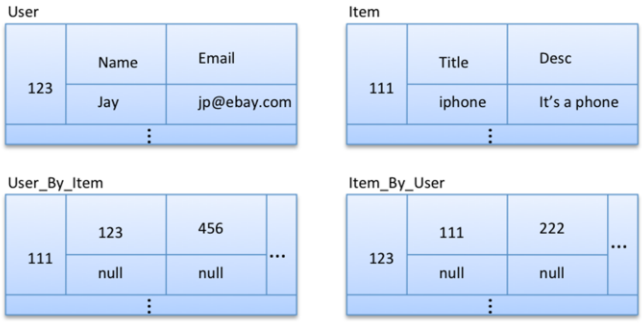
For this example, let’s say we would like to query data as follows:
- *Get user by user id*
- *Get item by item id*
- *Get all the items that a particular user likes*
- *Get all the users who like a particular item*



This model supports querying user data by user id and item data by item id. But there is no easy way to query all the items that a particular user likes or all the users who like a particular item. This is the worst way of modeling for this use case. Basically, User_Item_Like is not modeled correctly here.

Note that the 'timestamp' column (storing when the user liked the item) is dropped from User_Item_Like for simplicity.

Option 2: normalized entities with custom indexes



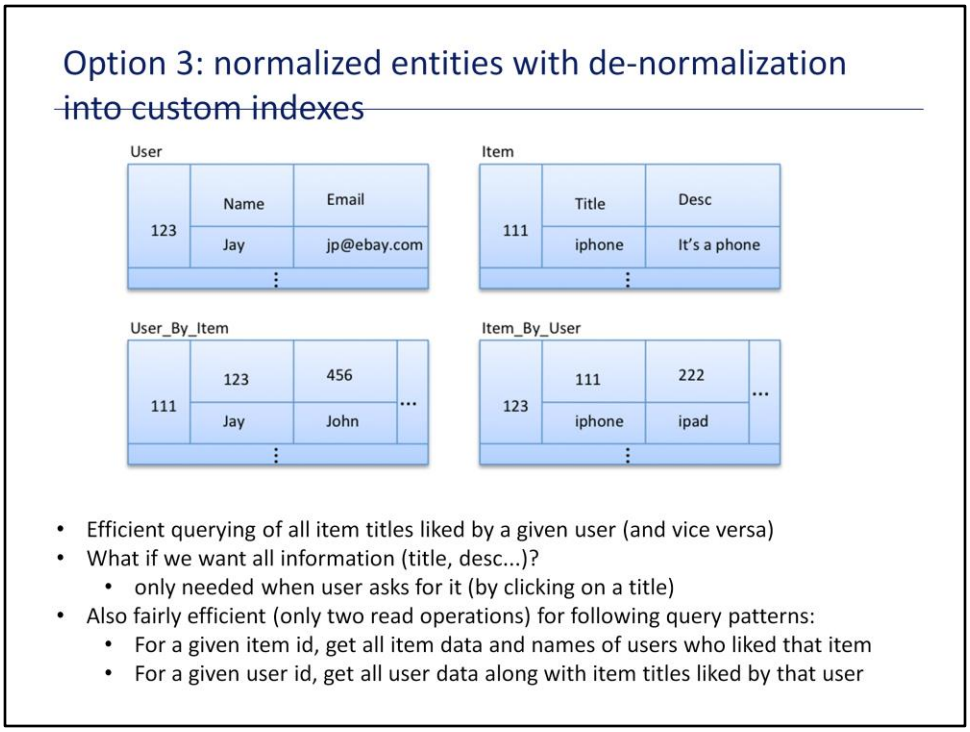
Problem

many additional queries when we look up usernames who like a given item and vice versa

This model has fairly normalized entities, except that user id and item id mapping is stored twice, first by item id and second by user id. Here, we can easily query all the items that a particular user likes using Item_By_User, and all the users who like a particular item using User_By_Item. We refer to these column families as custom secondary indexes, but they're just other column families.

Let's say we always want to get the item title in addition to the item id when we query items liked by a particular user. In the current model, we first need to query Item_By_User to get all the item ids that a given user likes; and then for each item id, we need to query Item to get the title.
Similarly, let's say we always want to get all the usernames in addition to user ids when we query users who like a particular item. With the current model, we first need to query User_By_Item to get the ids for all users who like a given item; and then for each user id, we need to query User to get the username.

It's possible that one item is liked by a couple hundred users, or an active user has liked many items — which will cause many additional queries when we look up usernames who like a given item and vice versa. So, it's better to optimize by de-normalizing item title in Item_by_User, and username in User_by_Item, as shown in option 3 on the next slide.



In this model, title and username are de-normalized in User_By_Item and Item_By_User respectively. This allows us to efficiently query all the item titles liked by a given user, and all the user names who like a given item. This is a fair amount of de-normalization for this use case.

What if we want to get all the information (title, desc, price, etc.) about the items liked by a given user? But we need to ask ourselves whether we really need this query, particularly for this use case. We can show all the item titles that a user likes and pull additional information only when the user asks for it (by clicking on a title). So, it's better not to do extreme de-normalization for this use case. (However, it's common to show both title and price up front. It's easy to do)

Let's consider the following two query patterns:

- For a given item id, get all of the item data (title, desc, etc.) along with the names of the users who liked that item.
- For a given user id, get all of the user data along with the item titles liked by that user.

These are reasonable queries for item detail and user detail pages in an application. Both will perform well with this model. Both will cause two lookups, one to query item data (or user data) and another to query user names (or item titles). As the user becomes more active (starts liking thousands of items?) or the item becomes hotter

(liked by a few million users?), the number of lookups will not grow; it will remain constant at two. That's not bad, and de-normalization may not yield as much benefit as we had when moving from option 2 to option 3.

Cassandra: clustering

```
CREATE TABLE playlists (
id uuid,
song_order int,
song_id uuid,
title text,
album text,
artist text,
PRIMARY KEY (id, song_order )
);
```

Compound primary key consists of:

- Partition key → determines which node stores which row
- Additional columns → determine clustering, how is data sorted on disk

A compound primary key consists of the partition key and one or more additional columns that determine clustering. The partition key determines which node stores the data. It is responsible for data distribution across the nodes. The additional columns determine per-partition clustering. Clustering is a storage engine process that sorts data within the partition.

The data for each partition is clustered by the remaining column or columns of the primary key definition. On a physical node, when rows for a partition key are stored in order based on the clustering columns, retrieval of rows is very efficient.

Time series data modelling

Example: weather station creating temperature reading every minute

Option 1: add column every minute

Figure 1

```
CREATE TABLE temperature (
weatherstation_id text,
event_time timestamp,
temperature text,
PRIMARY KEY (weatherstation_id,event_time) );
```

1234ABCD	2015-04-03 07:01:00	2015-04-03 07:02:00	2015-04-03 07:03:00	2015-04-03 07:04:00	2015-04-03 07:05:00
	11	11	12	12	12

Cassandra's data model is an excellent fit for handling data in sequence regardless of datatype or size. When writing data to Cassandra, data is sorted and written sequentially to disk. When retrieving data by row key and then by range, you get a fast and efficient access pattern due to minimal disk seeks – time series data is an excellent fit for this type of pattern.

The simplest model for storing time series data is creating a wide row of data for each source. In this first example, we will use the weather station ID as the row key. The timestamp of the reading will be the column name and the temperature the column value. Since each column is dynamic, our row will grow as needed to accommodate the data. The event_time is used to sort the data on disk.

Time series data modelling

```
SELECT ( weatherstation_id, event_time, temperature
FROM temperature
WHERE wheaterstation_id='1234ABCD'
AND event_time >= '2015-04-03 07:01:00'
AND event_time <= '2015-04-03 07:05:00'
```

sequential read on disk!

1234ABCD	2015-04-03 07:01:00	2015-04-03 07:02:00	2015-04-03 07:03:00	2015-04-03 07:04:00	2015-04-03 07:05:00
	11	11	12	12	12

If we then query the temperature measured for a specific weatherstation in a specific period, then the query can be executed very efficiently:

- The node with the row containing the requested data is easily found by the partition key (all data of a single weather station is stored on a single node)
- We then only perform a sequential read on the disk, since Cassandra orders the columns by the timestamp value

Compound partition key

In the previous lay-out, you would ultimately hit the 2 billion column limit

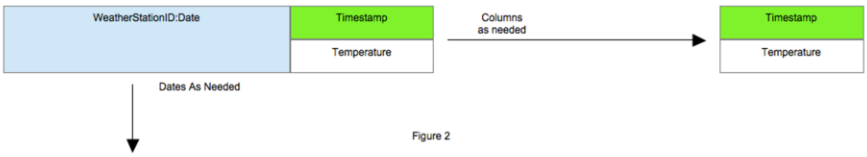


Figure 2

```
CREATE TABLE temperature_by_day (
  weatherstation_id text,
  date text,
  event_time timestamp,
  temperature text,
PRIMARY KEY ((weatherstation_id,date),event_time) );
```

Compound partition key will group all weather data for a single day on a single row (and thus node)

In some cases, the amount of data gathered for a single device isn't practical to fit onto a single row. Cassandra can store up to 2 billion columns per row, but if we're storing data every millisecond you wouldn't even get a month's worth of data. The solution is to use a pattern called row partitioning by adding data to the row key to limit the amount of columns you get per device. Using data already available in the event, we can use the date portion of the timestamp and add that to the weather station id. This will give us a row per day, per weather station, and an easy way to find the data.

Note the (*weatherstation_id,date*) portion. When we do that in the PRMARY KEY definition, the key will be compounded with the two elements. Now when we insert data, the key will group all weather data for a single day on a single row.

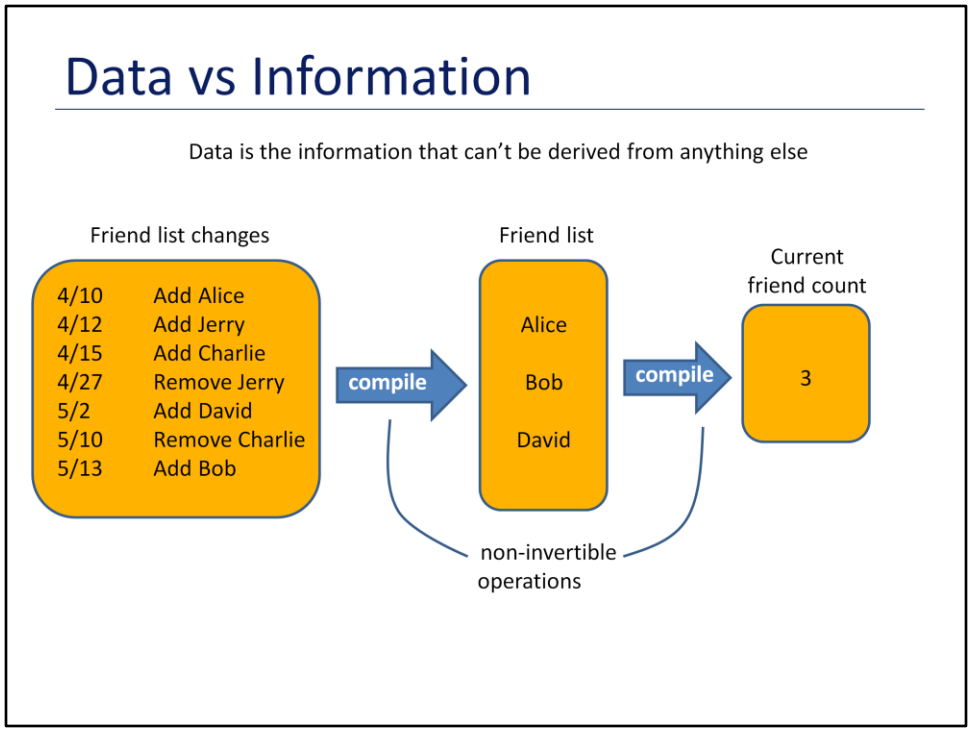
This comes however at the cost of having to perform reads from (possibly) two partitions if you want to read the data of multiple days of the same weather station.

Further reading

- D. McCreary, Making Sense of NoSQL: A guide for managers and the rest of us
- A. Fowler, NoSQL for Dummies
- R. Strickland, Cassandra High Availability

THE PROPERTIES OF DATA

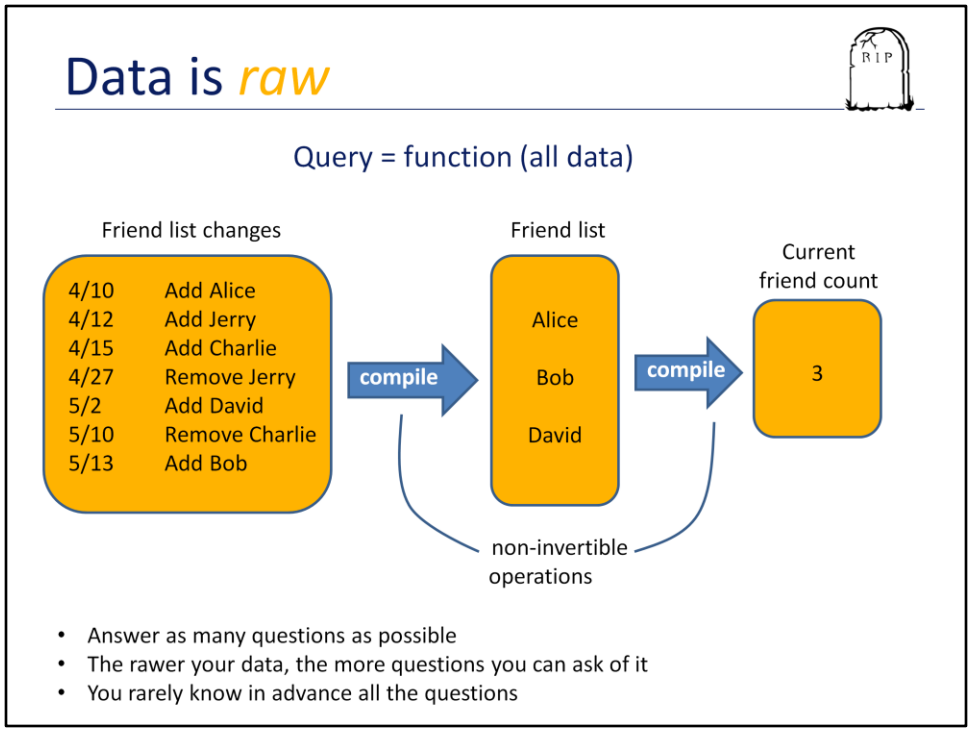
Raw
Immutable
Perpetual



The example illustrates information dependency. Each layer of information can be derived from the one to its left, but it's a one-way process. From the sequence of friend and unfriend events, you can determine the other quantities. But if you only have the number of friends, it is impossible to determine exactly who they are.

This shapes the following definitions:

- *Information* is the general collection of knowledge relevant to your big data system. It's synonymous with the colloquial usage of the word *data*.
- *Data* refers to the information that can't be derived from anything else. Data serves as the axioms from which everything else derives.
- *Queries* are questions you ask of your data. For example, you query your financial transaction history to determine your current bank account balance.
- *Views* are information that has been derived from your base data. They are built to assist with answering specific types of queries.

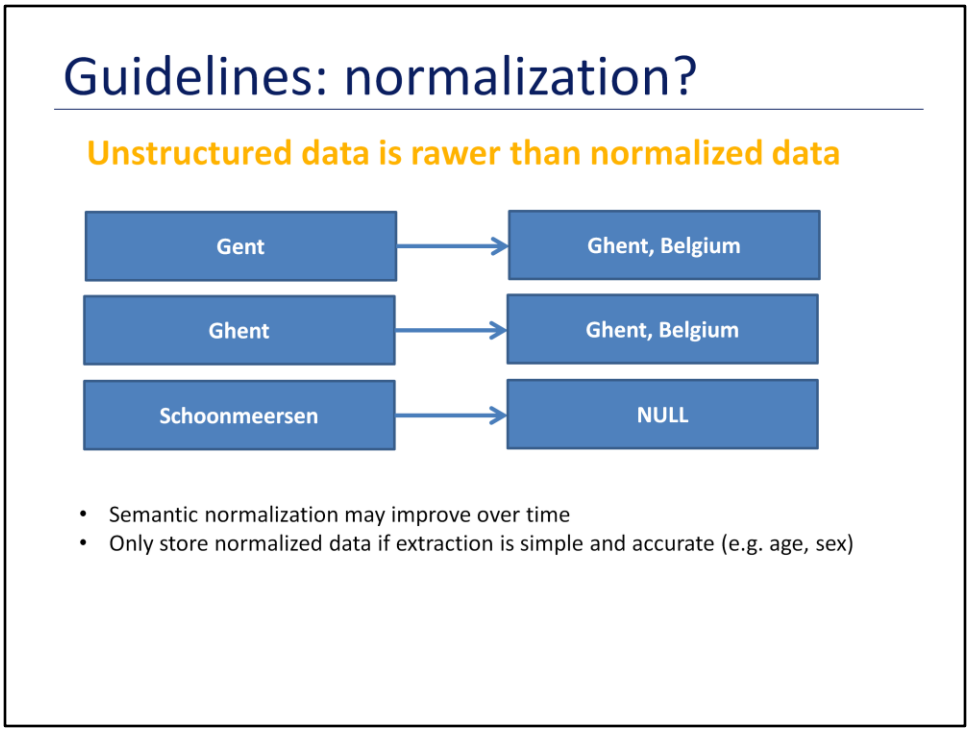


A big data system must be able to answer as many questions as possible. We'll colloquially call the property *rawness.* The rawer your data, the more questions you can ask of it. If you can, you want to store the rawest data you can get your hands on.

The example shows the data you might keep when designing a new social network. Each layer of information can be derived from the one to its left, but it's a one-way process. From the sequence of friend and unfriend events, you can determine the other quantities. But if you only have the number of friends, it is impossible to determine exactly who they are.
This shapes the following definitions:

- *Information* is the general collection of knowledge relevant to your big data system. It's synonymous with the colloquial usage of the word *data*.
- *Data* refers to the information that can't be derived from anything else. Data serves as the axioms from which everything else derives.

Storing raw data is hugely valuable because you rarely know in advance all the questions you want answered. By keeping the rawest data possible, you maximize your ability to obtain new insights, whereas summarizing, overwriting, or deleting information limits what your data can tell you. The trade-off is that rawer data typically entails more of it—sometimes much more. But Big Data technologies are designed to manage exabytes of data.



Although the concept of *rawness* is straightforward, it is not always clear what information you should store as your raw data. A common hazy area area is the line between *parsing* and *semantic normailization*. Semantic normalization is the process of reshaping free-form information into a structured form of data or pre-defined vocabulary (e.g. a dictionary of medical terms).

In our social network example, users may input anything for their location. A semantic normalization would try to match the input with a known place. We argue that it is better to store the unstructured string, because your semantic normalization algorithm may improve over time. If you store the unstructured string, you can renormalize that data at a later time when you have improved your algorithms. In the example, you may adapt the algorithm to recognize Schoonmeersen as a campus of Ghent University in Ghent.

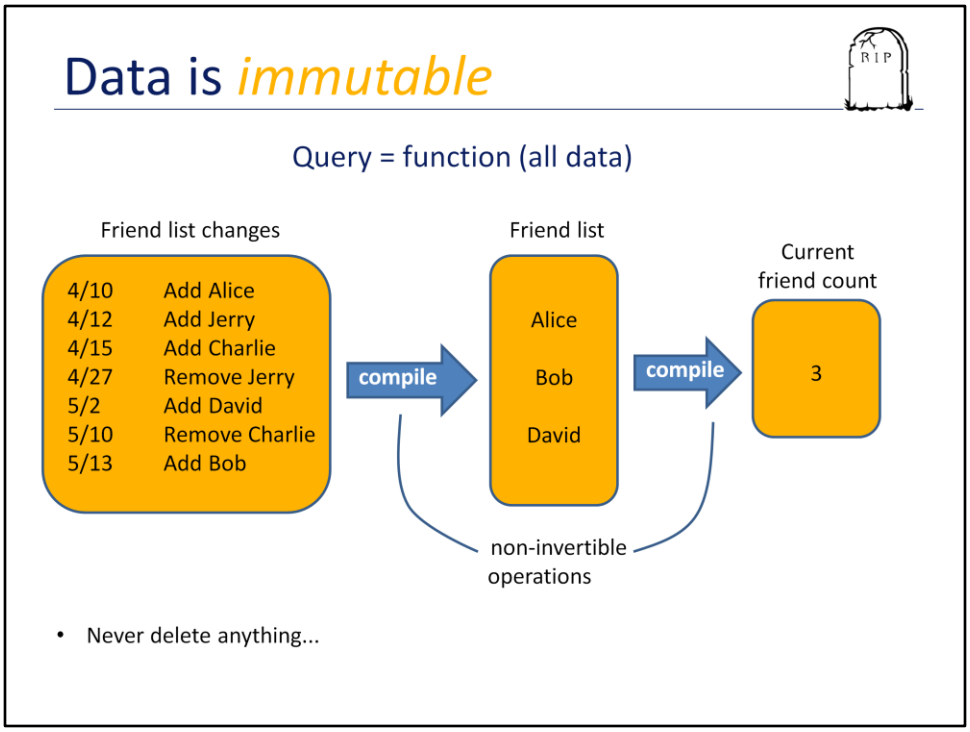
As a rule of thumb, you should store the results of an algorithm for data extraction when that algorithm is simple and accurate, like extracting an age from an HTML page. If the algorithm is subject to change, due to improvements or broadening the requirements, store the unstructured form of the data.

Guidelines: more data = rawer data?

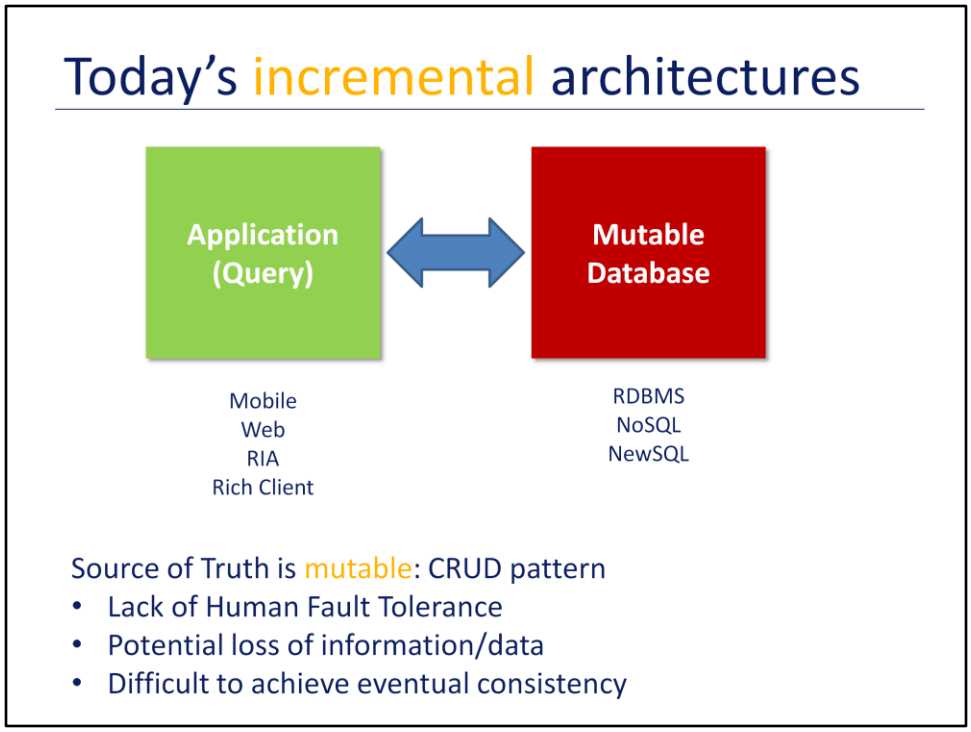
Bigger is not always better

- More data does not always equate to rawer data
- Example: URL of blog in social media profile
  - some HTML tags provide useful information on content
  - Javascript, CSS do not

It's easy to presume that more data equates to rawer data, but that's not always the case. Suppose that a social media user posts the URL of his new blog post on his profile. What exactly should you store? Storing the pure text of the blog entries is certainly a possibility. But any phrases in italics, boldface, or titles were deliberately emphasized by the user and could prove useful in text analysis. For example, you could use this additional information for an index to make your social network searchable. We'd thus argue that the annotated text entries (HTML tags) are a rawer form of data than the simple ASCII text strings. On the other hand, storing the color scheme, stylesheets and Javascript code of the blog website can't be used to derive any further information about the user. They serve only as the container for the contents of the site and shouln't be part of the raw data. You will strip this information of the HTML page before storing it as raw data.



You are unlikely to modify a data item already written to the storage. Instead; you will always append data.



What characterizes traditional application architectures is the use of read/write databases and maintaining the state in those databases *incrementally* as new data is seen. Applications continuously update items in the database. For example, an incremental approach to counting pageviews would be to process a new pageview by adding one to the counter for its URL. The problems with incremental architectures are significant: intolerance for human errors and the operational complexity as well as the contraints of the CAP theorem may lead to loss of information/data.

Note that this characterization of incremental architectures is a lot more fundamental than the discussion of relation vs. non-relational databases – the vast majority of both relation and non-relation databases deployments are done as fully incremental architectures.

Lack of human fault tolerance

- Bugs will be deployed to production over the lifetime of a data system
- Operational mistakes will be made
- Humans are part of the overall system
  - Just like hard disks, CPUs, memory, software
  - Design for human error like you design for any other fault
- Examples of human error:
  - Deploy a bug that increments counters by two
  - Accidentally delete data from database
  - Accidental DOS on important internal service

As long as an error does not lose or corrupt good data, you can fix what went wrong.

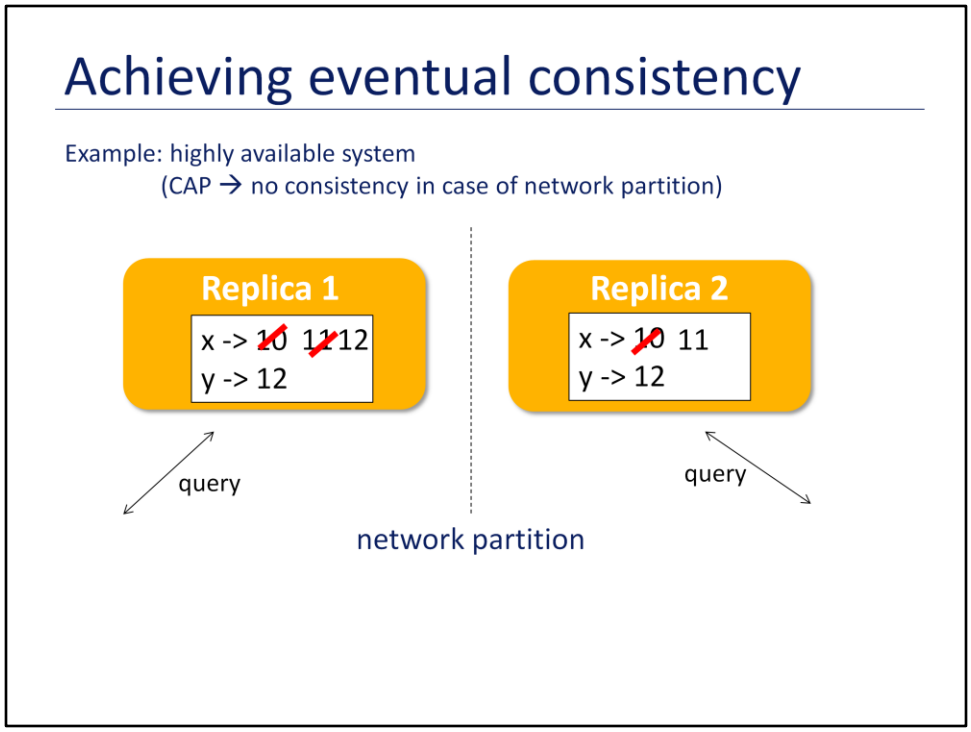
Operational complexity

- Parts on the disk become unused as items are modified or deleted
- Compaction is the process of reclaiming space
- Compaction is expensive
  - lowers performance of machine while running
  - can even cause cascading failures

One of the difficulties in operating a production infrastructure hosting an incremental architecture is the need for read/write databases to perform online compaction. In a read/write database, as a disk index is incrementally added to and modified, parts of the index become unused. These unused parts take up space and eventually need to be reclaimed to prevent the disk from filling up. Reclaiming space as soon as it becomes unused is too expensive, so the space is occasionally reclaimed in bulk in a process called *compaction*.

Compaction is an intensive operation. The server places substantially higher demand on the CPU and disk during compaction, which dramatically lowers the performance of that machine during that period. The performance loss can even cause cascading failure – if too many machines compact at the same time, the load they were supporting will have to be handled by other machines in the cluster. This can potentially overload the rest of your cluster, causing total failure.

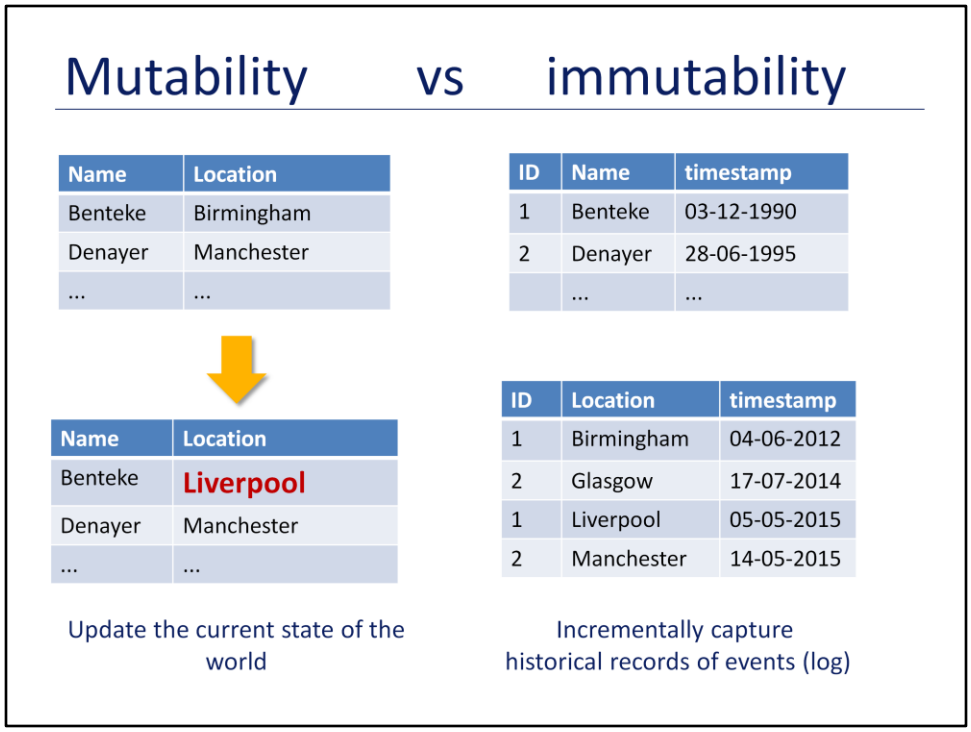
A competent operational staff can correctly schedule compactions on each node so that not too many nodes are afffected at one. But this requires knowledge on how long a compaction takes (as well as the variance) and enough disk capacity on the nodes to last between compactions. A better approach is to avoid online compaction as much as possible.



Another complexity of incremental architectures emerges when trying to make systems highly available. A highly available system allows for queries and updates even in the presence of machine or partial network failure. As stated by the CAP theorem, it is impossible to realize consistency in a highly available system when a network partition occurs. So a highly available system sometimes returns stale results during a network partition.

In order for a highly available system to return to consistency once a network partition ends (*eventual consistency*), a lot of help is required from your application. Take for example the basic use case of maintaining a count in a database. The obvious way to go is to increment a number in the database whenever an event is received that requires the count to go up. To achieve high availability, distributed databases will keep multiple replicas. The information remains available even if a machine goes down or the network gets partitioned. During a network partition, a highly available system has clients update whatever replicas are reachable to them. This causes replicas to diverge and received different sets of updates. Only when the partition goes away can the replicas be merged together into a common value. In the example above, the network partition is resolved when x has the value of 12 in replica 1 and a value of 11 in replica 2. What should the merged value be? Although the correct answer is 13 (since x had the value 10 when the network partition started), there is no way to know just by looking at the numbers 12 and 11. The replicas could have diverged at 11 (in this case the answer would be 12), or at 0 (in this case the answer

would be 23.
To do highly available counting correctly, you need a data structure that is amenable to merging when values diverge and need to implement repairing code. That is an amazing amount of complexity just to maintain a simple count.



In traditional database, you update existing records. In the example, we keep track of the location of football players. When a player updates his location because he is transferred to another team, in a mutable system we will update the location field in the table. The mutable system thus stores a current snapshot of the world.

In a immutable system, you create a separate record every time the data changes. Accomplishing this requires two changes. First, you track each field in a separate table. Second, you tie each unit of data to a moment in time when the information is known to be true.

By keeping each field in a separate table, you only record the information that changed. This requires less space for storage and guarantees that each record is new information and is not simply carried over from the last record.

Advantages of data immutability

- Human-fault tolerance
  - No data can be lost
  - Delete bad data units and recompute if necessary
- Simplicity
  - No data index needed
  - Only need to append new data units
- Trade-offs with data storage

Using an immutable schema for big data systems means that there are no updates or deletes of data. Instead, you only add more data, which gains you two vital advantages:

- **Human-fault tolerance** – People will make mistakes, and you must limit the impact of such mistakes and have mechanisms for recovering from them. With a mutable data model, a mistake can cause data to be lost, because values are actually overridden in the database. With an immutable data model, *no data can be lost*. If bad data is written, earlier (good) data units still exist. Fixing the data system is just a matter of deleting the bad data units and recomputing the views built from the master dataset.
- **Simplicity –** Mutable data models imply that the data must be indexed in some way so that specific data objects can be retrieved and updated. In contrast, with an immutable data model you only need the ability to append new data units to the master dataset. This doesn't require an index for your data, which is a huge simplification. Storing a master dataset can be as simple as storing a flat file.

One of the trade-offs of the immutable approach is that it uses more storage than a mutable schema. First, the user ID is specified for every property, rather than just once per row, as with a mutable approach. Additionally, the entire history of events is stored rather than just the current view of the world. You should take advantage of the ability to store large amounts of data using Big Data technologies to get the benefits of immutability. The importance of having a simple and strongly human-fault tolerant master dataset can't be overstated.

Data is perpetual

- Each piece of data is true in perpetuity
  - Once true, always true
- Tagging each piece of data with a timestamp is a practical way to achieve this

Deleting data is a statement about the *value* and not about thruthfulness

- Garbage collection
- Regulations

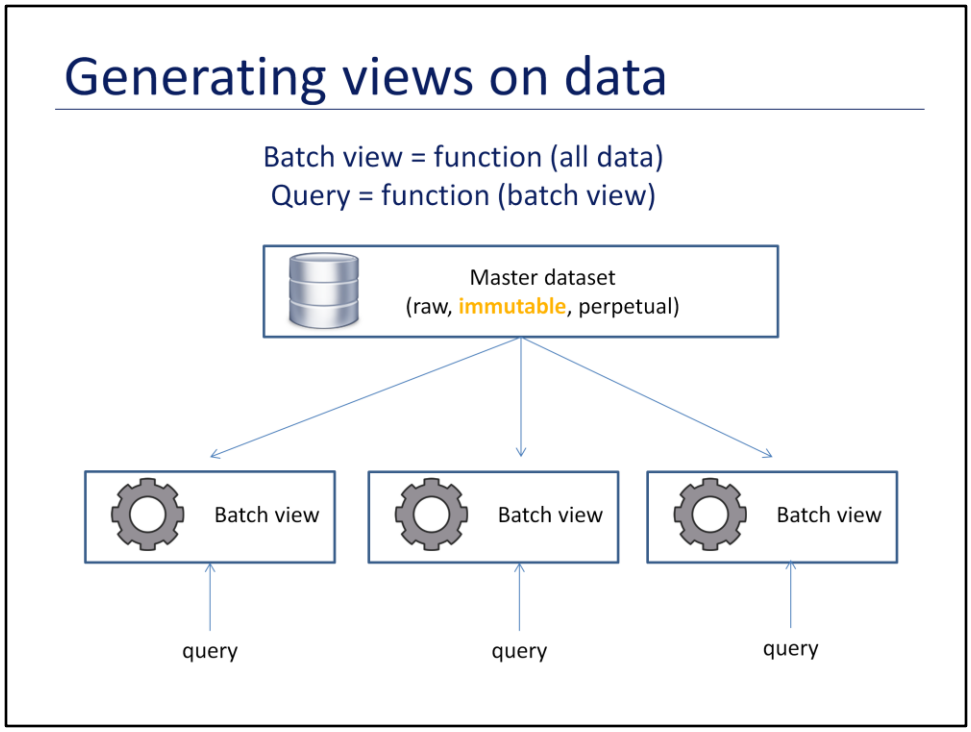
The key consequence of data immutability is that each piece of data is true in perpetuity. A piece of data, once true, must always be true; Immutability wouldn't make sense without this property, and tagging each piece of data with a timestamp is a practical way to make data eternally true.

In general, the master dataset consistently grows by adding new immutable and eternally true pieces of data. There are some special cases in which you do delete data, but these cases are not incompatible with data being eternally true.

- Garbage collection - you delete all data units that have low value. You can use garbage collection to implement data retention policies that control the growth of the dataset. For example, you may decide to keep only one location per person per year instead of the full history of each time a user changes locations.
- Regulations – government regulations may require you to purge data from your databases under certain conditions

In both cases, deleting the data is not a statement about the truthfulness of the data. Instead, it is a statement about the value of the data. Although the data is eternally true, you may prefer to forget the information either because you must or because it doesn't provide enough value for the storage cost.

BATCH PROCESSING

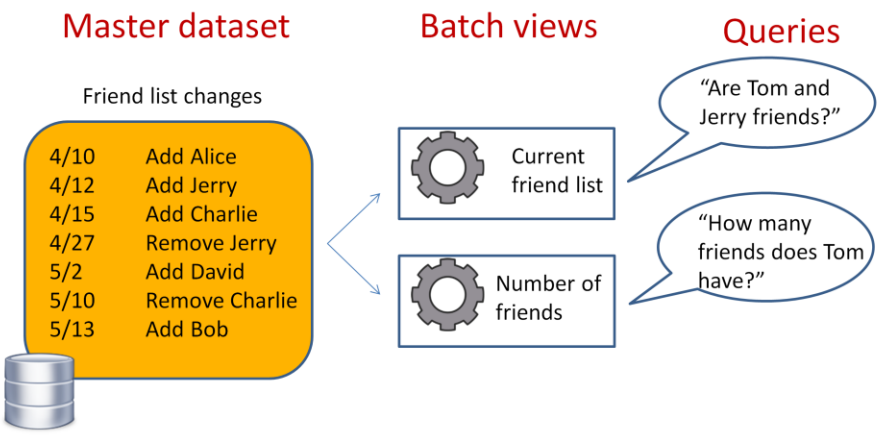


Because of the many problems with today's incremental architectures, we will study another approach to cope with data. The foundation of the approach lies in the construction of the master dataset. This master dataset is structured along the RIP principles discussed in the previous section: it contains raw data that is no longer touched and kept forever.

Views are information that has been derived from the master dataset. They are pre-computed to assist with answering specific types of queries. The precomputed view is indexed so that it can be accessed with random reads. In this system, you run a function on all the data to get the view. Then, when you want to know the value for a query, you run a function on that view. In the master dataset, no data is stored redundantly. For efficiency reasons, the result of the batch views may contain duplicate data: one piece of data from the master dataset may get indexed into many batch views.

Of course, the master dataset is continuously expanding with new data. By the time the result of the view calculation is available, it is very likely that the result is already out-of-date. We will discuss how to cope with this later in this course. For now, we focus on *batch* processing: processing a snapshot of the master dataset.

Example



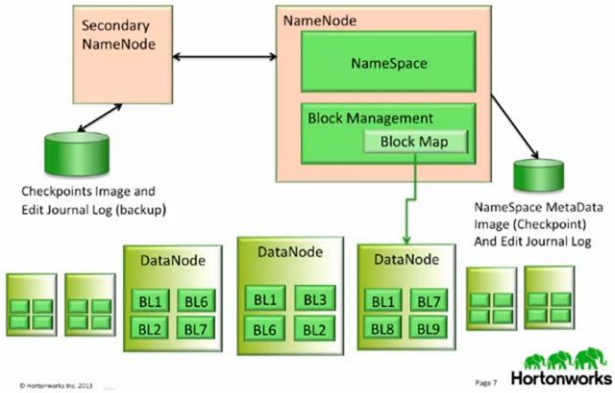
- Batch views are indexed for fast querying
- Duplicate information between different batch views possible

MapReduce

- Execution framework and programming model
- large-scale data processing
- builds on a distributed file system
- Apache Hadoop MapReduce is the most widely known and widely used open source implementation of the Google MapReduce paradigm
- Unlike traditional HPC clusters, Hadoop uses the same set of compute nodes for data storage as well as to perform the computations

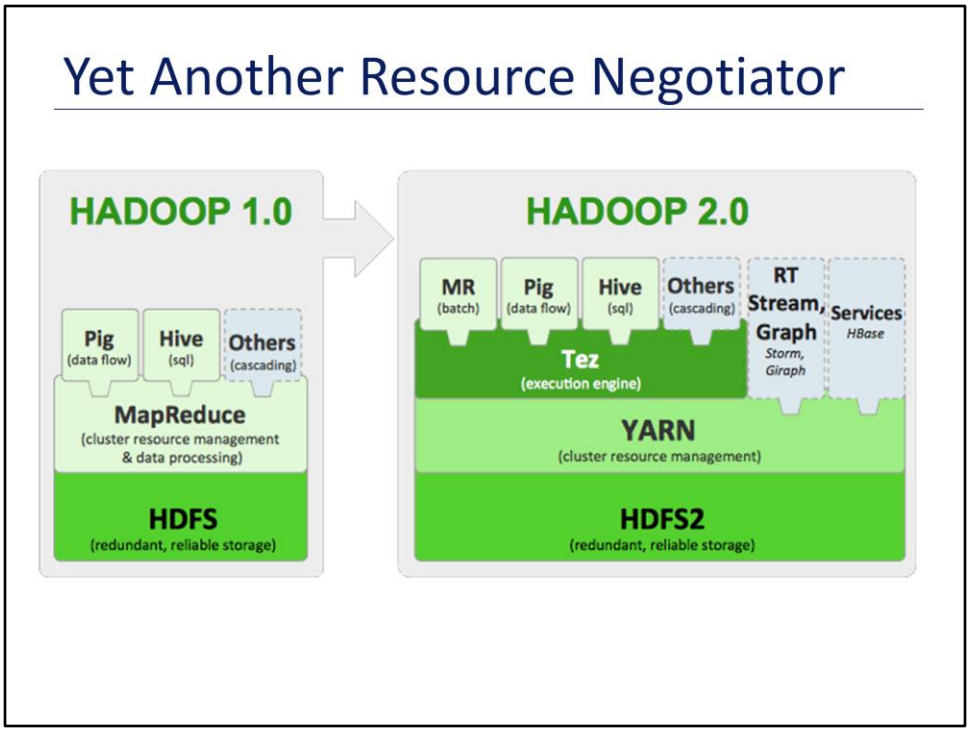
Hadoop Distributed File System recap

- High throughput parallel file system
- Massive amounts of data stored on commodity computers



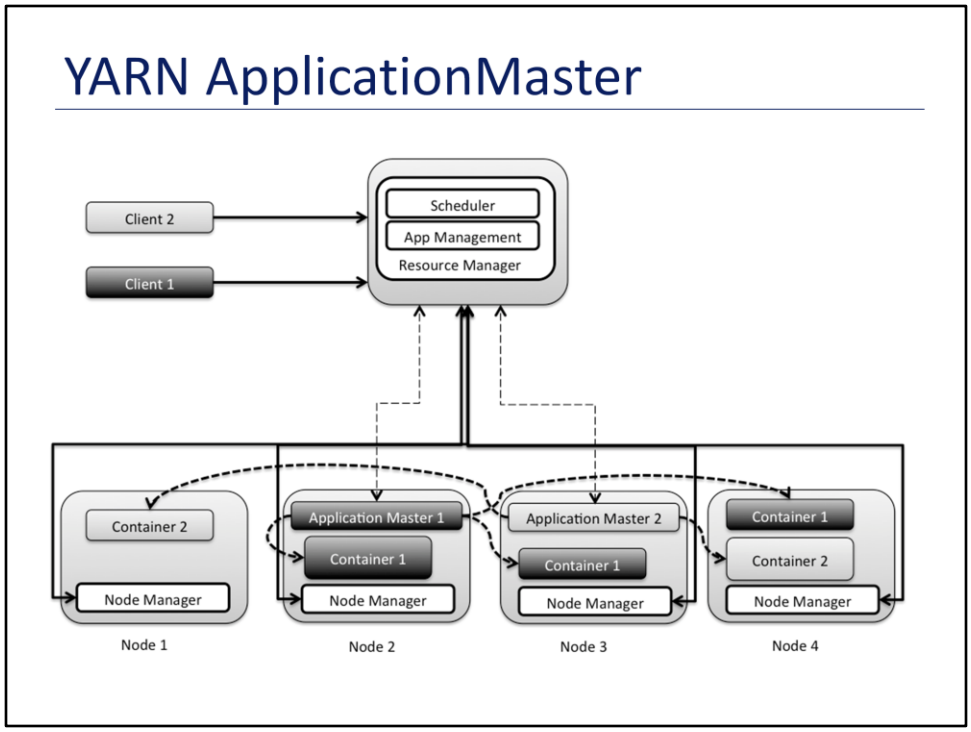
The lowest layer of the MapReduce framework is the Hadoop Distributed File System. Since we discussed this earlier, we briefly recapitulate the concepts most relevants for batch processing.

HDFS consists of **NameNode** and **DataNode** services providing the basis for the distributed filesystem. NameNode stores, manages, and serves the metadata of the filesystem. NameNode does not store any real data blocks. DataNode is a per node service that manages the actual data block storage in the DataNodes. When retrieving data, client applications first contact the NameNode to get the list of locations the requested data resides in and then contact the DataNodes directly to retrieve the actual data.



YARN (**Yet Another Resource Negotiator**) is a resource management system that allows multiple distributed processing frameworks to effectively share the compute resources of a Hadoop cluster and to utilize the data stored in HDFS. The primary goal of YARN is to separate concerns relating to resource management and application execution. By separating these functions, it provides a common platform for many different types of distributed applications. Users can thus utilize multiple distributed application frameworks side by side sharing a single cluster and the HDFS filesystem.

The batch processing based MapReduce framework was the only natively supported data processing framework in Hadoop v1. While MapReduce works well for analyzing large amounts of data, MapReduce by itself is not sufficient enough to support the growing number of other distributed processing use cases such as real-time data computations, graph computations, iterative computations, and real-time data queries.



This slide lays out the architecture of a YARN-based cluster. YARN has a concept called containers, which is the unit of resource allocation. Each allocated container has the rights to a certain amount of CPU and memory in a particular compute node. Applications can request resources from YARN by specifying the required number of containers and the CPU and memory required by each container.

YARN abstracts out resource management functions to a platform layer called **ResourceManager (RM).** The RM is a per-cluster daemon that solely manages and allocates resources to the different applications (also known as jobs) submitted to the cluster. It has two main components: the Scheduler and the ApplicationsManager.

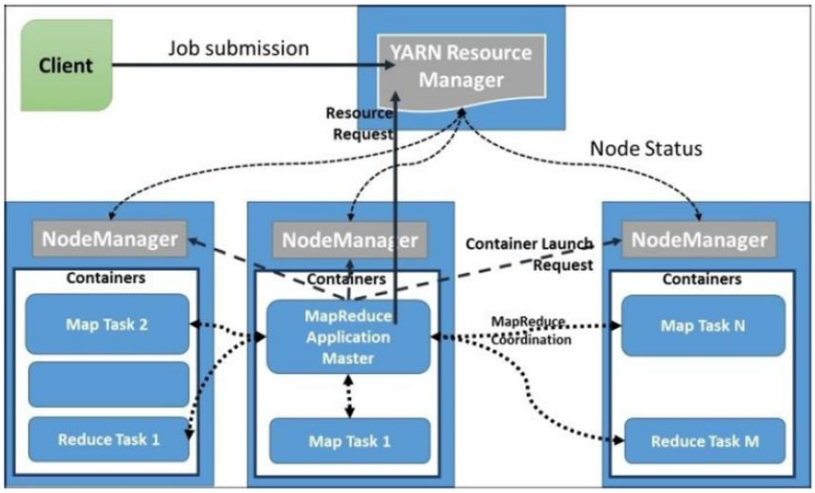
The Scheduler is responsible for allocating resources to the various applications that are running in the cluster. It does not have any insight into the status of the application: it does not guarantee restarts on application or hardware failures. It uses queues and capacity parameters during the allocation process. The ApplicationsManager is the component responsible for handling application submissions made by clients. It also bootstraps applications by negotiationg the container on behalf of the application for the Application Master. The ApplicationsManager also provides the services of restarting the Application Master in case of failures.

The **NodeManager (NM)** is a per-node daemon that does local container

management, ranging from authentication to resource monitoring (CPU, memory, disk health). The NM reports monitoring parameters to the RM. The RM scheduler can take decisions on container scheduling based on the load or health of the node.

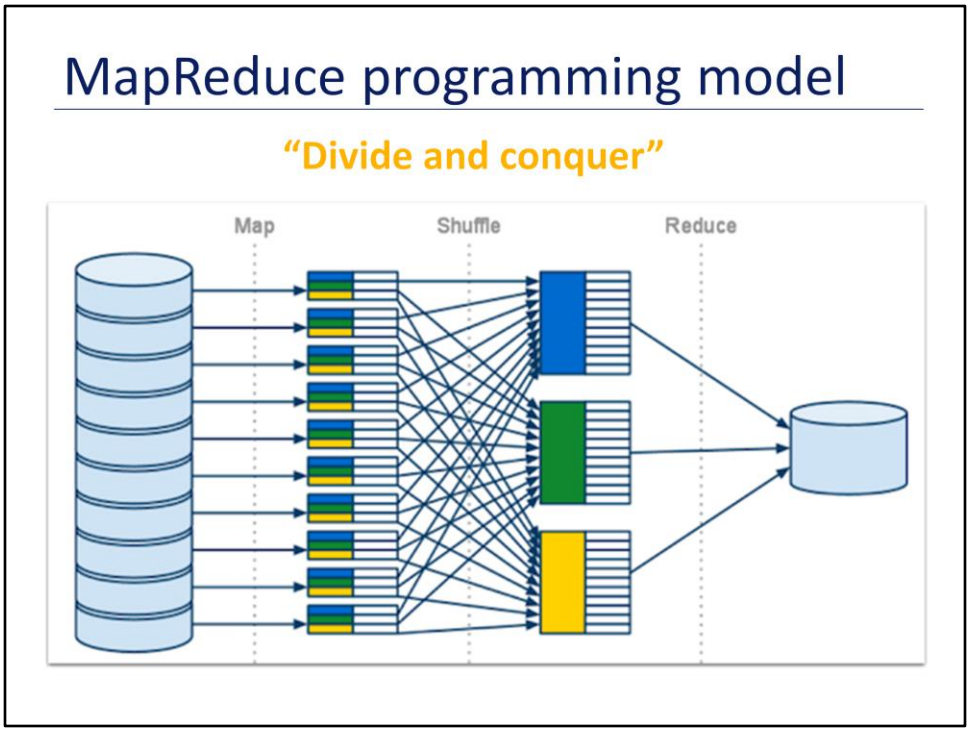
The **ApplicationMaster (AM)** is a per-application process that coordinates the computations for a single application. The first step of executing a YARN application is to deploy the AM. After an application is submitted by a YARN client, the RM allocates a container and deploys the AM for that application. Once deployed, the AM is responsible for requesting and negotiating the necessary resource containers from the RM. Once the resources are allocated by the RM, AM coordinates with the NM to launch and monitor the application containers in the allocated resources.
Having separate AMs for each submitted application improves the scalability of the cluster as opposed to having a single process bottleneck to coordinate all the application instances.

YARN ApplicationMaster



The diagram depicts the interactions between various YARN components, when a MapReduce application is submitted to the cluster. When the MapReduce ApplicationManager requests the Resource Manager for containers, it assigns a Map task to the container whose data is local or close to the allocated container node. The decision of which Map task is executed in the container thus happens *after* the AM receives the containers.

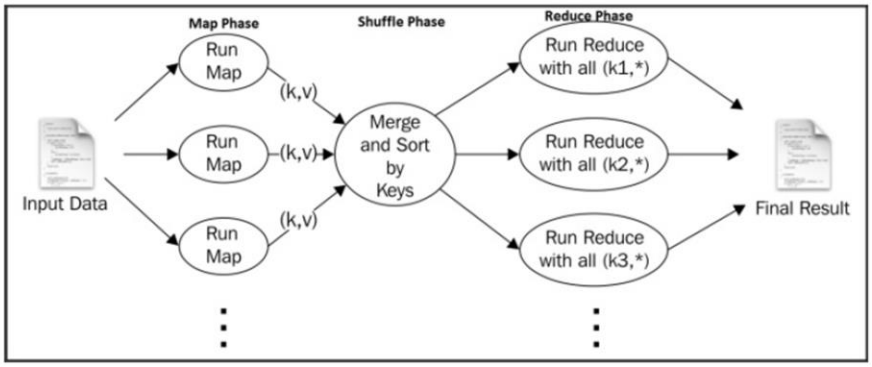
This concept is called late binding: the container spawned might not directly be related to the AM's request. The state at which the AM requests resources might change by the time the resource is allocated.



After having discussed the execution framework of MapReduce, we now turn our attention to the programming model.

The MapReduce programming model adopts a divide and conquer strategy to handle large data sets. First, the dataset is partitioned into smaller, independent data chunks to be processed in parallel ("map"). Then, the results from the previous step are combined, merged or otherwise aggregated ("reduce").

MapReduce programming model



User specifies two functions:

- map: (k1, v1) → list [k2, v2]
- reduce: (k2, list [v2]) → list [k3,v 3]

Sorting of intermediate keys between map and reduce phase.

The MapReduce programming model consists of Map and Reduce functions. The Map function receives each record of the input data (lines of a file, rows of a database, and so on) as key-value pairs and outputs key-value pairs as the result. By design, each Map function invocation is independent of each other allowing the framework to use divide and conquer to execute the computation in parallel. This also allows duplicate executions or re-executions of the Map tasks in case of failures or load imbalances without affecting the results of the computation. Typically, Hadoop creates a single Map task instance for each HDFS data block of the input data. The number of Map function invocations inside a Map task instance is equal to the number of data records in the input data block of the particular Map task instance.

Hadoop MapReduce groups the output key-value records of all the Map tasks of a computation by the **key** and distributes them to the Reduce tasks. This distribution and transmission of data to the Reduce tasks is called the Shuffle phase of the MapReduce computation. Input data to each Reduce task would also be sorted and grouped by the key. The Reduce function gets invoked for each key and the group of values of that key (*reduce <key, list_of_values>*) in the sorted order of the keys. In a typical MapReduce program, users only have to implement the Map and Reduce functions and Hadoop takes care of scheduling and executing them in parallel. Hadoop will rerun any failed tasks and also provide measures to mitigate any unbalanced computations.

Example: word count

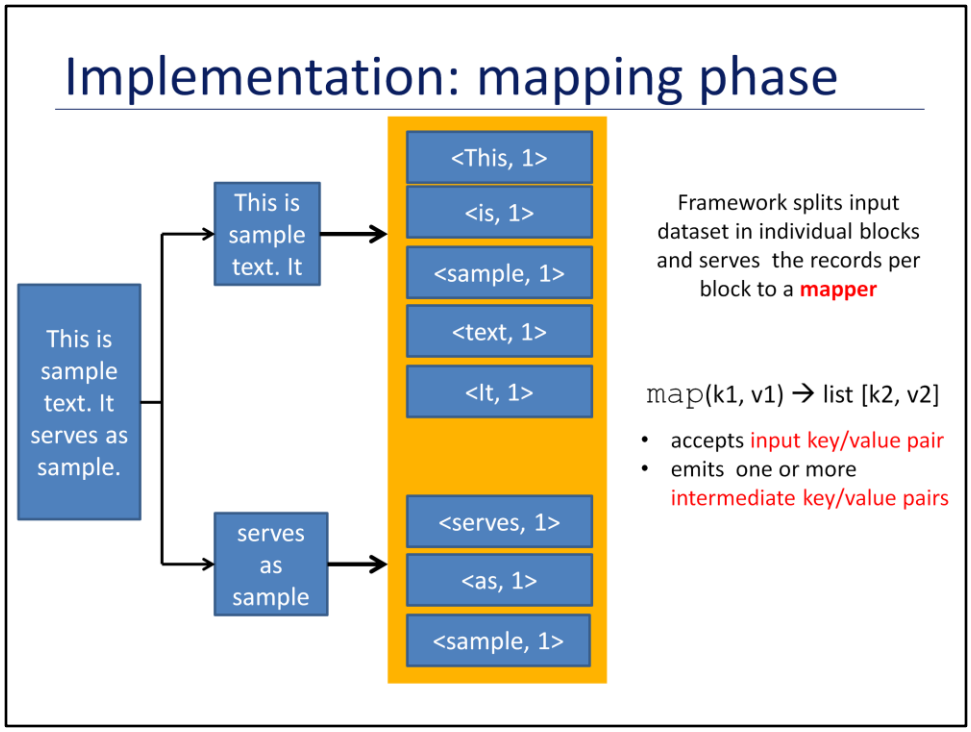
Count how many times each word occurs in a given (huge) text

input

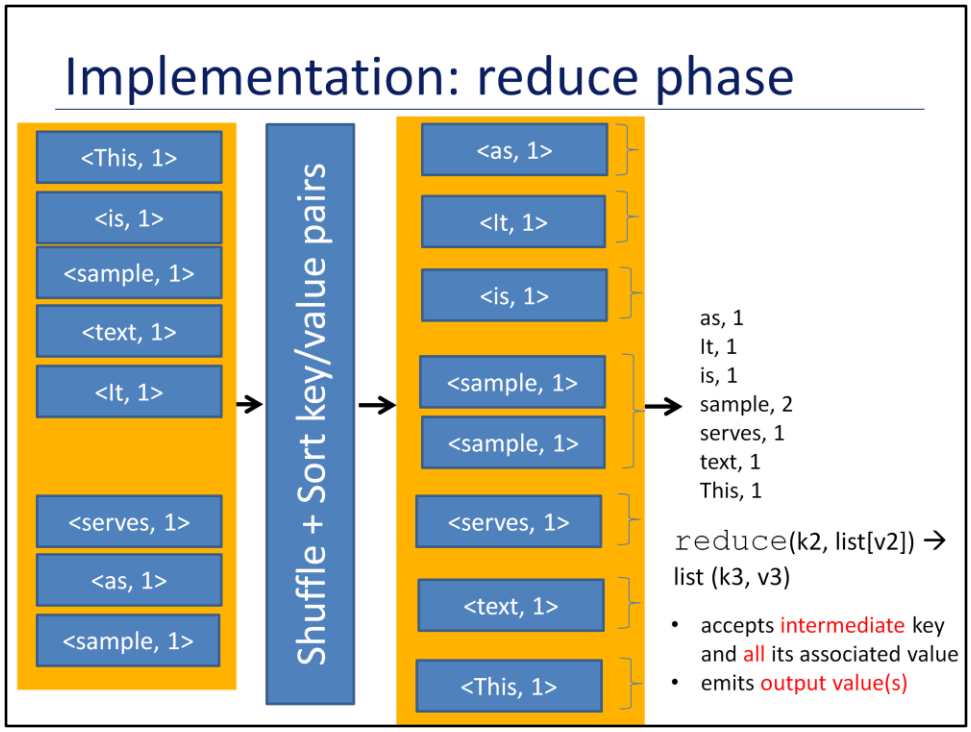
This is sample text
serving as a sample.

output

<this, 1>
<is, 1>
<sample, 2>
…



In this example, each word serves as key in the intermediate <key, value> pairs. The value is always 1 (unsigned integer).



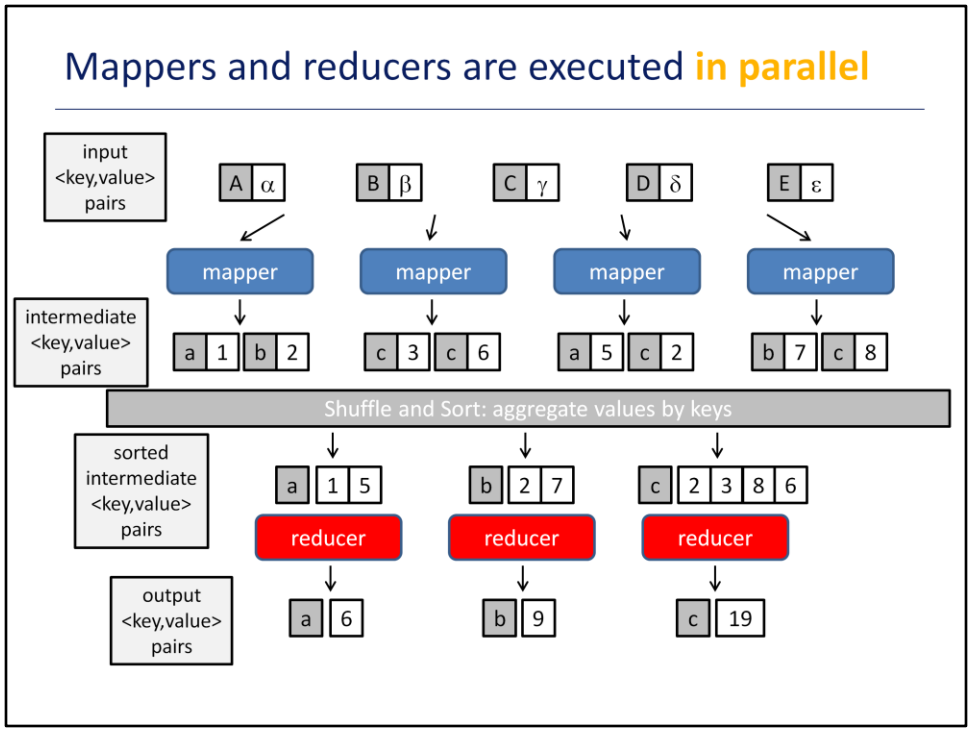
The intermediate pairs are shuffled and sorted across the different nodes of the cluster (based on the key value). Each reducer then handles a number of intermediate keys (and the associated value lists). In this simple example, the reducer simply sums all values in the list and emits this value.

Note that in general the type of intermediate keys/values should not be the same as the type of output keys/values.

Pseudocode

```
function map(record){
        for word in record:
                emit (word, 1)
}
```

```
function reduce(key, values){
        emit (key, sum(values))
}
```



Mappers run on the worker nodes and execute map tasks by applying the map function to a part of the input data. Different map tasks can be executed in parallel. This means that there can be no dependencies between different map tasks. In other words, it should be possible to process part of the input data without having access to the remaining parts.

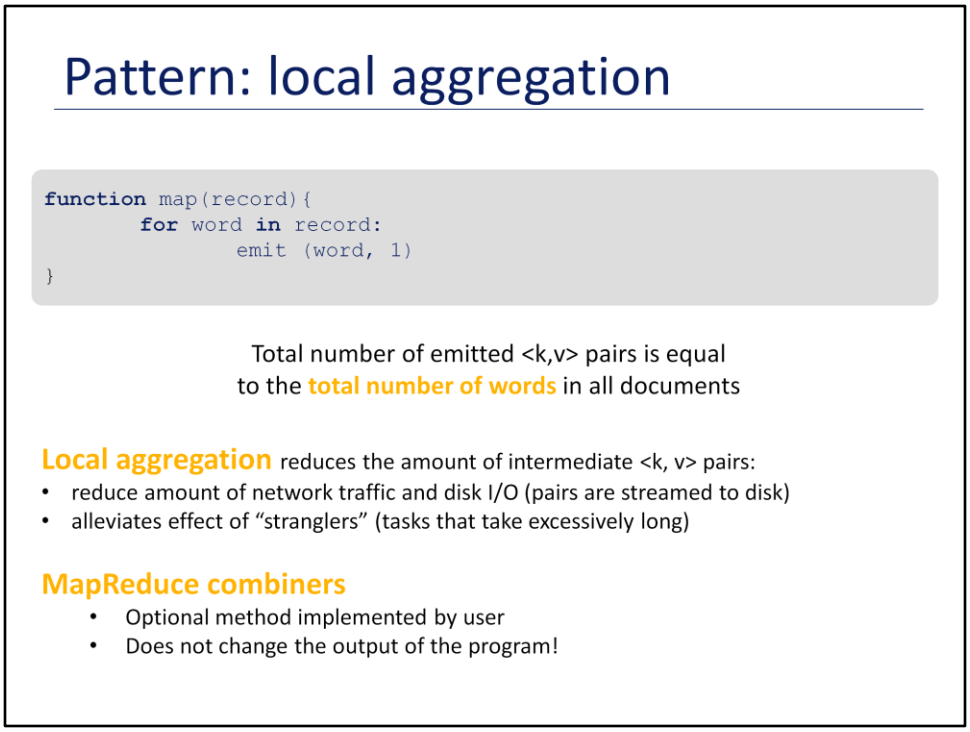
Reducers run on the worker nodes and execute reduce tasks by applying the reduce function on a single intermediate key and its corresponding values. Different reduce tasks can be executed in parallel. This means that there can be no dependencies between different reduce tasks. In other words, it should be possible to process a certain intermediate key and all of its corresponding values without having access to other intermediate keys and values.

MapReduce Design pattern

- User responsibilities
  - prepare the data
  - Implement mapper/reducer (+ combiner/partitioner – see further)
- All the rest is handled by the framework
- Possible extras by the user:
  - Complex, user defined data types in <key, value> pairs
  - User-specific initialization code in mapper/reducer
  - Ability to preserve state across multiple inputs
  - Determine the sort order of intermediate terms
  - Control the partitioning of the intermediate key space

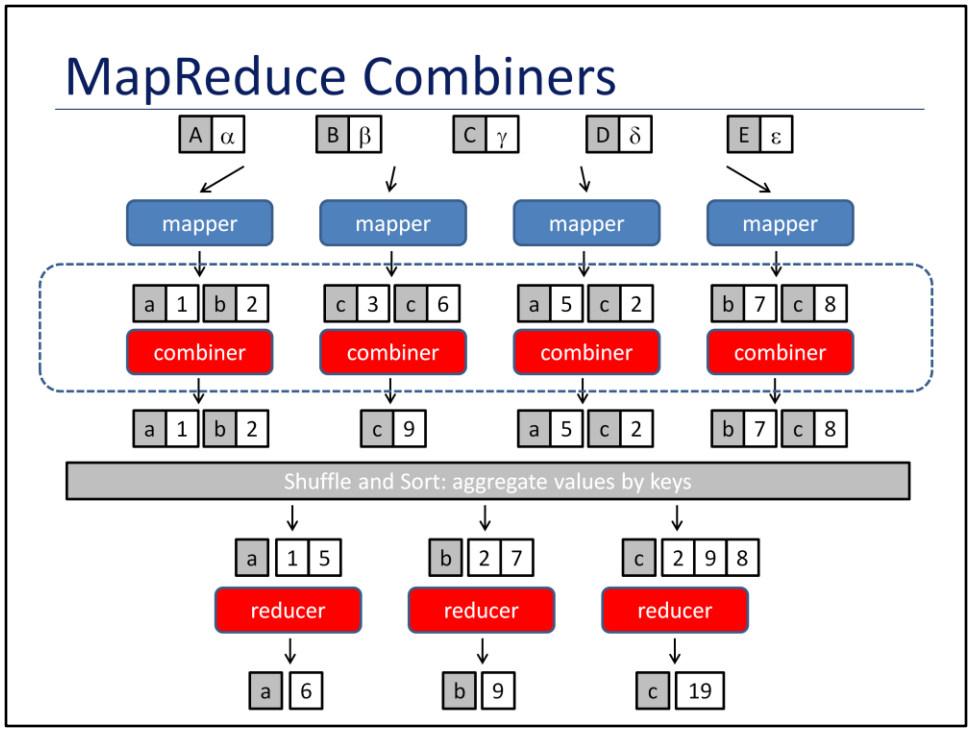
MapReduce design patterns

- Complex algorithms
  - Require sequence or hierarchy of jobs
  - External (sequential) program ("driver") can control and launch MapReduce jobs
- Design patterns help control the scalability
  - data X 2 → time X 2
  - # nodes X 2 → time/2



Even though this implementation is correct, it is not really efficient. The number of emitted <key, value> pairs equals the total number of words in a document (key = a word, value = always the number 1). All these data have to written to disk and sorted, which can be time-consuming for extremely large datasets.

To optimize such scenarios, Hadoop supports a special function called **combiner**, which performs local aggregation of the Map task output key-value pairs. When provided, Hadoop calls the combiner function on the Map task outputs before persisting the data on the disk to shuffle the Reduce tasks. This can significantly reduce the amount of data shuffled from the Map tasks to the Reduce tasks. It should be noted that the combiner is an optional step of the MapReduce flow. Even when you provide a combiner implementation, Hadoop may decide to invoke it only for a subset of the Map output data or may decide to not invoke it at all.



Combiners do **not** reduce the number of intermediate <key, value> pairs emitted by the mappers. They do, however, reduce the number of intermediate <key, value> pairs that have to be **sorted**.

Pseudocode

```
function map(record){
        for word in record:
                emit (word, 1)
}
```

```
function combine(key, [int1, int2, int3]){
        emit (key, sum([int1, int2, int3]))
}
```

```
function reduce(key, [intA, intB]){
        emit (key, sum([intA, intB]))
}
```

This slide shows pseudocode for a combiner in the word counting example. The combiner takes as input a key and a list of values, generated by one or more mappers. In this example, the key will correspond to a specific word. The combiner then emits a pair that has the same key, and the sum of its value list.

Note that the code still works correct when the combiner is not executed. This is an important check you have to do as programmer. The reasoning is quite simply: the combiner simply makes a partial sum of the number of word occurrences in a part of the dataset.

Combiners

- operates like a reducer but only on a subset of the key/values output
- input and output key-value data types must match the mapper output type
- called at the discretion of the framework
  - program must give same result with/without combiners
  - combiners may only be called for subset of key/values
- Only for commutative and associative operations
  - a x b = b x a
  - a x (b x c) = (a x b) x c

Local aggregation correctness

Compute the ***mean of values*** associated with the same key

```
function map(string t, integer r){
        emit (t, r)
}
```

```
function reduce (string t, integer [r1, r2…]){
        s = sum(r1, r2 ...)
       cnt = [r1, r2 ...].length()
        emit (t, s/cnt)
}
```

Cannot use **reducer** as **combiner**:
mean(1,2,3,4,5) ≠ mean(mean(1,2), mean(3,4,5))

In this case, we cannot use a combiner: the mean of the numbers [1, 2, 3, 4, 5] is 3. If we would use combiners, which each calculate the mean on a subset of the data (say [1,2] and [3,4,5]), then we get a different an wrong results.
The reducer cannot distinguish between output coming from a mapper and output coming from a combiner.

What is wrong with this code?

```
function map(string t, integer r){
        emit(t, r)
}
```

```
function combine(string t, integer [r1, r2…]){
        s = sum(r1, r2 ...)
       cnt = [r1, r2 ...].length()
        emit (t, pair(sum, cnt))
}
```

```
function reduce(string t, pairs [(s1,c1), (s2,c2)...]){
        sum = sum_1st ([(s1,c1), (s2,c2)...] ) //first element
        cnt = sum_2nd([(s1,c1), (s2,c2)...]) //second element
        r_avg = sum/cnt
        emit(t, r_avg)
}
```

To address the problem of mean calculation introduced on the previous slide using combiners, one could argue that we let combiners only calculate the partial sum and the count of how many numbers are in that sum. The combiner thus emits a value that consists of a pair. The elements of this pair are the partial sum, and the number of elements summed. This way, the reducer would be able to correctly calculate the mean: it sums the partial sums, and it knows how many numbers there were in total in the dataset.

The code above is however incorrect in one aspect…

Must match with output of mapper!

```
function map(string t){
        emit(t, pair(r, 1))
}
```

```
function combine(string t, pairs [(s1,c1), (s2,c2)...]){
        s = sum(r1, r2 ...)
       cnt = [r1, r2 ...].length()
        emit (t, pair(sum, cnt))
}
```

```
function reduce(string t, pairs [(s1,c1), (s2,c2)...]){
        sum = sum_1st ([(s1,c1), (s2,c2)...] ) //first element
        cnt = sum_2nd([(s1,c1), (s2,c2)...]) //second element
        r_avg = sum/cnt
        emit(t, r_avg)
}
```

The code must keep giving the correct result, even if the combiner is only applied to a subset of the key-value pairs generated by the mapper! Otherwise stated, the reducer cannot know which of the <key, value> pairs it receives are coming from a mapper and which ones are coming from a reducer.

This also means that the output of the mapper and the output of the combiner must have the same format. Hence, we must adjust the mapper to also emit a value pair. The count element in the emitted pair is always set to 1.

Pattern: constructing complex keys/values

- package data in more complex key/value types
  - e.g. pair, array
- example: co-occurrence matrix $[m_{ij}]$
  - m_{ij} = # of co-occurences of w_i and w_j
  - space requirement $O(n^2)$
  - many use cases
    - to identify correlated product purchases
    - to identify words likely to occur nearby a specific word
- two patterns
  - "pairs": emit ($<w_i, w_j>$, 1)
  - "stripes": emit(w_i, $H(w_j)$) (H = associative array)

One common approach for synchronization in MapReduce is to construct complex keys and values in such a way that data necessary for a computation are naturally brought together by the execution framework. We first touched on this technique in the previous slides, in the context of "packaging" partial sums and counts in a complex value (i.e., pair) that is passed from mapper to combiner to reducer.

In the next slides, we will study another common MapReduce example: the calculation of co-occurrences between item i and item j. Examples are e.g. to identify correlated product purchases (how many times are item i and item j bought together), or to identify common word patterns (how many times is word i found next to word j in all documents in the dataset)?

We will study two approaches to solve this pattern. Viewed abstractly, the pairs and stripes algorithms represent two different approaches to counting co-occurring events from a large number of observations. This general description captures the gist of many algorithms in fields as diverse as text processing, data mining, and bioinformatics. For this reason, these two design patterns are broadly useful and frequently observed in a variety of applications.

Pairs approach

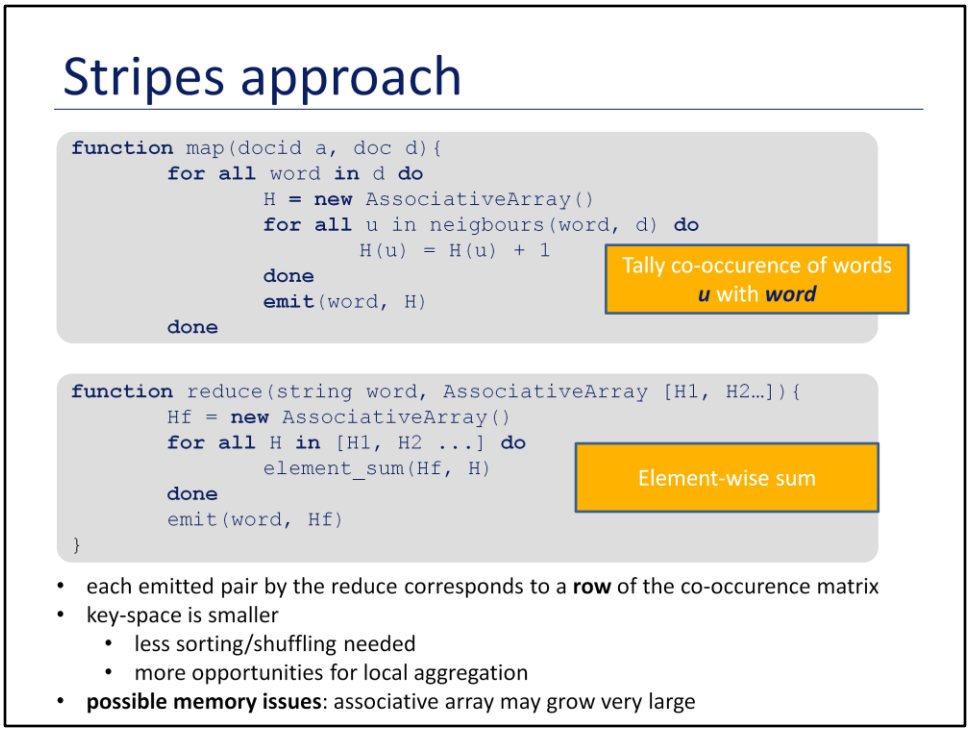
```
function map(docid a, doc d){
        for all word in d do
                for all u in neigbours(word, d) do
                        emit(pair(word, u), 1)
                done
        done
```

```
function reduce(pair p, integer [r1, r2…]){
        emit(p, sum([r1, r2]))
}
```

- each emitted pair by the reduce corresponds to a **cell** of the co-occurence matrix
- generates enormous amount of key-value pairs (even after combining)
- no memory issues even for very large datasets (only pairs are emitted)

Pseudo-code for the "pairs" approach is shown here. Document ids and the corresponding document content make up the input key-value pairs. The mapper processes each input document and emits intermediate key-value pairs with each co-occurring word pair as the key and the integer one (i.e., the count) as the value. This is straightforwardly accomplished by two nested loops: the outer loop iterates over all words (the left element in the pair), and the inner loop iterates over all neighbors of the first word (the right element in the pair). The neighbors of a word can either be defined in terms of a sliding window (e.g. no further away than N words) or some other contextual unit such as a sentence.

The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Thus, in this case the reducer simply sums up all the values associated with the same co-occurring word pair to arrive at the absolute count observed in the dataset, which is then emitted as the final key-value pair. Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.



An alternative approach, dubbed the "stripes" approach, is presented here. Like the pairs approach, co-occurring word pairs are generated by two nested loops. However, the major difference is that instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted H. The mapper emits key-value pairs with words as keys and corresponding associative arrays as values, where each associative array encodes the co-occurrence counts of the neighbors of a particular word.

The MapReduce execution framework guarantees that all associative arrays with the same key will be brought together in the reduce phase of processing. The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key. In contrast to the pairs approach, each final key-value pair encodes a row in the co-occurrence matrix.

It is immediately obvious that the pairs algorithm generates an immense number of key-value pairs compared to the stripes approach. The stripes representation is much more compact, since with pairs the left element is repeated for every co-occurring word pair. The stripes approach also generates fewer and shorter intermediate keys, and therefore the execution framework has less sorting to perform. However, values in the stripes approach are more complex, and come with more serialization and deserialization overhead than with the pairs approach.

Both algorithms can benefit from the use of combiners, since the respective operations in their reducers (addition and element-wise sum of associative arrays) are both commutative and associative. However, combiners with the stripes approach have more opportunities to perform local aggregation because the key space is the vocabulary— associative arrays can be merged whenever a word is encountered multiple times by a mapper. In contrast, the key space in the pairs approach is the cross of the vocabulary with itself, which is far larger—counts can be aggregated only when the same co-occurring word pair is observed multiple times by an individual mapper (which is less likely than observing multiple occurrences of a word, as in the stripes case).

Pairs vs Stripes

- In general, stripes is faster
  - but has potential memory bottleneck
- approaches are endpoints of a continuum
  - divide entire vocabulary in *b* buckets (e.g. by hashing)
  - words co-occuring with *w* spread over *b* assoc. arrays
  - keys emitted are form (*w*, 1), (*w*,2) ... (*w*, b)

It is important to consider potential scalability bottlenecks of either algorithm. The stripes approach makes the assumption that, at any point in time, each associative array is small enough to fit into memory—otherwise, memory paging will significantly impact performance. The size of the associative array is bounded by the vocabulary size, which is itself unbounded with respect to the total size of the dataset (in worst case, each word in the dataset is different!). Therefore, as the size increase, this will become an increasingly pressing issue—perhaps not for gigabyte-sized datasets, but certainly for terabyte-sized and petabyte-sized sets that will be commonplace tomorrow. The pairs approach, on the other hand, does not suffer from this limitation, since it does not need to hold intermediate data in memory.

To conclude, it is worth noting that the pairs and stripes approaches represent endpoints along a continuum of possibilities. The pairs approach individually records each co-occurring event, while the stripes approach records all co-occurring events with respect to a conditioning event. A middle ground might be to record a subset of the co-occurring events with respect to a conditioning event. We might divide up the entire vocabulary into b buckets (e.g., via hashing), so that words co-occurring with word i (w_i) would be divided into b smaller "sub-stripes", associated with ten separate keys, (w_i , 1),(w_i , 2). . .(w_i , b). This would be a reasonable solution to the memory limitations of the stripes approach, since each of the sub-stripes would be smaller. In the case of b = |V |, where |V | is the vocabulary size, this is equivalent to the pairs approach. In the case of b = 1, this is equivalent to the standard stripes approach.

Pattern: order inversion

- occurrence matrix has absolute counts
  - some words appear more frequently than others
  - [m_{ij}] can be high simply because w_i is very common
- convert to relative frequencies

$$f(w_j \mid w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

  - stripes
    - All data is present in reducer == trivial
    - Memory bottleneck still present
  - pairs
    - pairs <wi, wj> are sent to the reducer
    - denominator is not available in the reducer

Let us build on the pairs and stripes algorithms and continue with our running example of constructing the word co-occurrence matrix M for a large text dataset. Recall that in this large square n × n matrix, where n = |V | (the vocabulary size), cell m_{ij} contains the number of times word w_i co-occurs with word w_j within a specific context. The drawback of absolute counts is that it doesn't take into account the fact that some words appear more frequently than others. Word w_i may co-occur frequently with w_j simply because one of the words is very common. A simple remedy is to convert absolute counts into relative frequencies, $f(w_j \mid w_i)$. That is, what proportion of the time does w_j appear in the context of w_i?

In the formula above, $N(\cdot, \cdot)$ indicates the number of times a particular co-occurring word pair is observed in the corpus. We need the count of the joint event (word co-occurrence), divided by what is known as the marginal (the sum of the counts of the conditioning variable co-occurring with anything else).

Computing relative frequencies with the stripes approach is straightforward. In the reducer, counts of all words that co-occur with the conditioning variable (w_i in the above example) are available in the associative array. Therefore, it suffices to sum all those counts to arrive at the marginal and then divide all the joint counts by the marginal to arrive at the relative frequency for all words. This implementation requires minimal modification to the original stripes algorithm. Through appropriate structuring of keys and values, one can use the MapReduce execution framework to

bring together all the pieces of data required to perform a computation. Note that, as with before, this algorithm also assumes that each associative array fits into memory

In the pairs approach, the reducer receives (w_i , w_j) as the key and the count as the value. From this alone it is not possible to compute $f(w_j | w_i)$ since we do not have the marginal.

Order inversion

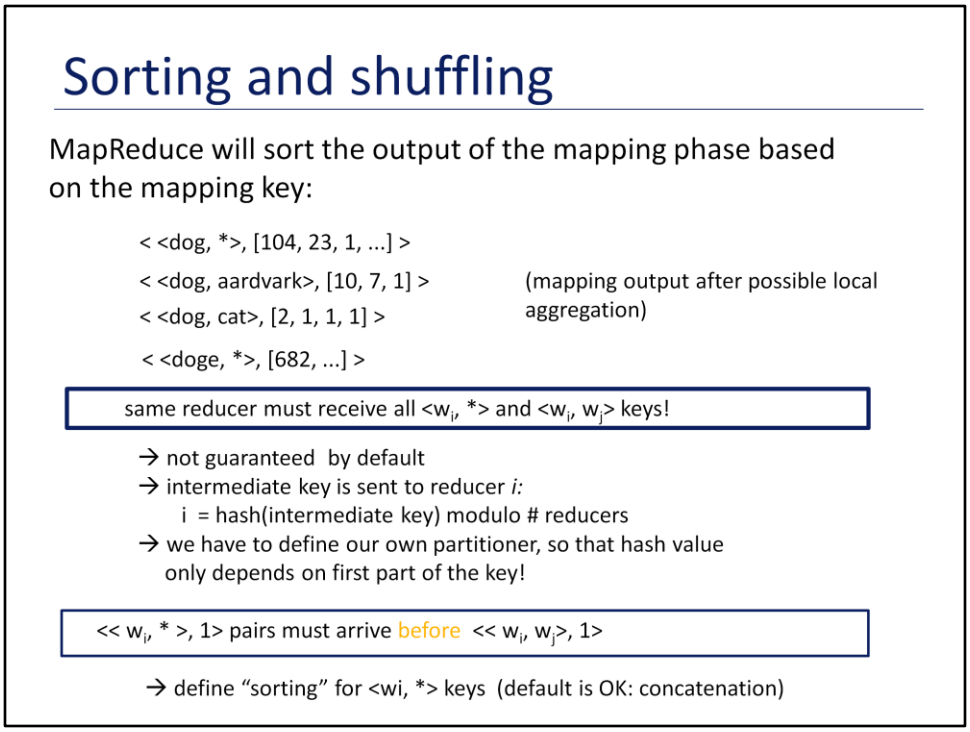
- General idea:
  - compute denominator *before* the pair count
  - mapper emits extra < <w_i, *>, 1 > pair for each <w_i, w_j> pair

```
function map(docid a, doc d){
        for all word in d do
                for all u in neigbours(word, d) do
                        emit(pair(word, u), 1)
                        emit(pair(word, *), 1)
                done
        done
```

- Requirements for the reducer
  - sum << w_i, * >, N> pairs
  - same reducer must receive all <w_i, *> and <w_i, w_j> keys!

Recall that in the basic pairs algorithm, each mapper emits a key-value pair with the co-occurring word pair as the key. To compute relative frequencies, we modify the mapper so that it additionally emits a "special" key of the form (w_i , $*$), with a value of one, that represents the contribution of the word pair to the marginal. Through use of combiners, these intermediate pairs can be aggregated into partial marginal counts (<w_i, *>, N) before being sent to the reducers. In the reducer, we can then sum the values of all (<w_i, *>, N) pairs to get the number of occurrences of w_i.

To accomplish this, we must keep "state" between individual calls to the reduce method. This is possible in principle, using an initialize method in your reducer class. We initialize an array in which we store all intermediate pairs. After all have received, we iterate over the array and emit the calculated relative frequencies. But there is a better way…



Although the algorithm in the previous slide will work, it suffers from the same drawback as the stripes approach: at some point we might run out of memory because we keep all intermediate pairs in arrays in the reducer.

If we could however influence the order in which the intermediate pairs are delivered to the reducers, we can reduce the memory consumption. MapReduce allows programmers to define the sorting order of keys so that intermediate pairs needed earlier is presented to the reducer before intermediate pairs that are needed later.

Note that in this example, we work with keys composed of two words. If we make sure that the key $\langle w_i, * \rangle$ comes "lexicographically" before any other key $\langle w_i, w_j \rangle$, then we can simply sum the values of $\langle w_i, * \rangle$ in the reducer to get the total count of w_i. There is no need for storing intermediate pairs in an array. As the intermediate pairs with key $\langle w_i, w_j \rangle$ arrive, we can simply divide the sum of their values by this total count and emit the relative frequency.

First, the reducer is presented with the special key <dog, *> and a number of values, each of which represent a partial marginal contribution from the map phase. The reducer accumulates these values to arrive at the total number of occurrences of the word "dog". The reducer holds this value in its internal state as it processes subsequent keys, like <dog, aardvark> and <dog, cat>. When it encounters a new special key (<doge, *>), it resets its internal state and the process begins again.

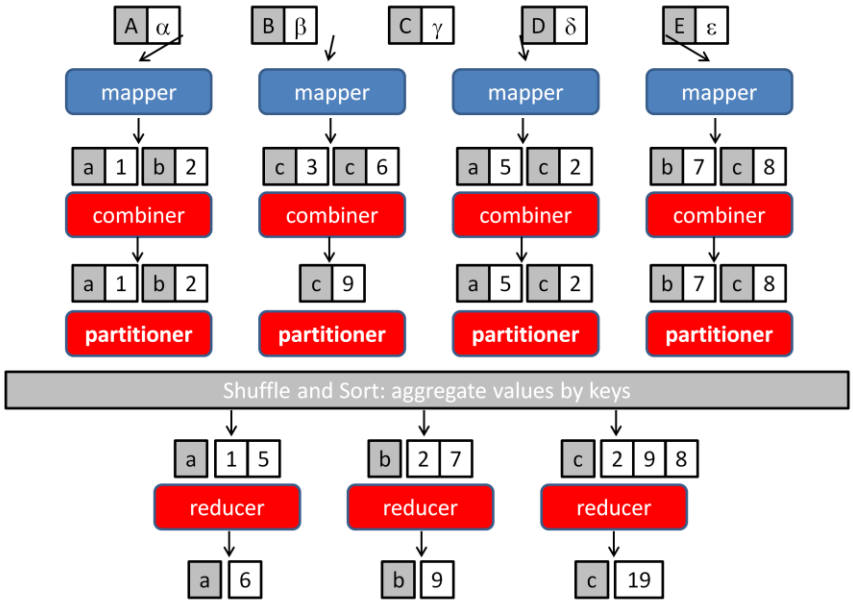
However, even with correct sorting, we must ensure that all pairs with the same left word (w_i) in the key are sent to the **same reducer**. This, unfortunately, does not happen automatically: the default partitioner is based on the hash value of the intermediate key, modulo the number of reducers. For a complex key, the raw byte representation is used to compute the hash value. As a result, there is no guarantee that, for example, intermediate pairs with key (dog, aardvark) and with key (dog, zebra) are assigned to the same reducer. To produce the desired behavior, we must define a custom partitioner that only pays attention to the left word. That is, the partitioner should partition based on the hash of the left word only.

This design pattern, which we call "order inversion", occurs surprisingly often and across applications in many domains. It is so named because through proper coordination, we can access the result of a computation in the reducer (for example, an aggregate statistic) before processing the data needed for that computation. The key insight is to convert the sequencing of computations into a sorting problem. In most cases, an algorithm requires data in some fixed order: by controlling how keys are sorted and how the key space is partitioned, we can present data to the reducer in the order necessary to perform the proper computations. This greatly cuts down on the amount of partial results that the reducer needs to hold in memory.

To summarize, the specific application of the order inversion design pattern for computing relative frequencies requires the following:

- Emitting a special key-value pair for each co-occurring word pair in the mapper to capture its contribution to the marginal.
- Controlling the sort order of the intermediate key so that the key-value pairs representing the marginal contributions are processed by the reducer before any of the pairs representing the joint word co-occurrence counts.
- Defining a custom partitioner to ensure that all pairs with the same left word are shuffled to the same reducer.
- Preserving state across multiple keys in the reducer to first compute the marginal based on the special key-value pairs and then dividing the counts by the marginals to arrive at the relative frequencies

Complete MapReduce framework

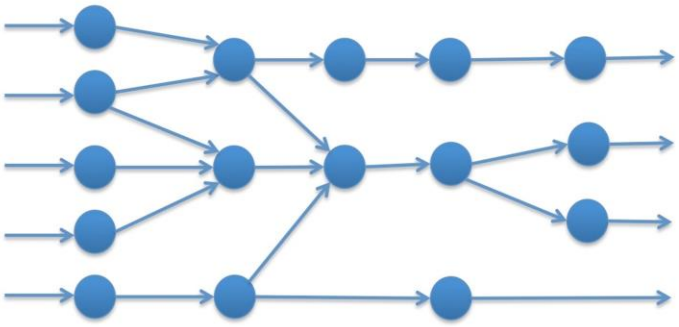


Further reading

- N. Marz and J. Warren, “Big Data”
- Hadoop MapReduce v2 Cookbook
- J. Lin and C. Dyer, “Data-Intensive Text Processing with MapReduce”
- S. Karanth, “Mastering Hadoop”

STREAM PROCESSING

Stream (real-time) processing



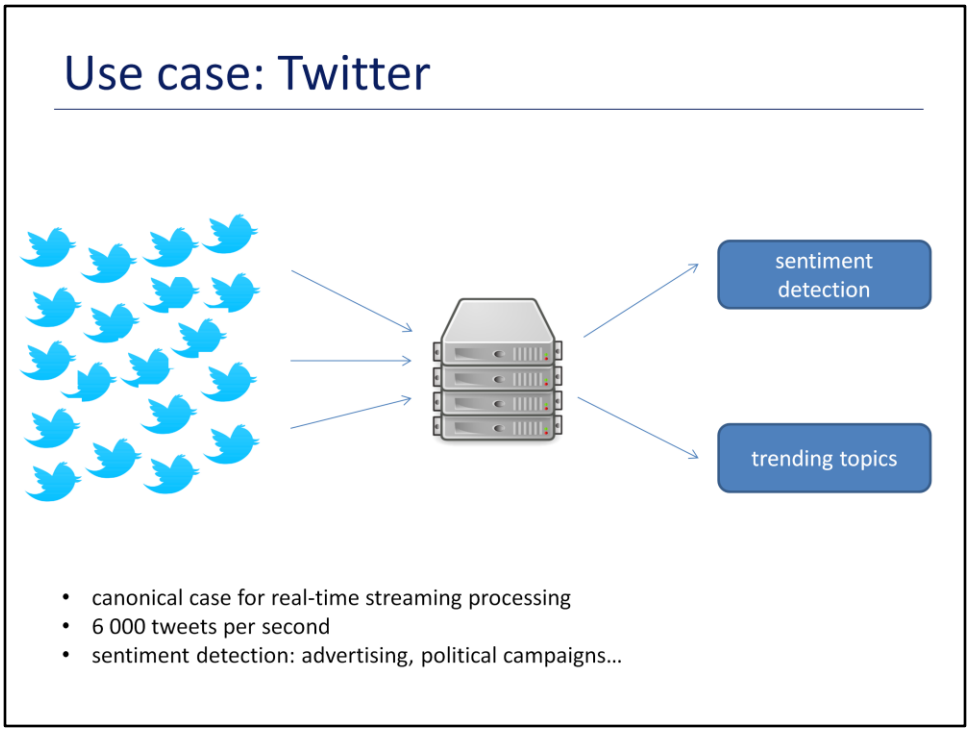
The Velocity in Big Data

- Batch processing is store-first, process-second
  - Unable to scale for real-time Big Data applications
- Stream processing is an alternative
  - Network of concurrently executing but independent continuous queries
  - Operate on data streams as they flow through
  - Scale with number of incoming streams, not the size of the data

Batch processing is store-first, process-second. However, in many cases we need to be able to analyze data as it comes in (e.g. from sensors, bank card transactions, etc.). Real time (or stream) data processing and analytics allows to take immediate action for those times when acting within seconds or minutes is significant. The goal is to obtain the insight required to act prudently at the right time - which increasingly means immediately.

The high level architecture of a stream processor is a network of processing nodes, where each node performs some action or transformation on the data as it flows through. Each node in the system is a continuously executing and independent query that performs operations on data streams such as filtering, aggregation and analytics.

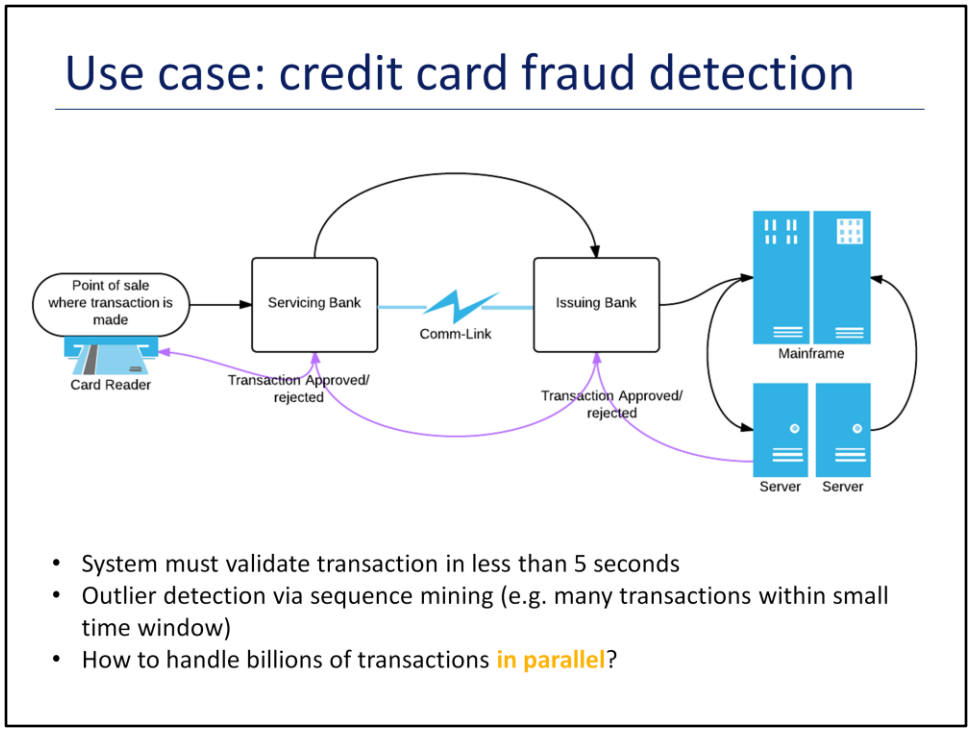
- Each node is an independent continuous query, that is, a query that never ends.
- All nodes execute concurrently, subscribing or consuming one or more input data streams, and generating one or more output streams.
- The stream processing platform is responsible for the scheduling, query optimization, and runtime execution management, including the movement (or clocking) of data through the system.



Twitter allows users to post tweets, messages of up to 140 characters, on its social network. The number of tweets sent per day is sometimes daunting. Every second, on average, around 6,000 tweets are tweeted on Twitter [1], which corresponds to over 350,000 tweets sent per minute, **500 million tweets per day** and around 200 billion tweets per year.

Most work to date has focused on post-facto analysis of tweets, with results coming days or even months after the collection time. However, because tweets are short and easy to send, they lend themselves to quick and dynamic expression of instant reactions to current events. Automated real-time sentiment analysis of this user-generated data can provide fast indications of changes in opinion, showing for example how an audience reacts to particular candidate's statements during a political debate.

[1] www.internetlivestats.com/twitter-statistics



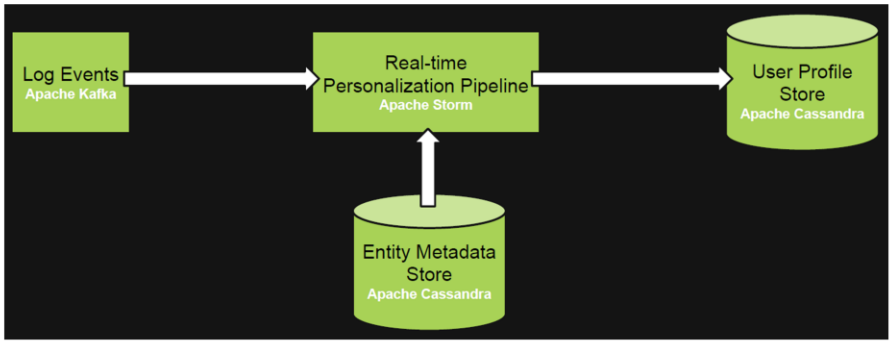
When we make any transaction and swipe our debit or credit card for payment, the duration within which the bank has to validate or reject the transaction in less than five seconds. In less than five seconds, data or transaction details have to be encrypted, travel over secure network from servicing back bank to the issuing bank, then at the issuing back bank the entire fuzzy logic for acceptance or decline of the transaction has to be computed, and the result has to travel back over the secure network.

The challenges such as network latency and delay can be optimized to some extent, but to achieve the preceding featuring transaction in less than 5 seconds, one has to design an application that is able to churn a considerable amount of data and generate results within 1 to 2 seconds.

A sample of how fraud detection is implemented can be found here: https://pkghosh.wordpress.com/2013/10/21/real-time-fraud-detection-with-sequence-mining/

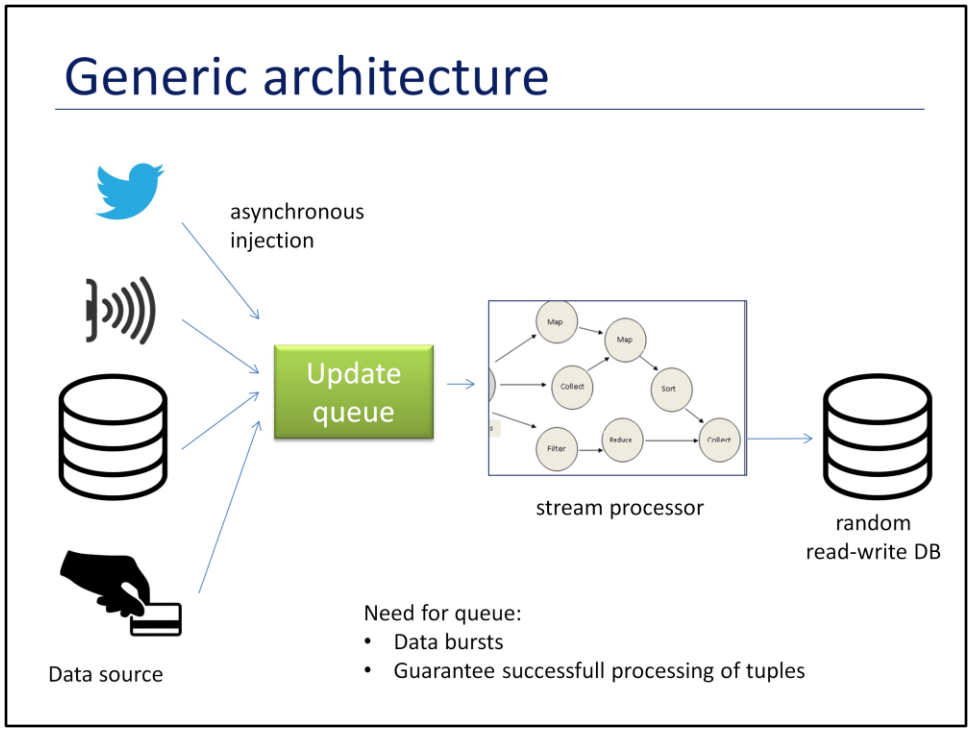
Use case: Spotify

- 200 000 tuples per second
- recommendation, ads, monitoring, analytics
  - real-time changing contexts and conditions



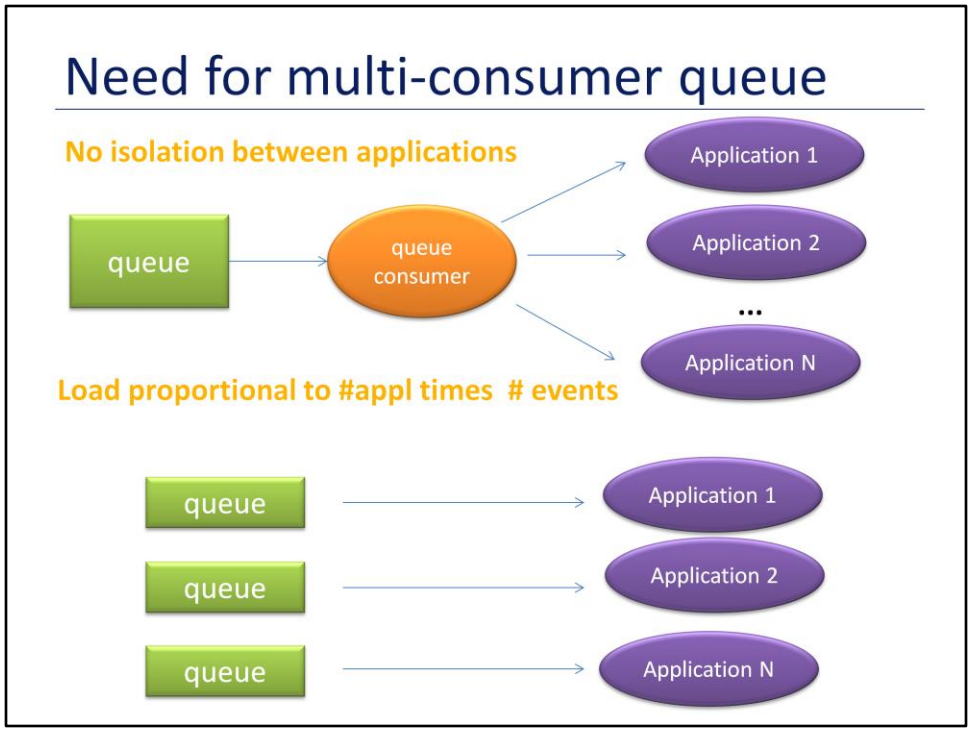
Also Spotify uses real-time streaming for recommendation, advertisement, monitoring and analytics. Recalculating recommendations over their entire batch of data (playlists, play history, user profile) would be infeasible: users can quickly skip a number of recommended tracks, which would quickly exhaust the calculated list of suggestions.

Moreover, it is important to cope quickly with changing contexts and conditions. For example, a metal genre listener might not enjoy an announcement for a metal genre album when they are trying to put their kid to sleep and playing kid's music at night.



The high-level architecture of any stream processing framework is depicted on the slide. The architecture is asynchronous: data sources simply inject their data into the stream processing framework without waiting for confirmation that the data has been persisted in the DB. Data sources are often external; or serve many consumers, so working synchronously would block them.

Data sources are injecting their data into an update queue. Data sources can generate data from specific events (tweets, sensor data, credit card transactions), but may also emit data from a database. To understand the need for a queue, let's hypothetize on a system without a queue. In such a system, events would be handed directly to worker nodes in the stream processor that would process each event independently. This fire-and-forget system cannot guarantee that all the data is successfully processed. A worker can die before completing its assigned task, but there is no mechanism to detect or correct the error. The architecture is also susceptible to bursts in traffic that exceed the resources of the processing cluster. In such a scenario, the client would be overwhelmed and messages could be lost.



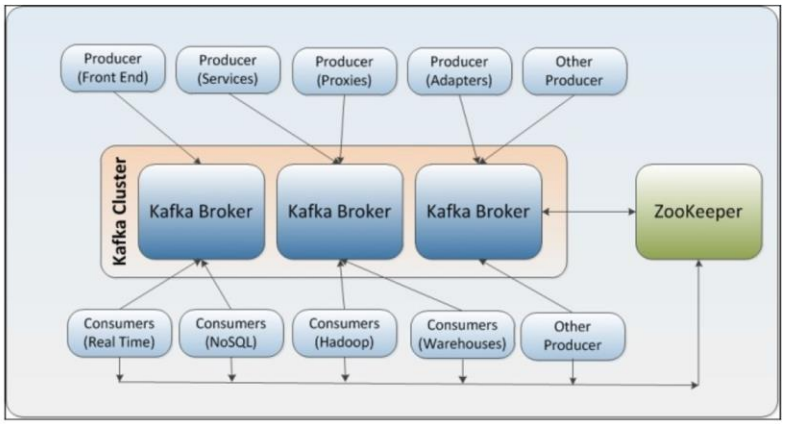
While it is now clear that an asynchronous architecture needs a queue to persist an event stream, the semantics of good queue design require further discussion. Many implementations, including the standard Java Queue and RabbitMQ are single-consumer queues. This design is based on the idea that when you read an event from the queue, the event is not immediately removed. Instead, the item taken from the queue contains an identifier that you later use to acknowledge success or report failure for the processing of the event. Only when an event is acked will it be removed from the queue. If the event processing fails or a timeout occurs, the queue server will allow another client to retrieve the same event via another get call. An event may therefore be processed multiple times with this approach (for example, when a client processes an event but dies before acknowledging it), but each event is guaranteed to be processed at least once.

There is a deep flaw in this queue design: what if multiple applications want to consume the same stream? This is quite common. For example, given a stream of pageviews, one application could build a view of pageviews over time while another could build a view of unique visitors over to time. One possible solution would be to wrap all the applications within the same consumer. This is a bad design as it eliminates any isolation among independent applications. If one application has a bug, it could potentially affect all the other applications running within the same consumer. With a single-consumer queue, the only way to achieve independence between applications is to maintain a separate queue for each consumer application.

If you have three applications, you maintain three separate copies of the queue. The load is now proportional to the number of applications multiplied by the number of incoming events, rather than just to the number of incoming events.

What we really need is a single queue that can be used by many consumers, where adding a consumer is simple and introduces a minimal increase in load. The fundamental issue with a single-consumer queue is that the queue is responsible for keeping track of what's consumed. Because of the restrictive condition that an item is either 'consumed' or 'not consumed', the queue is unable to gracefully handle multiple clients wanting to consume the same item.

Apache Kafka



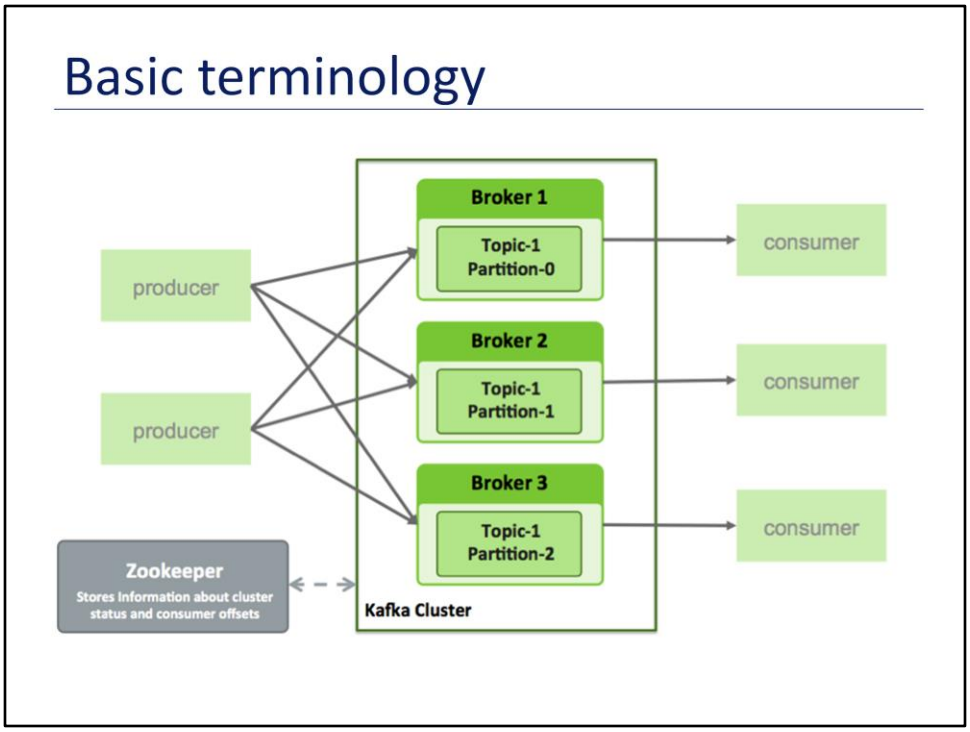
Open source, distributed, partitioned and replicated commit-log publish-subscribe messaging system

- Persistent messaging – no information loss
- High throughput – even on commodity hardware
- Real time – messages are immediately visible to consumer threads

Apache Kafka is an open source, distributed, partitioned, and replicated commit-log-based publish-subscribe messaging system, mainly designed with the following characteristics:

- **Persistent messaging**: To derive the real value from big data, any kind of information loss cannot be afforded. Apache Kafka is designed with O(1) disk structures that provide constant-time performance even with very large volumes of stored messages that are in the order of TBs. With Kafka, messages are persisted on disk as well as replicated within the cluster to prevent data loss.
- **High throughput**: Keeping big data in mind, Kafka is designed to work on commodity hardware and to handle hundreds of MBs of reads and writes per second from large number of clients.
- **Distributed**: Apache Kafka with its cluster-centric design explicitly supports message partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics. Kafka cluster can grow elastically and transparently without any downtime.
- **Real time**: Messages produced by the producer threads should be immediately visible to consumer threads; this feature is critical to event-based systems.

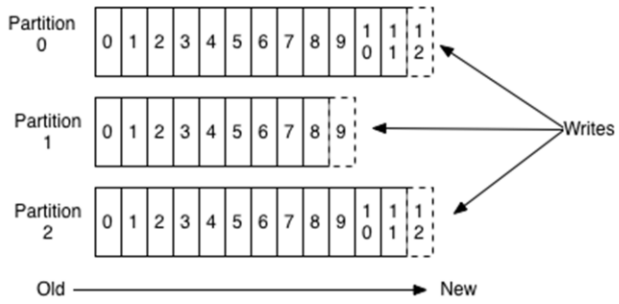
The diagram on the slide shows a typical big data aggregation-and-analysis scenario supported by the Apache Kafka messaging system, with different kinds of producers and different kinds of consumers.



A Kafka cluster primarily has five main components:

- Kafka maintains feeds of messages in categories called ***topics***. A topic is a category or feed name to which messages are published by the message producers. In Kafka, topics are partitioned.
- Kafka cluster consists of one or more servers where each one may have one or more server processes running and is called the **broker**. Topics are created within the context of broker processes.
- **ZooKeeper** serves as the coordination interface between the Kafka broker and consumers. ZooKeeper is akin to etcd: it allows distributed processes to coordinate with each other. It stores coordination data: status information, configuration, location information, and so on.
- Process that publish messages to a Kafka topic are referred to as ***producers***.
- Processes that subscribe to topics and process the feed of published messages are referred to as ***consumers***.

Topics: anatomy and publishing

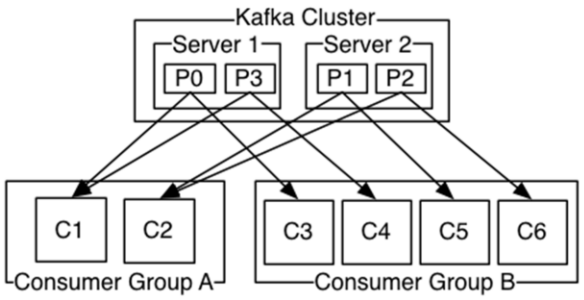


- Topic comprises a partitioned log
- Each log is ordered, immutable sequence of messages
- Append-only
- Producers choose appropriate partition within the topic to write to

For each topic, Kafka maintains a partitioned log as shown on the slide. Each partition is an ordered, immutable sequence of messages that is continually appended to – a commit log. The messages in the partitions are each assigned a sequential id number called the *offset* that uniquely identifies each message within the partition.

The partitions in the log serve several purposes. First, they allow the log to scale beyond a size that will fit on a single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data. Second they act as the unit of parallelism. Producers publish data to the topics by choosing the appropriate partition within the topic. For load balancing, the allocation of messages to the topic partition can be done in a round-robin fashion or using a custom defined function.

Consuming a topic



- Consumer processes ordered in **consumer groups**
- Message within a topic is consumed by a single process per consumer group
- Ordering guarantees *within* a partition: one partition → one consumer process
- Broker maintains offset per consumer, but offset is controlled entirely by consumer
- Cluster has retention period – older messages are automatically discarded

The Kafka platform has the concept of consumer groups. Here, each consumer is represented as a process and these processes are organized within groups called **consumer groups**.

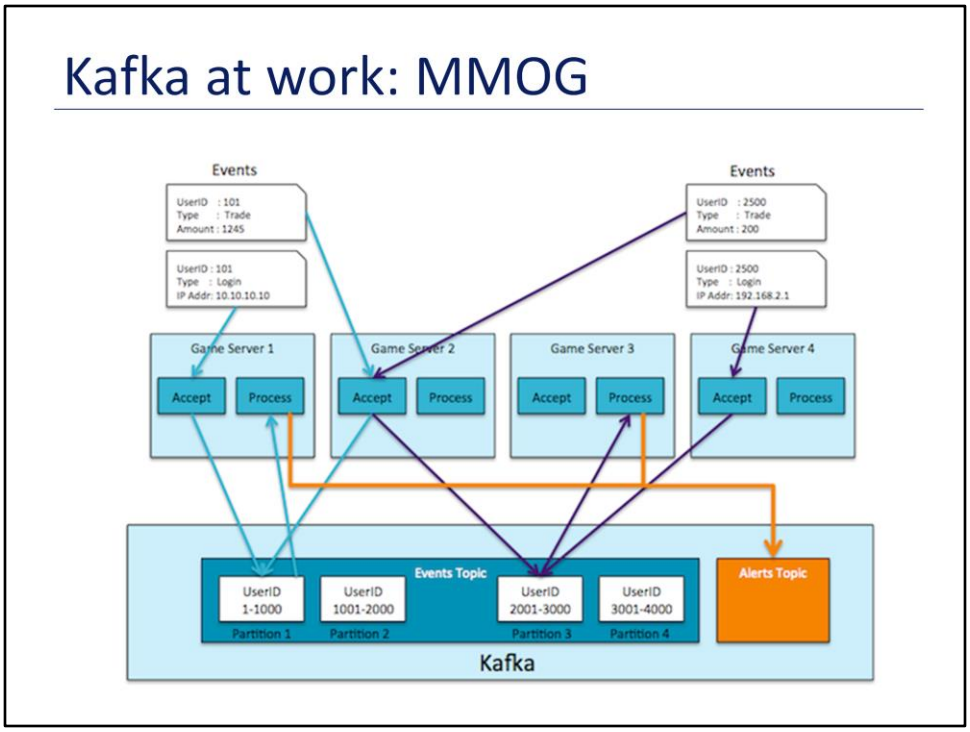
A message within a topic is consumed by a single process (consumer) within the consumer group and, if the requirement is such that a single message is to be consumed by multiple processes, all these consumer processes need to be kept in different consumer groups.

Kafka only provides a total order over messages *within* a partition. To achieve this, Kafka assigns the partitions in the topic to the consumers in the consumer group: each partition is consumed by exactly one consumer in the group. This ensures that a single process is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. Note however that there cannot be more consumer processes than partitions.

Consumers always consume messages from a particular partition sequentially and also acknowledge the message offset. This acknowledgement implies that the consumer has consumed all prior messages. Consumers issue an asynchronous pull request containing the offset of the message to be consumed to the broker and get the buffer of bytes.

Brokers are stateless, which means the message state (messages was successfully processed or not) of any consumed message is maintained within the message consumer, and the Kafka broker does not maintain a record of what is consumed by whom. The only metadata retained on a per-consumer basis is the position of the consumer in the log (“the offset”). This offset is controlled by the consumer: normally the consumer will advance its offset as it reads data, but it can consume messages in any order it likes. For example a consumer can reset to an older offset to reprocess.

The Kafka cluster retains all published for a configurable period of time. Messages older than the retention period will be deleted -- whether or not they have been consumed.



We will illustrate Kafka with the use case of a massive multiplayer online game (MMOG). In these games, players cooperate and compete with each other in a virtual world. Often players trade with each other, exchanging game items and money, so as game developers it is important to make sure players don't cheat: trades will be flagged if the trade amount is significantly larger than normal for the player and if the IP the player is logged in with is different than the IP used for the last 20 games. In addition to flagging trades in real-time, we also want to load the data to Apache Hadoop, where our data scientists can use it to train and test new algorithms.

For the real-time event flagging, it will be best if we can reach the decision quickly based on data that is cached on the game server memory, at least for our most active players. Our system has multiple game servers and the data set that includes the last 20 logins and last 20 trades for each player can fit in the memory we have, if we partition it between our game servers.

Our game servers have to perform two distinct roles: The first is to accept and propagate user actions and the second to process trade information in real time and flag suspicious events. To perform the second role effectively, we want the whole history of trade events for each user to reside in memory of a single server. This means we have to pass messages between the servers, since the server that accepts the user action may not have his trade history. To keep the roles loosely coupled, we use Kafka to pass messages between the servers, as you'll see below.

We have configured Kafka with a single topic for logins and trades. The reason we need a single topic is to make sure that trades arrive to our system after we already have information about the login (so we can make sure the gamer logged in from his usual IP). Kafka maintains order within a topic, but not between topics.

When a user logs in or makes a trade, the accepting server immediately sends the event into Kafka. We send messages with the user id as the key, and the event as the value. This guarantees that all trades and logins from the same user arrive to the same Kafka partition. Each event processing server runs a Kafka consumer, each of which is configured to be part of the same group—this way, each server reads data from few Kafka partitions, and all the data about a particular user arrives to the same event processing server (which can be different from the accepting server). When the event-processing server reads a user trade from Kafka, it adds the event to the user's event history it caches in local memory. Then it can access the user's event history from the local cache and flag suspicious events without additional network or disk overhead.

It's important to note that we create a partition per event-processing server, or per core on the event-processing servers for a multi-threaded approach. This may sound like a circuitous way to handle an event: send it from the game server to Kafka, read it from another game server and only then process it. However, this design decouples the two roles and allows us to manage capacity for each role as required. In addition, the approach does not add significantly to the timeline as Kafka is designed for high throughput and low latency; even a small three-node cluster can process close to a million events per second with an average latency of 3 ms.
When the server flags an event as suspicious, it sends the flagged event into a new Kafka topic—for example, Alerts—where alert servers and dashboards pick it up. Meanwhile, a separate process reads data from the Events and Alerts topics and writes them to Hadoop for further analysis.

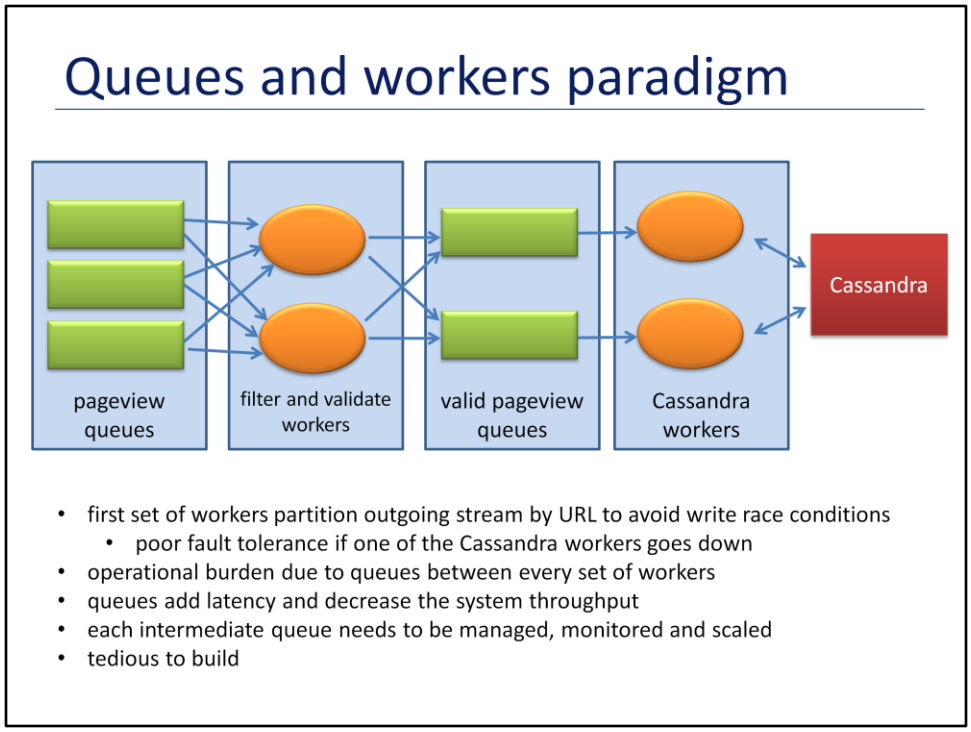
Models for stream processing

	One-at-a-time	Micro-batched
Lower latency	✓	
Higher throughput		✓
At-least-once semantics		✓
Exactly-once semantics	in some cases	✓
Simpler programming model	✓	

One-at-a-time and micro-batched are two recent models of stream processing. There are trade-offs to consider, since each has its strenghts and weaknesses. They are very much complementary – some applications are better suited for one-at-a-time stream processing, and micro-batch stream processing is a better choice for others.

A big advantage of one-at-a-time stream processing is that it can process streams with lower latency than micro-batched processing. Example applications that greatly benefit from this attribute include alerting and financial trading.

The latency and throughput characteristics are different for micro-batch processing. For any individual tuple, the latency from when it's added to the source queue to when it's fully processed is much higher in micro-batch processing. There is a small but significant amount of overhead to coordinating batches that increases latency, and instead of waiting for just one tuple to complete, processing needs to be done on many more tuples. In practice, this turns out to be latency on the order of hundreds of milliseconds to seconds.
But micro-batch processing can have higher throughput than one-at-a-time processing (number of tuples processed per second). Whereas one-at-a-time processing must do tracking on an invididual tuple level, micro-batch processing only has to track at a batch level. This means fewer resources are needed on average per tuple, allowing micro-batch processing to have higher throughput than one-at-a-time processing.



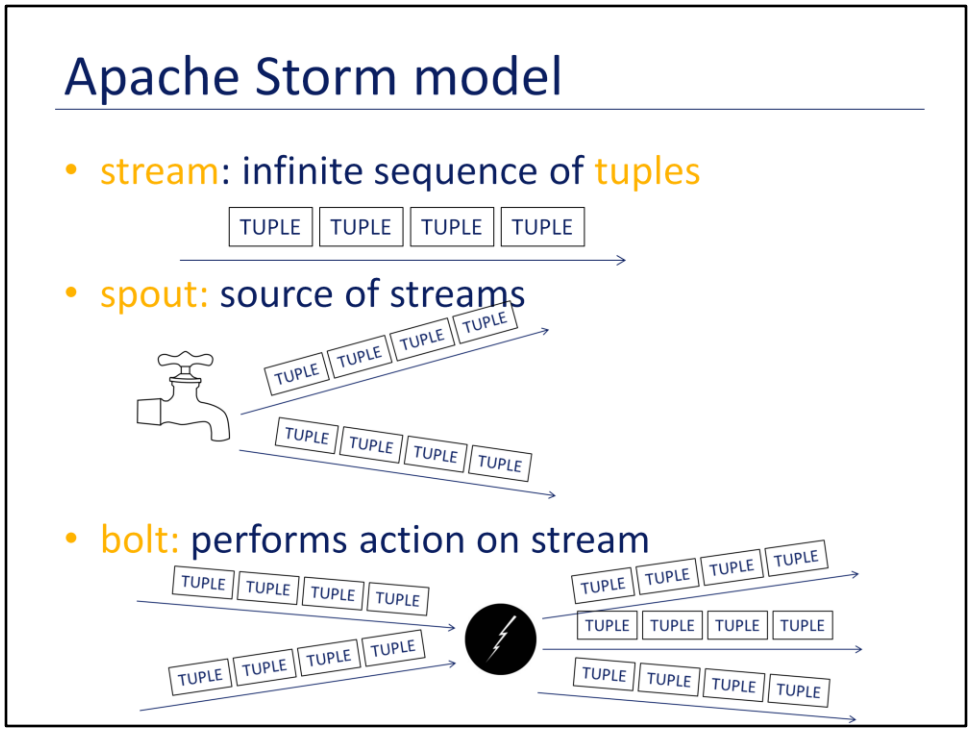
The slide illustrates a one-at-a-time stream processing model for a pageview (per URL) counting problem. The first set of workers reads pageview events from a set of queues, validates each pageview to filter out invalid URLs, and then passes the events to a second set of workers. The second set of workers then updates the pageview counts of the valid URLs.

The queues-and-workers paradigm is straightforward but not necessarily simple. One subtletly is the need to ensure that multiple workers don't attempt to update (increment) the pageview count of the same URL at the same time to avoid race conditions. To meet this guarantee, the first set of workers partitions its outgoing stream by the URL. With this partitioning, the entire set of URLs will still be spread among the queues, but pageview events for any given URL will always go to the same queue (e.g. by taking the modulo of the has of the URL). Unfortunately, a consequence of partitioning over queues is poor fault tolerance. If a worker that updates the pageview counts in the database goes down, no other workers will update the database for that portion of the stream. You'll have to manually start the failed worker somewhere else, or buld a custom system to automatically do so.

Another problem is that having queues between every set of workers adds to the operational burden of your system. If you need to change the topology of your processing, you will need to coordinate your actions so that the intermediate queues are cleared before your redeploy.

Queues also add latency and decrease the throughput of your system because each event passed from worker to worker is forced to go through a third party, where it must be persisted to disk.
On top of everything else, each intermediate queue needs to be managed and monitored and adds yet another layer that needs to be scaled.

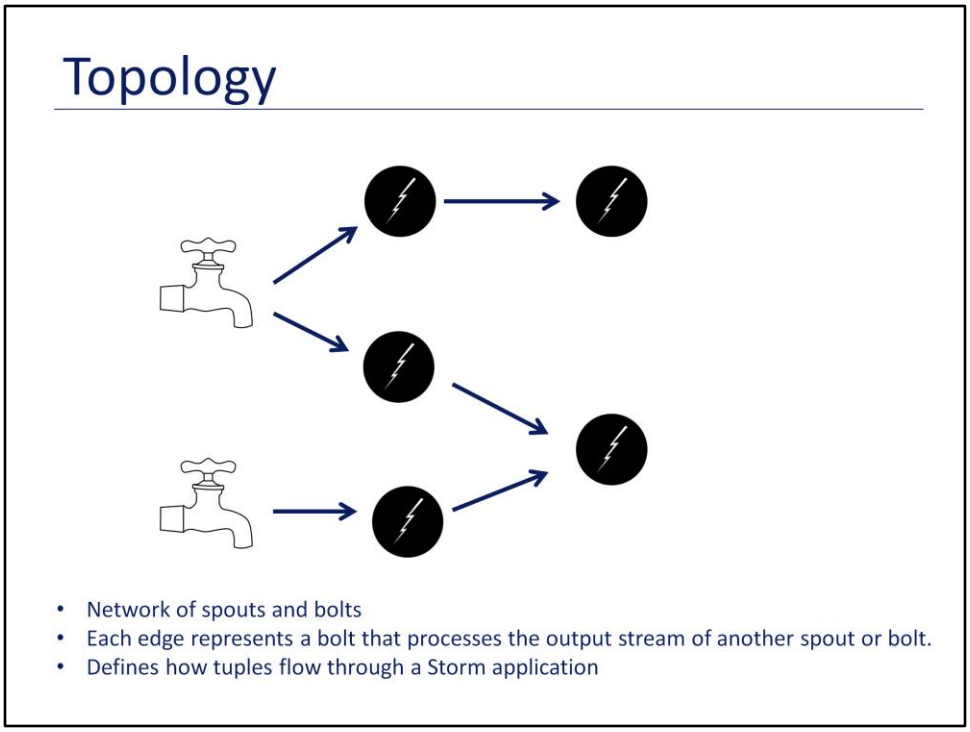
Perhaps the biggest problem with this approach is how tedious it is to build. Much code is required for (de)serialization, to pass objects through queues, routing logic to connect worker pools, and instructions for deploying workers over a cluster of servers.
Clearly, there is need for a higher-level abstraction framework.



At the core of the Storm model are *streams*. A stream is an infinite sequence of *tuples*, where a tuple is simply a named list of values. In essence, the Storm model is about transforming streams into new streams, potentially updating databases along the way.

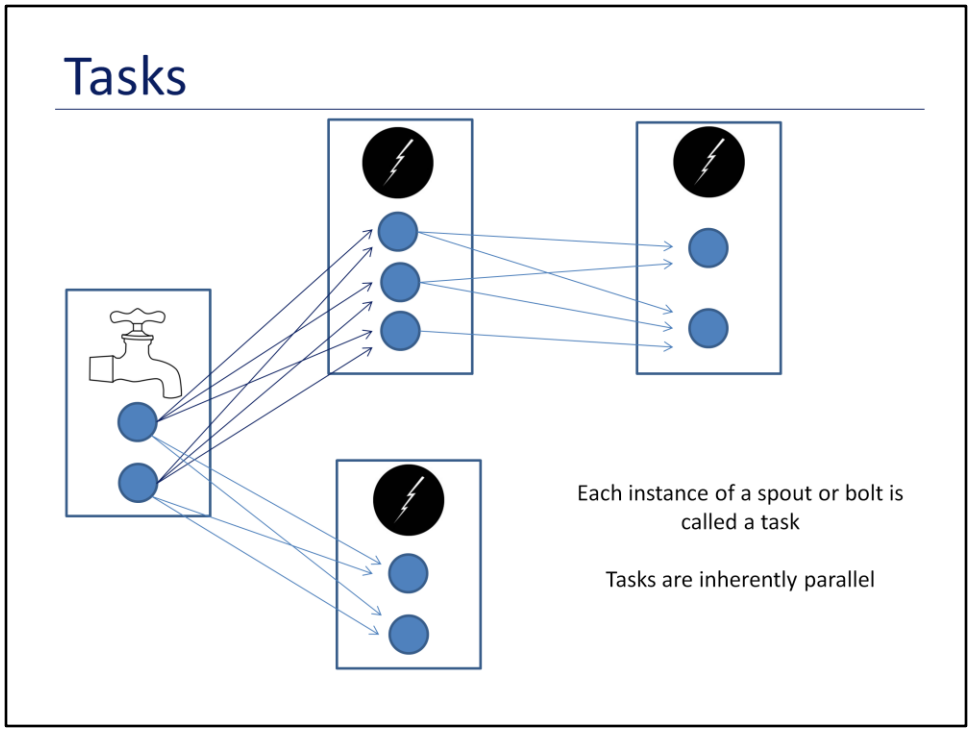
The next abstraction in the Storm model is the *spout*. A spout is a source of streams in a topology. A spout could read from some data source (e.g. a sensor, a Kafka queue) and turn the data into a tuple stream, or a timer spout could emit a tuple into its output stream every 10 seconds.

While spouts are sources of streams, the *bolt* abstraction performs actions on streams. A bolt takes any number of streams as input and produces any number of streams as output. Bolts implement most of the logic in a topology, they run functions, filter data, compute aggregations, do streaming joins, update databases, and so forth.

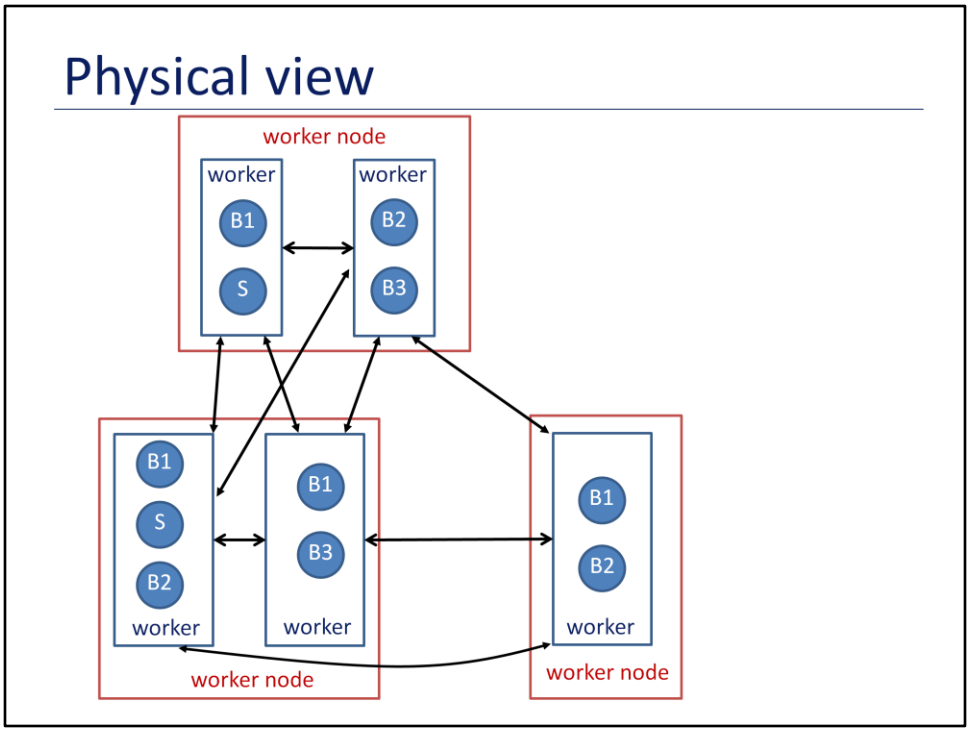


The Storm model represents the entire stream-processing pipeline as a graph of computation called a topology. Rather than write separate programs for each node of the topology and connect them manually, as required in the queues-and-workers scheme, the Storm model involves a single program that's deployed across a cluster. This flexible approach allows a single executable to filter data in one node, compute aggregates with a second node, and update realtime view databases with a third. Serialization, message passing, task discovery, and fault tolerance can be handled for you by the abstractions, and this can all be done while achieving very low latency.

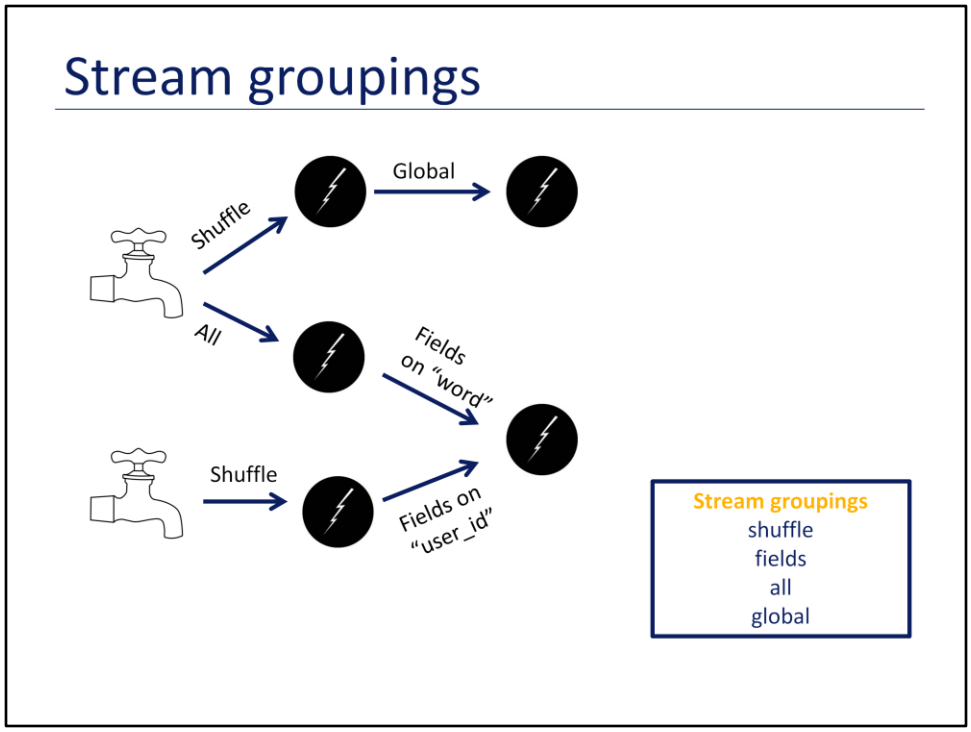
A topology is therefore a network of spouts and bolts with each edge representing a bolt that processes the output stream of another spout or bolt. The topology defines how tuples flow through a Storm application.



Each instance of a spout or bolt is called a *task*. The key to the Storm model is that tasks are inherently parallel – exactly like how map and reduce task are inherently parallel in MapReduce. Spouts and bolts consist of multiple tasks that are executed in parallel. A bolt task receives tuples from all tasks that generate the bolt's input stream.



All the tasks for a given spout or bolt will not necessarily run on the same machine. Instead, they are spread among the different workers of the cluster. The above figure depicts a topology grouped by physical machines.



The fact that spouts and bolts run in parallel brings up a key question: when a task emits a tuple, which of the consuming tasks should receive it? The Storm model requires *stream groupings* to specify how tuples should be partitioned among consuming tasks.

The simplest kind of stream grouping is a *shuffle grouping* that distributes tuples using a random round-robin algorithm. This grouping evenly splits the processing load by distributing the tuples randomly but equally to all consumers. Another common grouping is the *fields grouping* that distributes tuples by hashing a subset of the tuple fields and modding the result by the number of consuming tasks.

All grouping replicates the stream across all the bolt's tasks. This grouping must be used with care.

Global grouping makes the entire stream go to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id.

Example: word counting


```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(),8)
       .shuffleGrouping("spout");
builder.setBolt("count", new WordCount(),12)
       .fieldsGrouping("split", new Fields("word"));
```

Just as word count is the de facto introductory MapReduce example, let’s see what the streaming version of word count looks like in the Storm model.
The splitter bolt transforms a stream of sentences into a stream of words, and the word-count bolt consumes the words to compute the word counts. The key here is the fields grouping between the splitter bolt and the word-count bolt. That ensures that each word-count task sees every instance of every word they receive; making it possible for them to compute the correct count.

Counting bolt

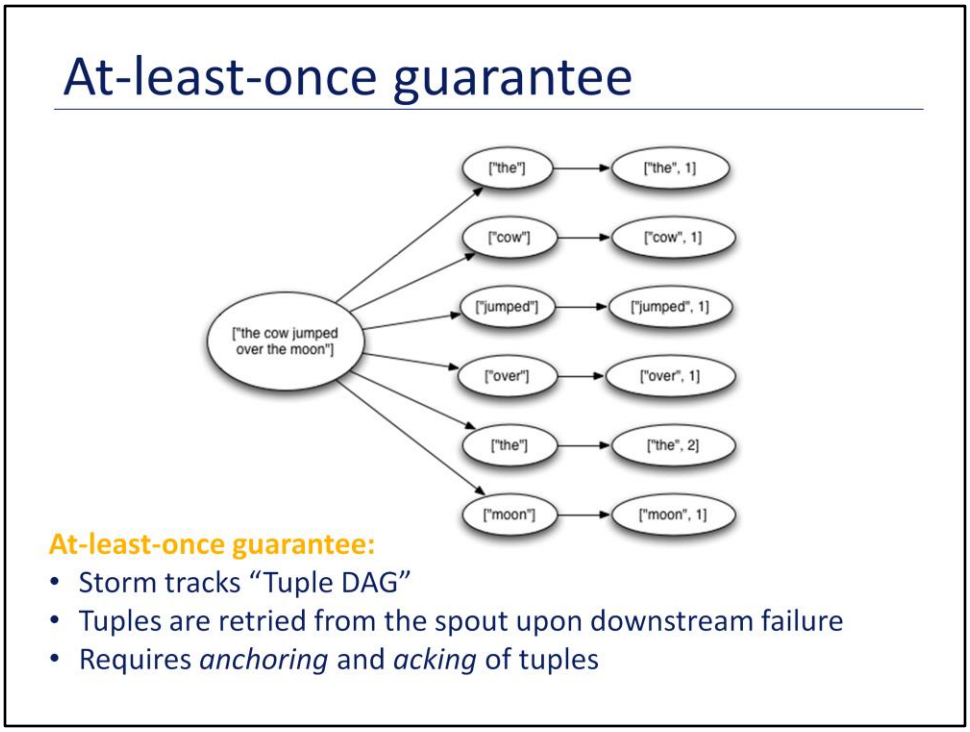
```
public static class WordCount extends BaseBasicBolt {
  Map<String, Integer> counts = new HashMap<String,Integer>();

  @Override
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.get(word);
    if (count == null)
      count = 0;
    count++;
    counts.put(word, count);
    collector.emit(new Values(word, count));
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
     declarer.declare(new Fields("word", "count"));
  }
}
```

This is the code implementing the WordCount bolt in Storm. The execute method receives a tuple and a collector to emit output to. As you can see, you can keep state per Bolt instance. To make sure that your state is consistent between different Bolts, you need to use the appropriate stream groupings. In our word count example, this means that each word should be in the Map object of only one Bolt instance.

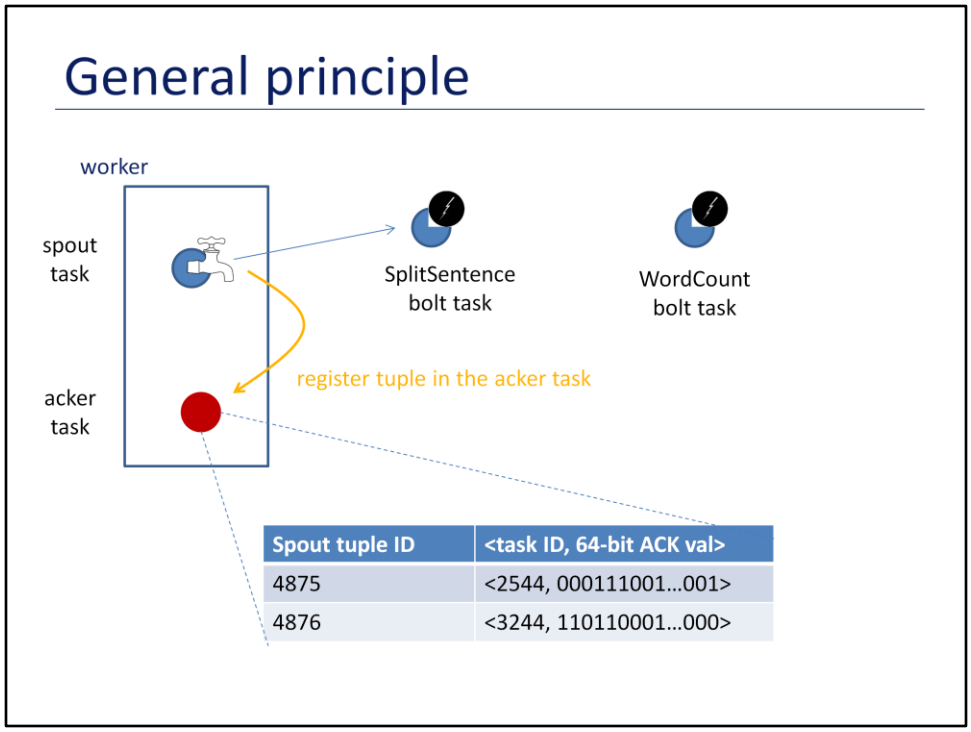
You can also see that the Storm model requires no logic around where to send tuples or how to serialize tuples. That is all handled underneath the Storm abstractions.



Worker nodes may go down during the processing. Storm guarantees that each tuple coming from a spout is processed *at least* once. To understand the mechanism, let's take a look at what the processing of a tuple looks like in the word-count topology. When a sentence tuple is generated by the spout, it's sent to whatever bolts subscribe to that spout. In this case, the word-splitter bolt creates six new tuples based on that spout tuple. Those word tuples go on to the word-count bolt, which creates a single tuple for every one of those word tuples. You can visualize all the tuples created during the processing of a single spout tuple as a directed acyclic graph (DAG). Let's call this the *tuple DAG* (shown in the slide). You could imagine much larger tuple DAGs for more involved topologies.

Storm uses an efficient and scalable algorithm for tracking tuple DAGs and retrying tuples from the spout if there's a failure somewhere downstream. Retrying tuples from the spout will cause the entire tuple DAG to be regenerated. Retrying from the spout may seem a step backward, since one failure somewhere deep in the DAG means redoing all the calculations, even those intermediate stages that had completed successfully. But upon further inspection, this model is no different than the queues-and-workers model. With queues and workers, a stage could succeed in processing, fail right before acknowledging the message and letting it be removed from the queue, and then be tried again. In both scenarios, the processing guarantee is still an at-least-once guarantee.

There's two things you have to do as a user to benefit from Storm's reliability capabilities. First, you need to tell Storm whenever you're creating a new link in the tree of tuples. Second, you need to tell Storm when you have finished processing an individual tuple. By doing both these things, Storm can detect when the tree of tuples is fully processed and can ack or fail the spout tuple appropriately. Storm's API provides a concise way of doing both of these tasks. Specifying a link in the tuple tree is called *anchoring*, and each bolt in the DAG will acknowledge the processing of the tuple.



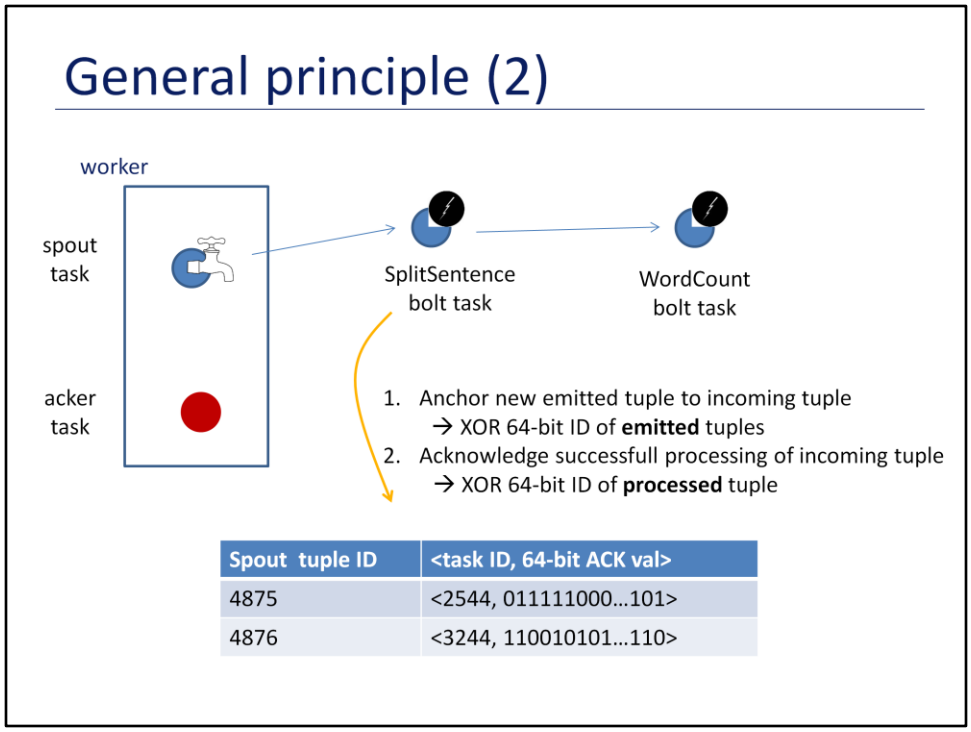
A Storm topology has a set of special "acker" tasks that track the DAG of tuples for every spout tuple. When an acker sees that a DAG is complete, it sends a message to the spout task that created the spout tuple to ack the message. Storm defaults the number of “acker tasks” to be equal to the number of workers configured in the topology.

When a tuple is created in a topology (be it in a spout or in a bolt), it is given a random 64 bit id. These ids are used by ackers to track the tuple DAG for every spout tuple.

When a spout task emits a new tuple, it simply sends a message to the appropriate acker telling it that its task id is responsible for that spout tuple. Then when an acker sees a tree has been completed, it knows to which task id to send the completion message. Acker tasks do not track the tree of tuples explicitly. For large tuple trees with tens of thousands of nodes (or more), tracking all the tuple trees could overwhelm the memory used by the ackers. Instead, the ackers take a different strategy that only requires a fixed amount of space per spout tuple (about 20 bytes). This tracking algorithm is the key to how Storm works and is one of its major breakthroughs.

An acker task stores a map from a spout tuple id to a pair of values. When emitting a tuple, the Spout provides a (self-chosen) "message id" that allows it to identify this

tuple. The first value in the map is the task id that created the spout tuple which is used later on to send completion messages. The second value is a 64 bit number called the "ack val". It is initialized to the random 64-bit ID that was given to the tuple by Storm.



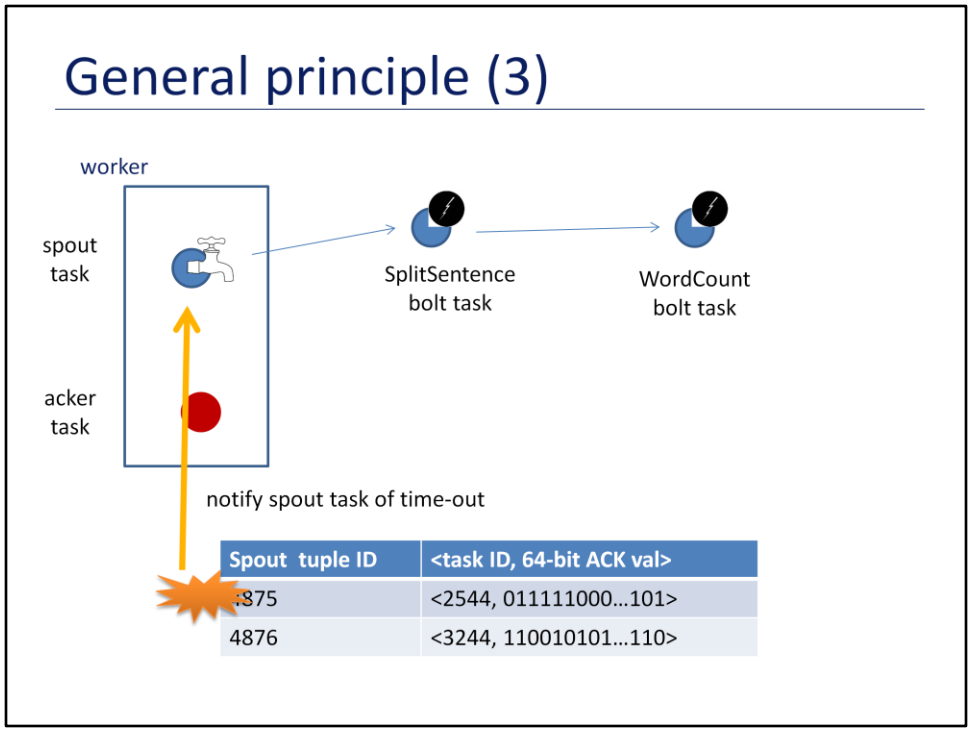
The tuples emitted by the spout are received by an instance (task) of the SplitSentence bolt. This bolt will process the incoming tuple (containing a complete sentence) and emit a number of new tuples (one per word). Storms automatically assigns a new random 64-bit ID to each of these tuple IDs. These newly emitted tuples represent new links in the tuple DAG that originates in the sentence tuple.

First, the bolt must tell Storm that it has created new links in the tuple DAG. This is called *anchoring* and should be done each time you emit a new tuple. Anchoring means that you contact the appropriate acker and XOR the 64-bit of all the **newly emitted tuples** in the 64-bit ACK val.
Second, the bolt must tell Storm that it has successfully processed an individual tuple. This is called *acking*. The 64-bit ID of the acked tuple is XORed with the 64-bit ACK val of the original Spout message ID.

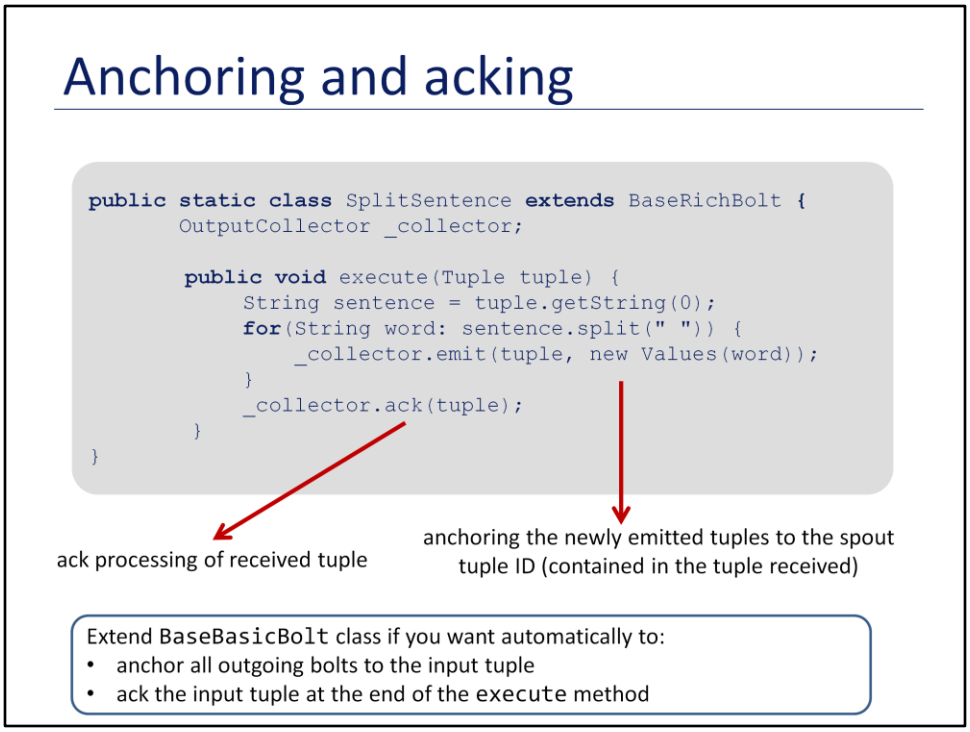
A similar acking is done by the WordCount task that receives the word tuples. If this WordCount bolt task is the last one in the topology, it will not emit new tuples and only acknowledge the received tuples. This means that the 64-bit ACK value will become zero.

The ack val is thus a representation of the state of the entire tuple tree, no matter how big or how small. When an acker task sees that an "ack val" has become 0, then it knows that the tuple tree is completed. Since tuple ids are random 64 bit numbers,

the chances of an "ack val" accidentally becoming 0 is extremely small and even then, it will only cause data loss if that tuple happens to fail in the topology.



Upon time-out of a non-zero 64-bit ACK value, the acker task will notify the spout task that registered this tuple that the tuple was not successfully processed along the entire topology. The spout task will then re-emit this tuple.



This bolt splits a tuple containing a sentence into a tuple for each word. Each word tuple is *anchored* by specifying the input tuple as the first argument to emit. Since the word tuple is anchored, the spout tuple at the root of the tree will be replayed later on if the word tuple failed to be processed downstream.

In contrast, let's look at what happens if the word tuple is emitted like this:

```
_collector.emit(new Values(word));
```

Emitting the word tuple this way causes it to be *unanchored*. If the tuple fails be processed downstream, the root tuple will not be replayed. Depending on the fault-tolerance guarantees you need in your topology, sometimes it can be appropriate to emit an unanchored tuple.

Aggregating/joining streams

incoming tuple not immediately ACK'ed

```
public static class MultiAnchorer extends BaseRichBolt {
  OutputCollector _collector;
  List<Tuple> _buffer = new ArrayList<Tuple>();
  int _sum = 0;

  public void execute(Tuple tuple) {
    _sum += tuple.getInteger(0);
    if(_buffer.size() < 100) {
        _buffer.add(tuple);
    }
    else {
        _collector.emit(_buffer, new Values(_sum));
        for(Tuple _tuple : buffer)
             _collector.ack(_tuple);
         _buffer.clear();
         _sum = 0;
     }
}
```

Emit tuple with sum, anchored to all tuples in the buffer

Ack all tuples in the buffer

This is an example of advanced anchoring/acking. This bolt emits the sum of 100 tuples received. Once 100 values have been summed, one tuple containing the sum is emitted. The first argument to the emit method is now a list of tuples, and the newly emitted tuple is anchored to the DAG tree of *all* tuples in this list. Subsequently, we individually ack all tuples in the buffer, since they have been sucessfully consumed.

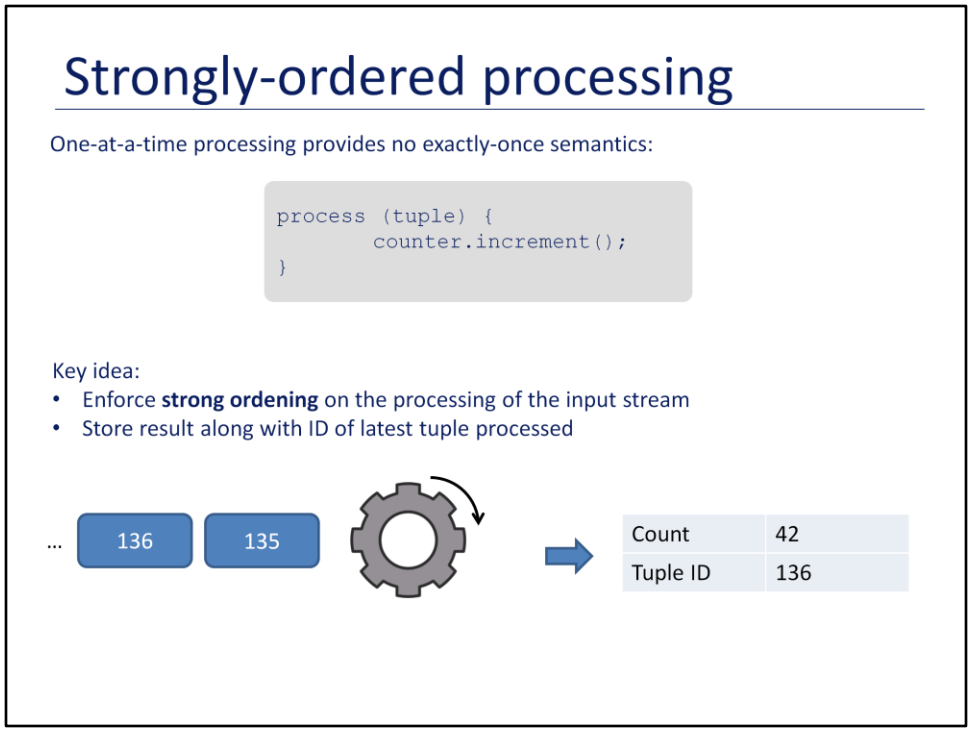
Note that this bolt is implemented as a subclass of the BaseRichBolt class, requiring you to explicitly handle anchoring and acking of tuples. It is a very common pattern for bolts to anchor all outgoing tuples to the input tuple, and then ack that tuple at the end. To automate this behavior, Storm provides a BaseBasicBolt class that takes care of this style of anchoring/acking. This style was illustrated in the source code of the WordCount bolt presented a few slides earlier.

Exactly-one semantics

- one-at-a-time stream processing
  - very low latency
  - simple
  - at-least-once processing guarantee during failure
    - inaccuracy (e.g. counting) sometimes unacceptable
- micro-batch stream processing
  - full accuracy all of the time
  - at the cost of higher latency

One-at-a-time stream processing is very low latency and simple to understand. But it can only provide an at-least-once processing guarantee during failures. Although this doesn't affect accuracy for *idempotent* operations, like adding elements to a set, it does affect accuracy for other operations such as counting.

Sometimes, this level accuracy is not sufficient. In those cases, micro-batch processing can give you the fault-tolerant accuracy you need, at the cost of higher latency in the order of hundreds of milliseconds to seconds.



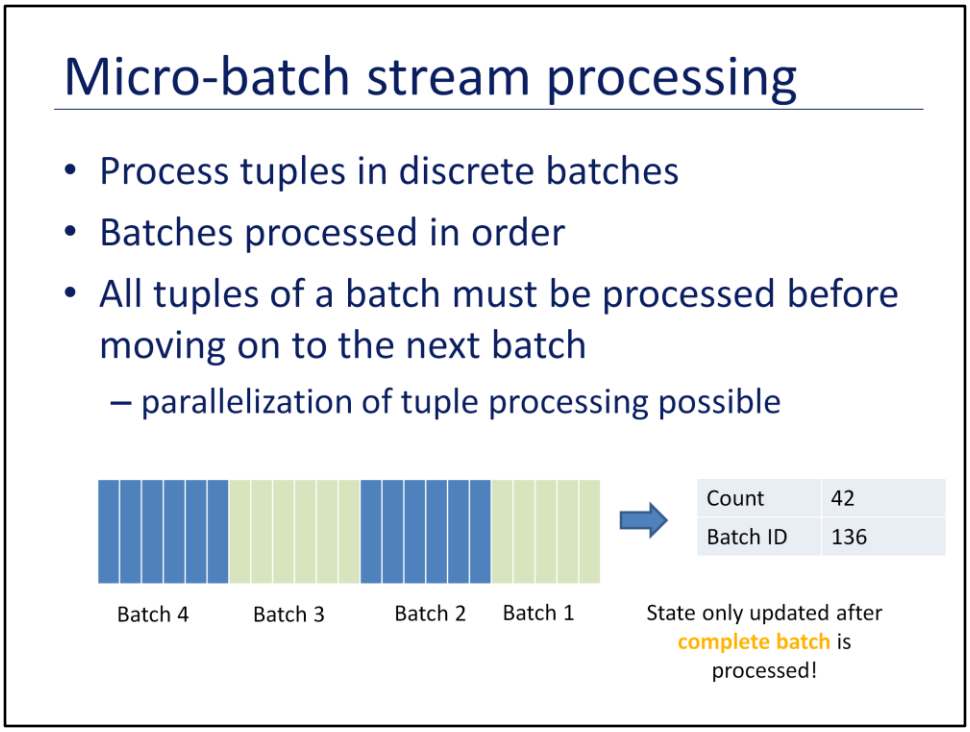
With one-at-a-time stream processing, tuples are processed independently of each other. Failures are tracked at an individual tuple level, and replays also happen at an individual tuple level. The one-at-a-time stream processing is very low latency and simple, but it can only provide an at-least-once processing guarantee during failures. Although this doesn't affect accuracy for certain operations, like adding elements to a set, it does affect accuracy for other operations such as counting.
In the one-at-a-time code on the slide, tuples will be replayed after failure but when it comes time to increment the count, you have no idea if that tuple was processed already or not. It is possible you incremented the count but then crashed immediately before acking the tuple. The only way to know is to store the ID of every tuple you have processed – but that's not a very viable solution.

The key to achieving exactly-once semantics is to enforce a strong ordering on the processing of the input stream. Assume you only process one tuple at a time and you don't move on to the next tuple until the current one is successfully processed through the entire topology. In addition, we assume that every tuple has a unique ID associated that is always the same no matter how many times it is replayed. The key idea is rather than just store the count, you store the count along with the ID of the latest tuple processed. When you are update the count, you first check the stored ID:

- The stored ID is the same as the current tuple ID. In this case, you know that the count already reflects the current tuple, so you do nothing
- The stored ID is different from the current tuple ID. In this case, you increment the

counter and update the stored ID. This works because tuples are processed in order, and the count and ID are updated atomically.

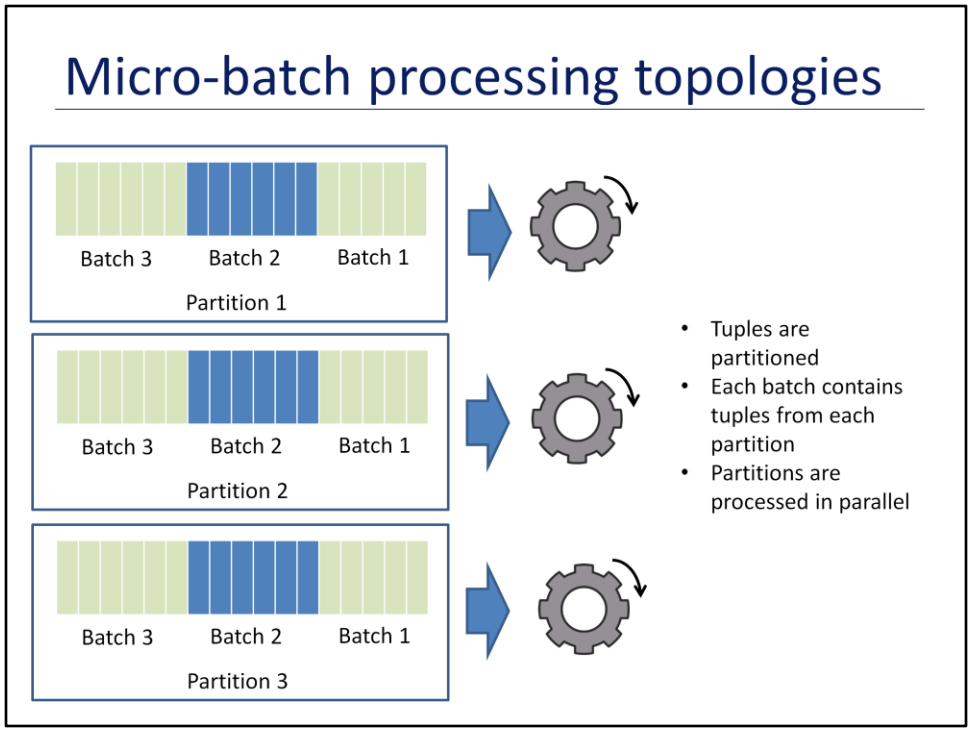
This update strategy is resilient to all failure scenarios. If the processing fails after updating the count, then the tuple will be replayed and the update will be skipped the second time around. If the processing fails before updating the count, then the update will occur the second time around.



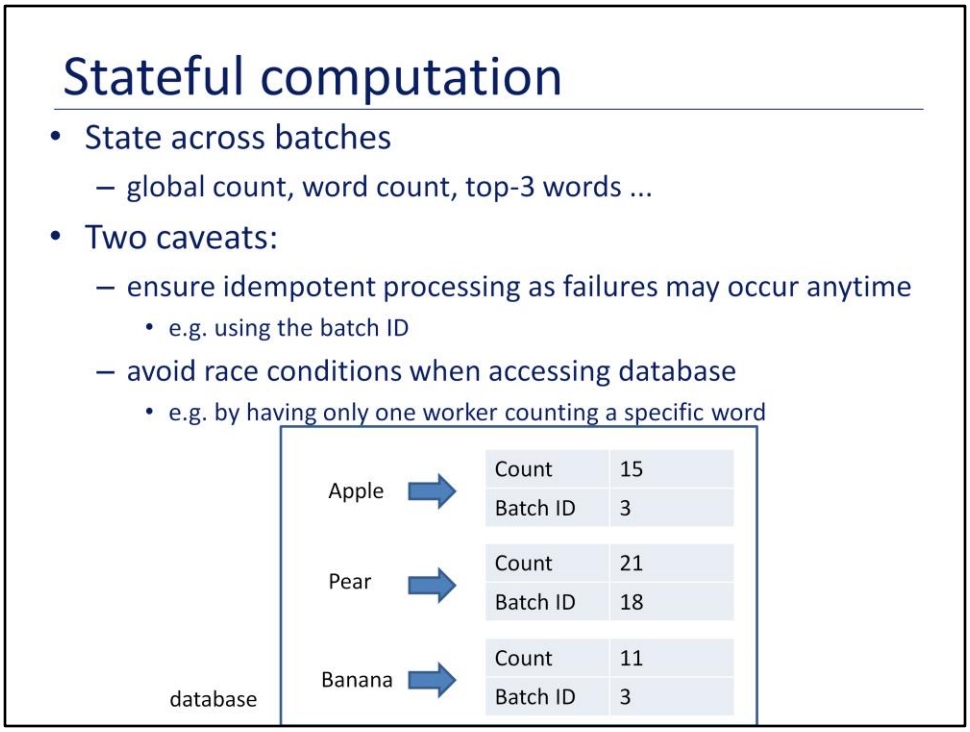
Processing one tuple at a time is of course highly inefficient. A better approach is to process the tuples as discrete batches. This is known as micro-batch stream processing.

The batches are processed in order, and each batch has a unique ID that is always the same on every replay. Because many tuples are processed per iteration rather than just one, the processing can be made scalable by parallelizing it. Batches must be processed to completion before moving on to the next batch.

Retaking our global counting example, we now store the count along with the latest batch ID involved in updating the count. Now suppose that after the state in the database is updated, something fails in the stream processor and the message that the batch was finished is never received. The stream processor will timeout the batch and retry the batch. When it comes time to update the database, it sees that the state has already been updated to include that batch. So rather than increment the count, it does nothing and moves on to the next batch.



Just like how MapReduce and one-at-a-time stream processing partition data and process each partition in parallel, the same is done with micro-batched processing. Processing a batch of words looks like the figure. A single batch includes tuples from all partitions in the incoming stream.
The parallel processing of the different partitions have an impact when you have to save your state. This is further discussed on the next slide.



When a failure occurs, the entire batch will be replayed. This poses no problem for batch-local computation: computation that occurs solely within a single batch. Examples are repartitioning a stream according to a given field (e.g. word) or simply counting the number of tuples in a batch.

Sometimes you need to keep state across all batches, an example is updating a global count or per-word count over all batches. This is where you have to be really careful about how you do update the state (e.g. in a database).

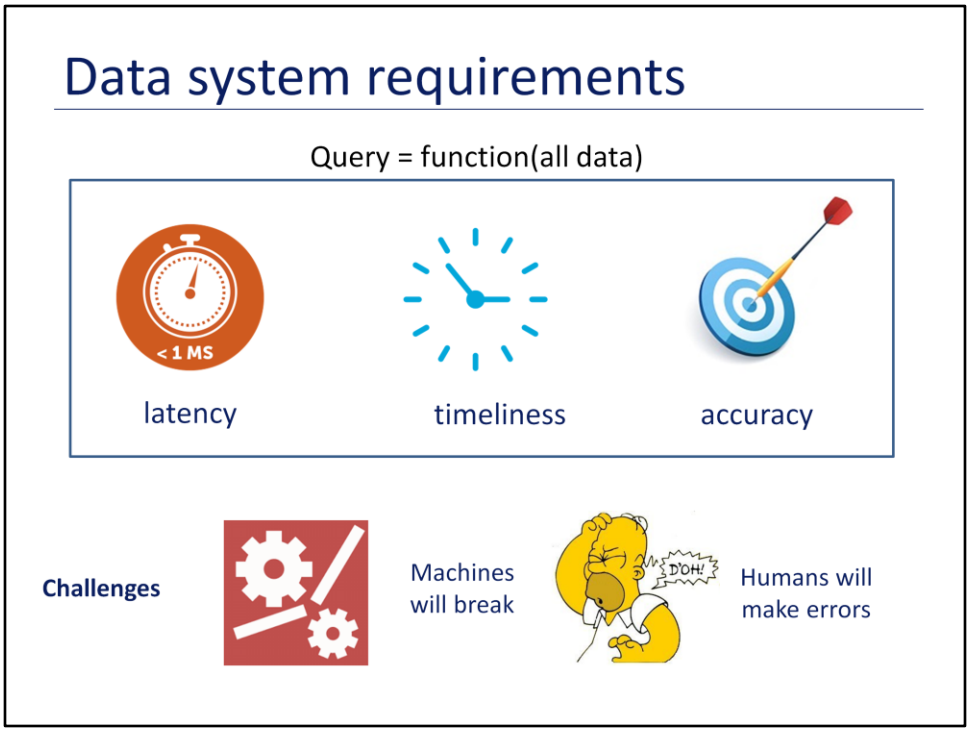
A first caveat is that you should avoid race conditions between workers when updating a particular field in the database. In the canonical word counting example, this means that you must first repartition the tuples by the field containg the word itself, so that only one worker will update the state for that specific word.

A second caveat is that you should ensure that all state updates are idempotent. The trick of storing the batch ID with the state is a particular way of achieving this.
Let's consider a failure scenario. Suppose a machine dies in the cluster while a batch is being processed, and only some partitions succeeded in updating the database. Some words will have counts reflecting reflecting the current batch, and others won't be updated yet. When the batch is replayed, the words that have state including the current batch ID won't be updated, whereas the words that haven't been updated yet will be updated like norma.

Further Reading

- N. Garg, Learning Apache Kafka (2nd edition)
- S. Saxena, Real-time Analytics with Storm and Cassandra

BIG DATA ARCHITECTURES



A data system answers questions based on data you have seen in the past. Or put more formally: a data system computes queries that are functions of all the data you have ever seen (query = function(all data)). There are a number of properties you are concerned about with your queries:

- **Latency –** The time it takes to run a query. In many cases, your latency requirements will be very low – on the order of milliseconds. Other times it is okay for a query to take a few seconds. When doing ad hoc analysis, your latency requirements are often very lax, even on the order of hours.
- **Timeliness** – How up-to-date the query results are. A completely timely query takes into account all data ever seen in the past, whereas a less timely query may not include results from the recent minutes or hours.
- **Accuracy** – In many cases, in order to make queries performant or scalable, you must make approximations in your query implementations.

A huge part of building data systems is making them fault tolerant. You have to plan for how your system will behave when you encounter machine failures. Oftentimes this means making trade-offs with the preceding properties. For example, there is a fundamental tension between latency and timeliness. The CAP theorem shows that under partitions, a system can either be consistent (queries take into account all previous written data) or available (queries are answered at the moment). Consistency is just a form of timeliness, and availability just means the latency of the query is bounded. An eventually consistent system chooses latency over timeliness

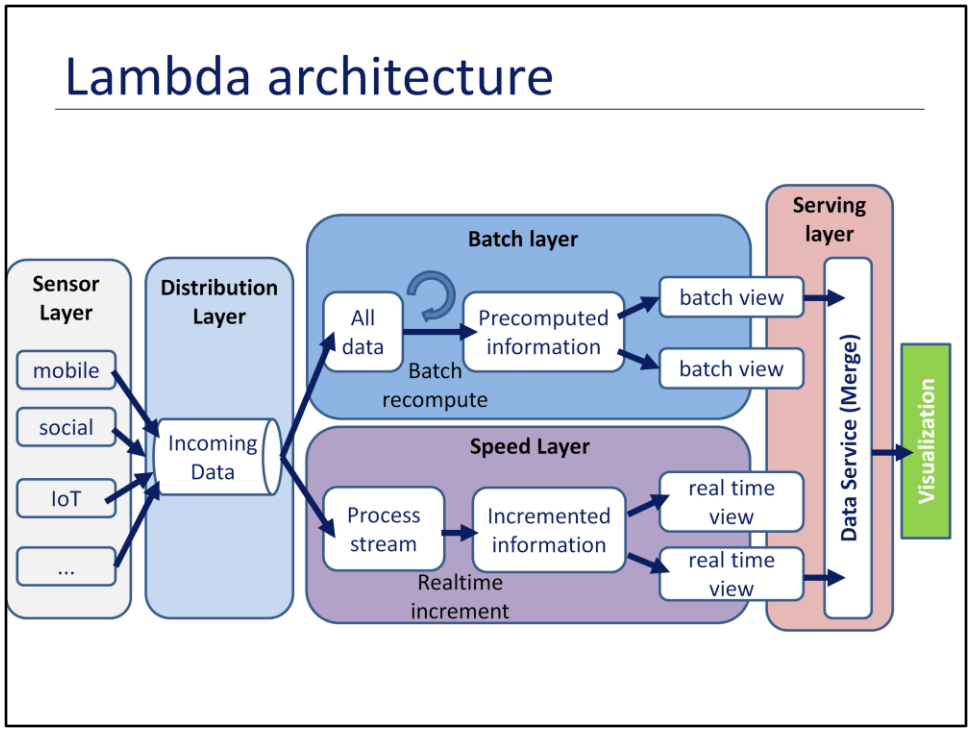
(queries are always answered, but may not take into account all prior data during failure scenarios).

Because data systems are dynamic, changing systems built by humans and with new features and analyses deployed all the time, humans are an integral part of any data system. Humans can and will fail. We saw earlier how mutability is fundamentally not human-fault tolerant. If a human can mutate data, then a mistake can mutate data. The only solution is to make your core data *immutable*, with the only write operation allowed being appending new data to your ever-growing set of data. You can set permissions on your core data to disallow deletes and updates, making your system far more robust.

This leads us to the basic model of data systems:

- A master dataset consisting of an ever-growing set of data
- Queries as functions that take in the entire master dataset as input.

The Lambda architecture is one possible implementation of this basic data model, and is discussed in the next slide.



Computing arbitrary functions on an arbitrary dataset in real time is a daunting task. There is no single tool that provides a complete solution. Instead, you have to use a variety of tools and techniques to build a complete Big Data system. The main idea of the Lambda architecture is to build Big Data systems as a series of layers. Each layer satisfies a subset of the properties and builds upon the functionality provided by the layers beneath it.

Everything starts from the *query = function(all data)* equation. Running queries on the fly would take a huge amount of resources and be unreasonably expensive. Imagine having to read a petabyte dataset every time you wanted to answer the query of someone's current location. The most obvious alternative approach is to precompute the query function, which we call batch views. The batch layer needs to able to do two things: store an immutable, constantly growing master dataset (a very large list of records) and compute arbitrary functions on that dataset. This type of processing is best done using batch-processing systems like Hadoop. Conceptually, the batch layer runs in a *while(true)* loop and continuously recomputes the batch views from scratch, since there is a continuous infeed of data from the sensor layer.

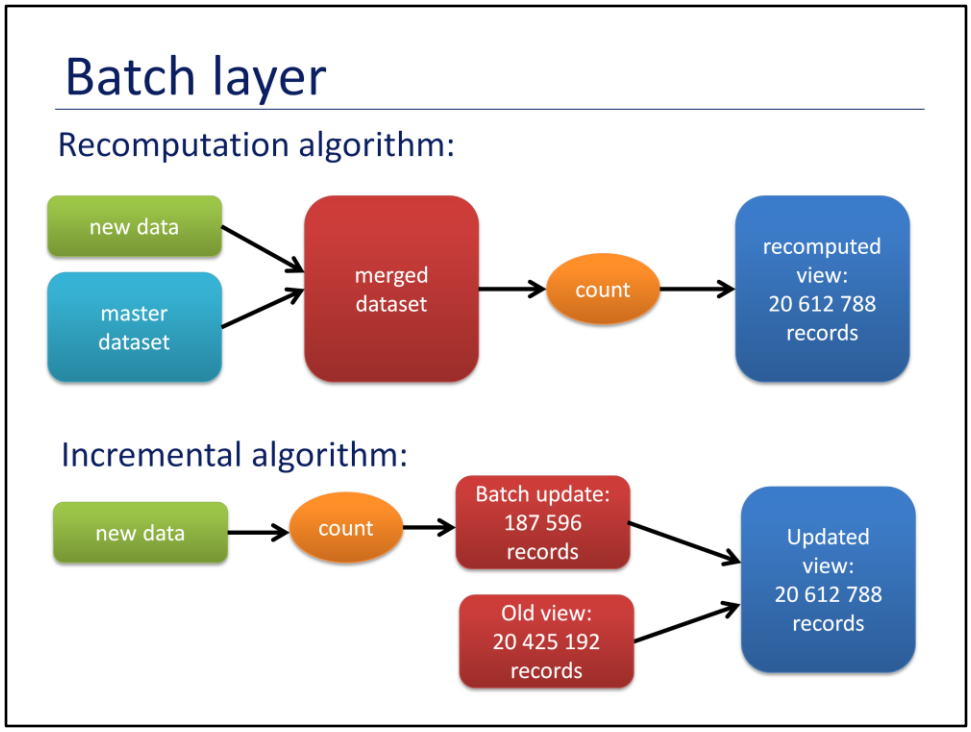
The batch layer emits batch views as the result of its functions. The next step is to load the views somewhere so that they can be queried. This is where the serving layer comes in. The serving layer is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it. When new batch views

are available, the serving layer automatically swaps those in so that more up-to-date results are available. A serving layer database supports batch updates and random reads. Most notably, it doesn't need to support random writes. This is a very important point, as random writes cause most of the complexity in databases. By not supporting random writes, these databases are extremely simple. That simplicity makes them robust, predictable, easy to configure and easy to operate. ElephantDB is one example, built from only a few thousands lines of code.

The batch and serving layers support arbitrary queries on an arbitrary dataset with the trade-off that queries will be out of date by a few hours. It takes a new piece of data a few hours to propagate through the batch layer into the serving layer where it can be queried. To have a fully real-time data system that allows to compute arbitrary functions on arbitrary data in real-time, we need to compensate for the data that came in while the batch precomputation was running. This is the purpose of the speed layer.

The speed layer is similar to the batch layer in that it produces views based on the data it receives. One big difference is that the speed layer only looks at recent data, whereas the batch layer looks at all the data at once. Another big difference is that in order to achieve the smallest latency possible, the speed layer doesn't look at all the new data at once. Instead, it updates the realtime views as it receives new data instead of recomputing the views from scratch like the batch layer does. The speed layer does incremental computation instead of the recomputation done in the batch layer.

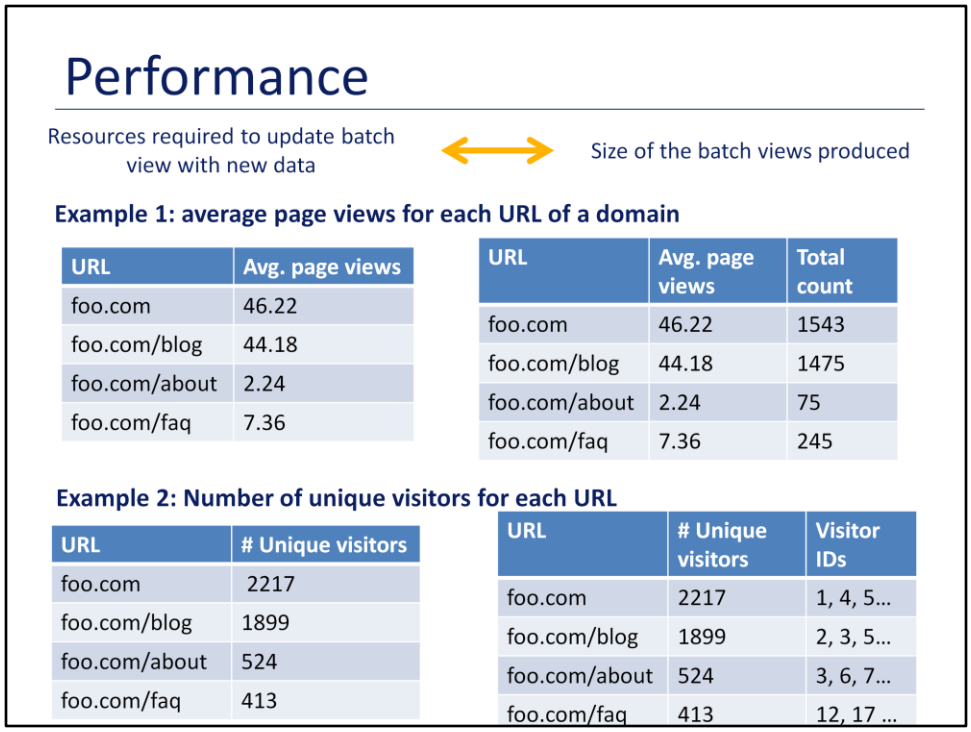
You resolve queries by looking at both the batch and realtime views and merging the results together. The speed layer uses databases that support random reads and random writes. Because these databases support random writes, they are orders of magnitude more complex than the databases you use in the serving layer, both in terms of implementation and operation.



Because your master dataset is continually growing, you must have a strategy for updating your batch views when new data becomes available. You could choose a recomputation algorithm, throwing away the old batch views and recomputing functions over the entire master dataset. Alternatively, an incremental algorithm will update the views directly when new data arrives.

As a basic example, consider a batch view containing the total number of records in your master dataset. A recomputation algorithm would update the count by first appending the new data to the master dataset and then counting all records from scratch. An incremental algorithm, on the other hand, would count the number of new data records and add it to the existing count.

You might be wondering why you would ever use a recomputation algorithm when you can use a vastly more efficient incremental algorithm instead. The key trade-offs between the two approaches are performance, human-fault tolerance and the generality of the algorithm. We'll discuss both types of algorithms in regard to each of these issues.



There are two aspects to the performance of a batch-layer algorithm: the amount of resources required to update a batch view with new data, and the size of the batch views produced. An incremental algorithm always uses significantly less resources to update a view because it uses new data and the current state of the batch view to perform an update. For a task such as computing URL pageviews over time, the view will be significantly smaller than the master dataset because of the aggregation. A recomputation algorithm looks at the entire master dataset, so the amount of resources needed for an update can be multiple orders of magnitude higher than an incremental algorithm. But the size of the batch view for an incremental algorithm can be significantly larger than the corresponding batch view for a recomputation algorithm. This is because the view needs to be formulated in such a way that it can be incrementally updated. We demonstrate this trade-off through two separate examples.

First, suppose you need to compute the average number of pageviews for each URL within a particular domain. The batch view generated by a recomputation algorithm would contain a map from each URL to its corresponding average. But this isn't suitable for an incremental algorithm, because updating the average incrementally requires that you also know the number of records used for computing the previous average. An incremental view would therefore store both the average and the total count for each URL, increasing the size of the incremental view over the recomputation-based view by a constant factor.

In other scenarios, the increase in the batch view size for an incremental algorithm is much more severe. Consider a query that computes the number of unique visitors for each URL. A recomputation view only requires a map from the URL to the unique count. In contrast, an incremental algorithm only examines the new pageviews, so its view must contain the full set of visitors for each URL so it can determine which records in the new data correspond to return visits. As such, the incremental view could potentially be as large as the master dataset.

The batch view generated by an incremental algorithm isn't always this large, but it can be far larger than the corresponding recomputation-based view.

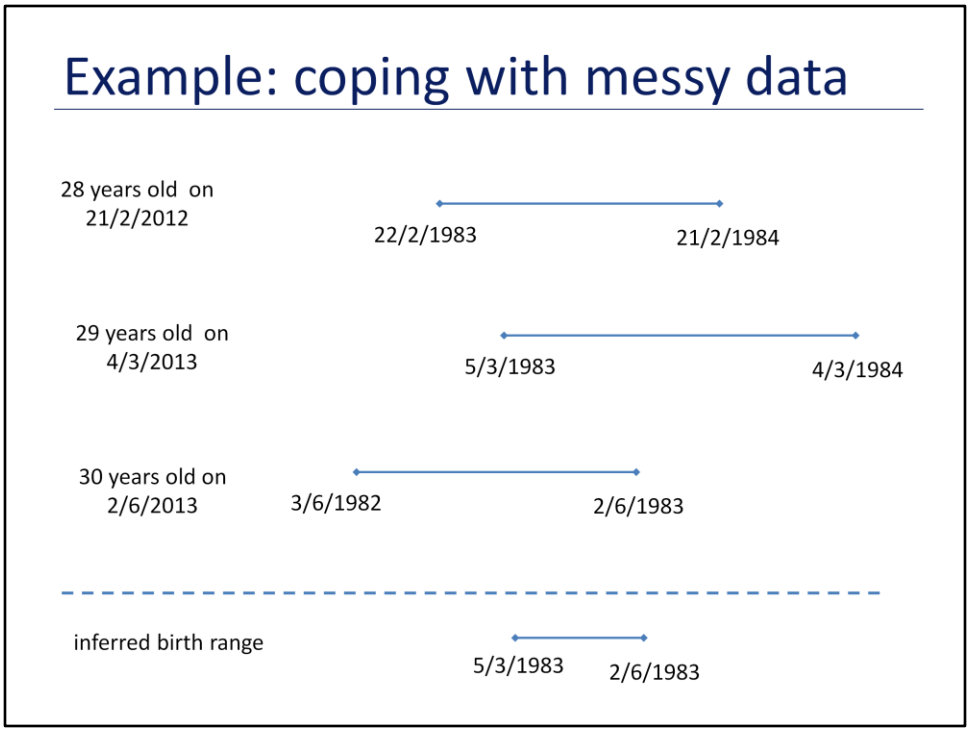
Fault tolerance and generality

- Human-fault tolerance
  - recomputation: fix algorithm and redeploy code
  - incremental: which records were affected?
- Generality of the algorithms
  - incremental algorithms must often be tailored
  - incremental algorithms shift complexity to on-the-fly computations

The other two aspects to consider when evaluating incremental vs recomputation algorithms are fault tolerance and generality of the algorithms.

Recomputation algorithms are inherently human-fault tolerant. Consider as an example a batch-layer algorithms that computes a global count of the number of records in the master dataset. Now suppose you make a mistake and deploy an algorithm that increments the global count for each record by two instead of by one. If your algorithm is recomputation-based, all that is required to fix the algorithm and redeploy the code: your batch view will be correct the next time the batch layer runs, since the algorithm recomputes the batch view from scratch. But if your algorithm is incremental, then correcting your view is not so simple. The only option is to identify the records that were overcounted, determine how many times each one was overcounted, and then correct the count for each affected record.
Although incremental algorithms can be faster to run, they must often be tailored to address the problem at hand. Incremental algorithms also shift complexity to on-the-fly computations, which increases their latency. Suppose you have improved a normalization component. You no longer only map “Ghent”, “Zwijnaarde”, “Ledeberg” to “Ghent, Belgium” but now you make a distinction between these city regions. In an incremental algorithm, this means that your batch view has to keep every name that was ever mapped to “Ghent, Belgium” to apply this update. Moreover, you will have to renormalize each city name every time a query is performed. This increases the latency of the on-the-fly component and could very

well take too long for your application’s requirements.

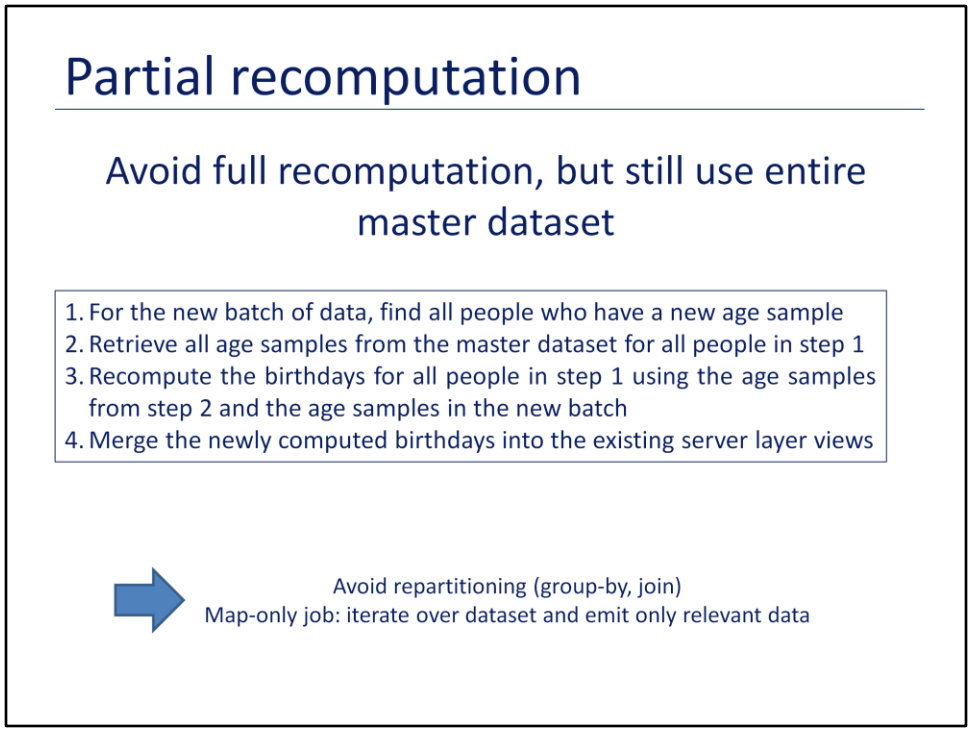


Let us consider another example where the choice between incremental and recomputation algorithms is more difficult: the "birthday inference" problem. Imagine you are writing a web crawler that collects people's ages from their public profiles. The profile does not contain a birthday, but only what the person's age is at the moment you crawled that web page. Given this raw data of [age, timestamp] pairs, your goal is to deduce the birthday of each person.

The idea of the algorithm is illustrated in the slide. Imagine you crawl the profile of Tom on January 4, 2012 and see his age is 23. Then you crawl his profile again on January 11, 2012 and see his age is 24. You can deduce that his birthday happened sometime between those datet. Likewise, if you crawl the profile of Jill on October 20, 2013 and see she is 43, and then crawl it again on November 4, 2013 and see she is still 43, you know her birthday is not between those dates. The more age samples you have, the better you can infer that someone's birthday is within a small range of dates.

In the real world, data can get messy. Someone may have incorrectly entered their birthday and then changed it a later date. This may cause your age inference algorithm to fail because every day of the year has been eliminated as possible birthday. You might modify your algorithm to search for the smallest number of age samples it can ignore to produce the smallest range of possible birthdays. The algorithm might prefer to use recent age samples over older age samples.

If you implement your birthday-inference batch layer using recomputation, it's easy. Your algorithm can look at all age samples for a person at once and do everything necessary to deal with messy data and emit a single range of dates as output. But incrementalizing the algorithm is much trickier: it is hard to see how you can deal with the messy data problem without having access to the full range of age samples.



Partial recomputation is an alternative that blurs the line between incrementalization and recomputation and gets you the best of both worlds. In the example of the previous slide: if a person has no new age samples since the last time the batch layer ran, then the inferred birthday for that person will not change at all. The idea is to do the following:

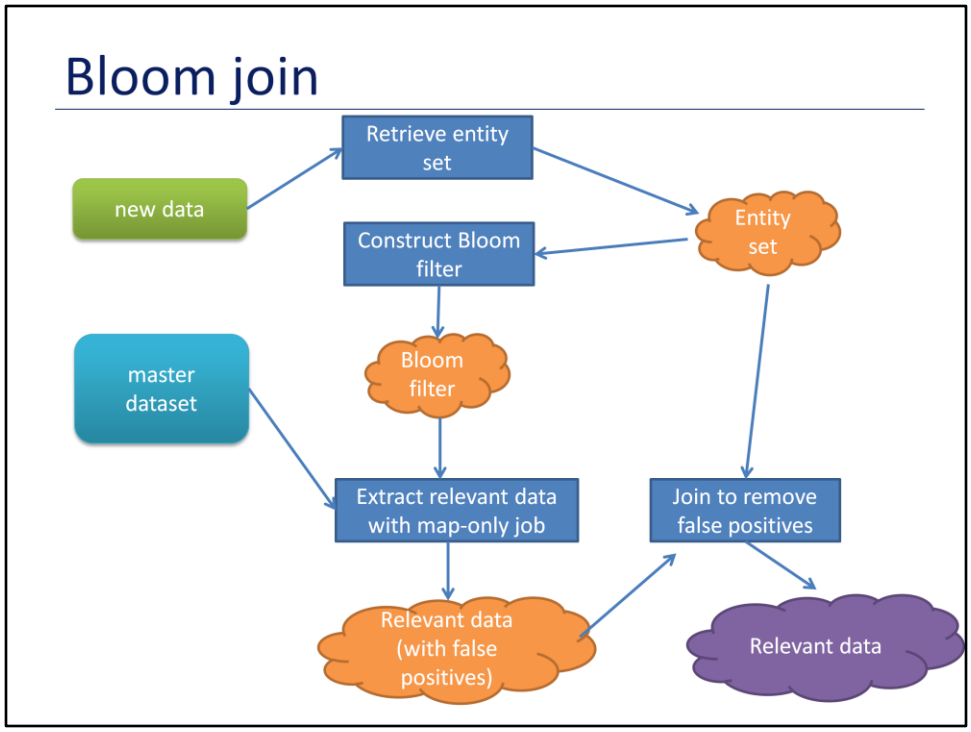
1. For the new batch of data, find all people who have a new age sample
2. Retrieve all age samples from the master dataset for all people in step 1
3. Recompute the birthdays for all people in step 1 using the age samples from step 2 and the age samples in the new batch
4. Merge the newly computed birthdays into the existing server layer views

This alternative implementation is not fully incremental because it still makes use of the master dataset. But it avoids most of the cost of a full recomputation by ignoring anyone who hasn't changed in the latest set of data.

The key idea to partial recomputation is to retrieve all the relevant data for the entities that changes, run a normal recompute algorithm on the retrieved data plus the new data, and then merge those results into the existing views. The nice thing about partial recomputes is that they can be implemented very efficiently. The most expensive step – looking over the entire master dataset to find relevant data – can be done relatively cheaply.

The key to making it efficient is to avoid having to repartition the entire master

dataset, as this is the most expensive part of batch algorithms. For example, repartitioning happens whenever you do a group-by operation or a join. Partitioning involves serialization/deserialization, network transfer and possibly buffering on disk. In contrast, operations that don't require partitioning can quickly scan through the data and operate on each piece of data as it's seen. Retrieving relevant data for a partial recompute can be done using the latter method.



The first step to retrieving relevant data is to construct a set of all the entities for which you need relevant data. You then scan over the entire master dataset and only emit data for those entities that exist in the set (each task would have a copy of that set). In a batch-processing system like Hadoop, this would correspond to a map-only job (with a trivial reducer).

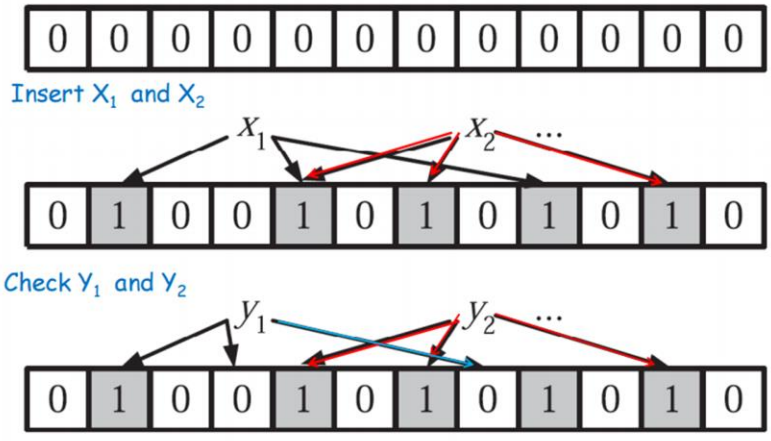
You are limited by memory, so your set can only be so big. But a data structure called a *Bloom filter* can make this work for much larger sets of entities. A Bloom filter is a compact data structure that represents a set of elements and allows you to ask if it contains an element. A Bloom filter is much more compact than a set, but as a trade-off, query operations on it are probabilistic. A Bloom filter will sometimes incorrectly tell you that an element exists in the set, but it will never tell you an element that was added to the set is not in the set. So a Bloom filter has false positives but no false negatives.

If you use a Bloom filter to retrieve relevant data from the master dataset, you will filter out the vast majority of the master dataset. Due to the false posivites, though, some data will be emitted that you didn't want to retrieve. You can then do a join between the retrieved data and the list of desired entities to filter out the false positives. A join requires a partitioning, but because the vast majority of the master dataset was already filtered out, getting rid of the false positives is not an expensive operation.

Bloom filter

- array of **m** bits representing a set S = {x_1, x_2 ... x_n} of **n** elements
  - initialized to 0
- **k** independent hash functions h_1 ... h_k with range {1, 2 ... m}
  - assume each hash function maps each item in the universe to a random number *uniformly* over the range {1... m}
- for each element x in S, the bit $h_i(x)$ in the array is set to 1, for 1 <= i <= k
  - a bit in the array may be set to 1 multiple times for different elements

Bloom filter example



Standard Bloom filter (cont.)

- To check membership of y in S, check whether $h_i(y)$, $1 <= i <= k$ are all set to 1
  - If not: y is definitely not in S
  - Else, we conclude that y is in S, but sometimes this conclusion is wrong (false positive)
- For many applications, false positives are acceptable as long as the probability of a false positive is small enough

Dimensioning the filter

- Optimal number of hash functions *k*

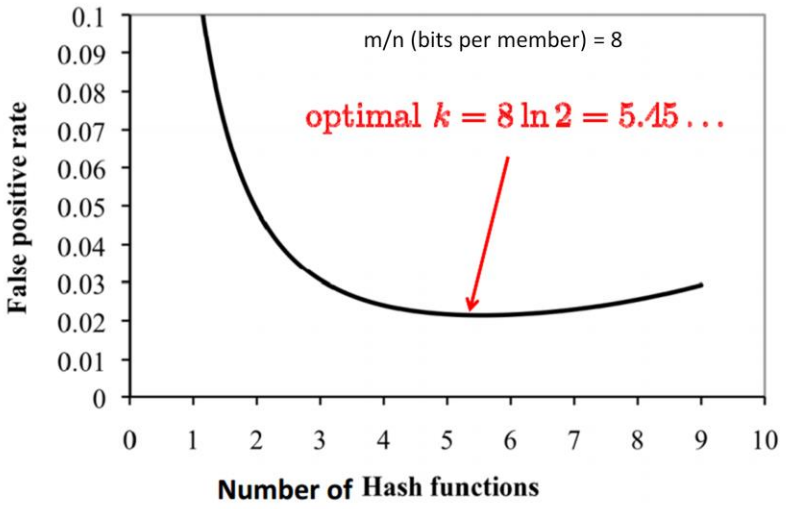
$$k = \frac{m}{n} \ln 2$$

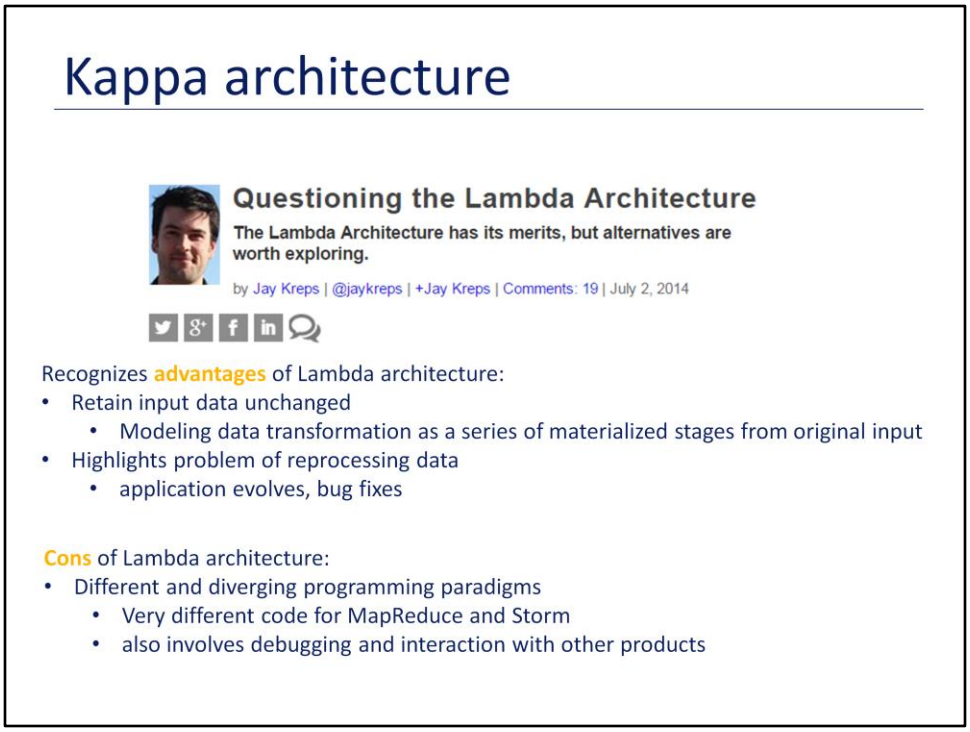
- Required number of bits *m*, given
  - *n* (number of inserted elements)
  - desired false positive probability *p*
  - the optimal value of *k* (see above) is used

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

proportional to size of input set n!

False positive rate vs. *k*



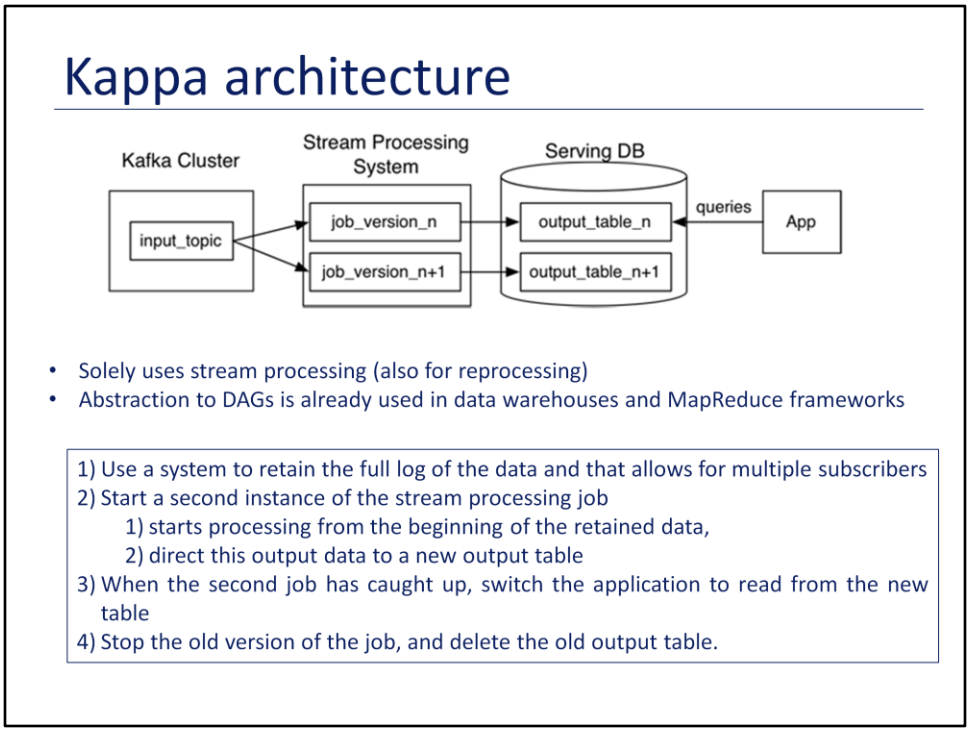


In July 2014, Jay Kreps coined the term "Kappa Architecture" in a blog post (http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html). At that time, Jay Kreps was working on the big data systems at LinkedIn.

In his blog post, he first acknowledges some merits of the Lambda architecture:

- It highlights the importance of keeping the input data unchanged. Jay Kreps also believes that modeling data transformation as a series of materialized stages from an original input has a lot of merit.
- It highlights the problem of reprocessing data: processing input data over again to re-derive output. Reprocessing is needed because applications evolve (e.g. you want to compute new output fields) or bugs have to be fixed

But he also refers to some problems he sees with the Lambda architecture. Programming in distributed frameworks like Storm and Hadoop is complex. Inevitably, code ends up being specifically engineerd toward the framework it runs on. In the Lambda architecture, you need to code in both (types of) frameworks. In addition, the operational burden of running and debugging two systems is going to be very high.



The crux of the Kappa architecture is that it uses stream processing systems to handle the reprocessing when code changes. The argument is that stream processing systems already have a notion of parallelism; so why not just handle reprocessing by increasing the parallelism and replaying history very, very fast.

Many people have a notion that stream processing is inherently something that computes results off some ephemeral streams and then throws all the underlying data away. But there is no reason this should be true. The fundamental abstraction in stream processing is data flow DAGs (directed acyclic graphs), which are exactly the same underlying abstraction in traditional data warehouse (a la Volcano) as well as being the fundamental abstraction in the MapReduce successor Tez. Stream processing is just a generalization of this data-flow model that exposes checkpointing of intermediate results and continual output to the user.
So, how can the reprocessing be done directly from the stream processing job?

1) Use Kafka (or some other system) to retain the full log of the data you want to be able to reprocess and that allows for multiple subscribers
2) When you want to do the reprocessing, start a second instance of your stream processing job that starts processing from the beginning of the retained data, but direct this output data to a new output table
3) When the second job has caught up, switch the application to read from the new table
4) Stop the old version of the job, and delete the old output table.

Polyglot persistence/processing

The term polyglot is borrowed and redefined for big data as a set of applications that use several core database technologies, and this is the most likely outcome of your implementation planning. The official definition of*polyglot* is “someone who speaks or writes several languages.” It is going to be difficult to choose one persistence style no matter how narrow your approach to big data might be.

Further reading

- S. Lam, Bloom Filters [online - 1]
- G. Schmutz, Big Data and Fast Data – Lambda Architecture in Action [online - 2]
- J. Kreps, Questioning the Lambda Architecture [online - 3]
- N. Marz and J. Warren, Big Data [book]

[1] http://www.cs.utexas.edu/users/lam/386p/slides/Bloom%20Filters.pdf
[2] http://www.slideshare.net/gschmutz/big-data-and-fast-data-lambda-architecture-in-action?related=1
[3] http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html