

UNIT II NOSQL DATA MANAGEMENT

Introduction to NoSQL - aggregate data models - aggregates - key-value and document data models - relationships - graph databases - schemaless databases - materialized views - distribution models - sharding - master-slave replication - peer-peer replication - sharding and replication - consistency - relaxing consistency - version stamps - map-reduce - partitioning and combining - composing map-reduce calculations

NOSQL DATA MANAGEMENT

What is Nosql?

NoSQL database, also called **Not Only SQL**, is an approach to data management and database design that's useful for very large sets of distributed data. NoSQL is a whole new way of thinking about a database. NoSQL is not a relational database. The reality is that a relational database model may not be the best solution for all situations. The easiest way to think of NoSQL, is that of a database which does not adhering to the traditional relational database management system (RDMS) structure. Sometimes you will also see it referred to as 'not only SQL'. the most popular NoSQL database is Apache Cassandra. Cassandra, which was once Facebook's proprietary database, was released as open source in 2008. Other NoSQL implementations include SimpleDB, Google BigTable, Apache Hadoop, MapReduce, MemcacheDB, and Voldemort. Companies that use NoSQL include Netflix, LinkedIn and Twitter.

Why Are NoSQL Databases Interesting? / Why we should use Nosql? / when to use Nosql?

There are several reasons why people consider using a NoSQL database.

Application development productivity. A lot of application development effort is spent on mapping data between in-memory data structures and a relational database. A NoSQL database may provide a data model that better fits the application's needs, thus simplifying that interaction and resulting in less code to write, debug, and evolve.

Large data. Organizations are finding it valuable to capture more data and process it more quickly. They are finding it expensive, if even possible, to do so with relational databases. The primary reason is that a relational database is designed to run on a single machine, but it is usually more economic to run large data and computing loads on clusters of many smaller and cheaper machines. Many NoSQL databases are designed explicitly to run on clusters, so they make a better fit for big data scenarios.

Analytics. One reason to consider adding a NoSQL database to your corporate infrastructure is that many NoSQL databases are well suited to performing analytical queries.

Scalability. NoSQL databases are designed to scale; it's one of the primary reasons that people choose a NoSQL database. Typically, with a relational database like SQL Server or Oracle, you scale by purchasing larger and faster servers and storage or by employing specialists to provide additional tuning. Unlike relational databases, NoSQL databases are designed to easily scale out as they grow. Data is partitioned and balanced across multiple nodes in a cluster, and aggregate queries are distributed by default.

Massive write performance. This is probably the canonical usage based on Google's influence. High volume. Facebook needs to store 135 billion messages a month. Twitter, for example, has the problem of storing 7 TB/data per day with the prospect of this requirement doubling multiple times per year. This is the data is too big to fit on one node problem. At 80 MB/s it takes a day to store 7TB so writes need to be distributed over a cluster, which implies key-value access, MapReduce, replication, fault tolerance, consistency issues, and all the rest. For faster writes in-memory systems can be used.

Fast key-value access. This is probably the second most cited virtue of NoSQL in the general mind set. When latency is important it's hard to beat hashing on a key and reading the value directly from memory or in as little as one disk seek. Not every NoSQL product is about fast access, some are more about reliability, for example. but what people have wanted for a long time was a better memcached and many NoSQL systems offer that.

Flexible data model and flexible datatypes. NoSQL products support a whole range of new data types, and this is a major area of innovation in NoSQL. We have: column-oriented, graph, advanced data structures, document-oriented, and key-value. Complex objects can be easily stored without a lot of mapping. Developers love avoiding complex schemas and ORM frameworks. Lack of structure allows for much more flexibility. We also have program and programmer friendly compatible datatypes likes JSON.

Schema migration. Schemalessness makes it easier to deal with schema migrations without so much worrying. Schemas are in a sense dynamic, because they are imposed

by the application at run-time, so different parts of an application can have a different view of the schema.

Write availability. Do your writes need to succeed no matter what? Then we can get into partitioning, CAP, eventual consistency and all that jazz.

Easier maintainability, administration and operations. This is very product specific, but many NoSQL vendors are trying to gain adoption by making it easy for developers to adopt them. They are spending a lot of effort on ease of use, minimal administration, and automated operations. This can lead to lower operations costs as special code doesn't have to be written to scale a system that was never intended to be used that way.

No single point of failure. Not every product is delivering on this, but we are seeing a definite convergence on relatively easy to configure and manage high availability with automatic load balancing and cluster sizing. A perfect cloud partner.

Generally available parallel computing. We are seeing MapReduce baked into products, which makes parallel computing something that will be a normal part of development in the future.

Programmer ease of use. Accessing your data should be easy. While the relational model is intuitive for end users, like accountants, it's not very intuitive for developers. Programmers grok keys, values, JSON, Javascript stored procedures, HTTP, and so on. NoSQL is for programmers. This is a developer led coup. The response to a database problem can't always be to hire a really knowledgeable DBA, get your schema right, denormalize a little, etc., programmers would prefer a system that they can make work for themselves. It shouldn't be so hard to make a product perform. Money is part of the issue. If it costs a lot to scale a product then won't you go with the cheaper product, that you control, that's easier to use, and that's easier to scale?

Use the right data model for the right problem. Different data models are used to solve different problems. Much effort has been put into, for example, wedging graph operations into a relational model, but it doesn't work. Isn't it better to solve a graph problem in a graph database? We are now seeing a general strategy of trying find the best fit between a problem and solution.

Distributed systems and cloud computing support. Not everyone is worried about scale or performance over and above that which can be achieved by non-NoSQL systems. What they need is a distributed system that can span datacenters while handling failure scenarios without a hiccup. NoSQL systems, because they have focussed on scale, tend to exploit partitions, tend not use heavy strict consistency protocols, and so are well positioned to operate in distributed scenarios.

Difference between Sql and Nosql

- SQL databases are primarily called as Relational Databases (RDBMS); whereas NoSQL database are primarily called as non-relational or distributed database.
- SQL databases are table based databases whereas NoSQL databases are document based, key-value pairs, graph databases or wide-column stores. This means that SQL databases represent data in form of tables which consists of n number of rows of data whereas NoSQL databases are the collection of key-value pair, documents, graph databases or wide-column stores which do not have standard schema definitions which it needs to adhered to.
- SQL databases have predefined schema whereas NoSQL databases have dynamic schema for unstructured data.
- SQL databases are vertically scalable whereas the NoSQL databases are horizontally scalable. SQL databases are scaled by increasing the horse-power of the hardware. NoSQL databases are scaled by increasing the databases servers in the pool of resources to reduce the load.
- SQL databases uses SQL (structured query language) for defining and manipulating the data, which is very powerful. In NoSQL database, queries are focused on collection of documents. Sometimes it is also called as UnQL (Unstructured Query Language). The syntax of using UnQL varies from database to database.
- SQL database examples: MySql, Oracle, Sqlite, Postgres and MS-SQL. NoSQL database examples: MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4j and CouchDb
- For complex queries: SQL databases are good fit for the complex query intensive environment whereas NoSQL databases are not good fit for complex queries. On a high-level, NoSQL don't have standard interfaces to perform complex queries, and the queries themselves in NoSQL are not as powerful as SQL query language.
- For the type of data to be stored: SQL databases are not best fit for hierarchical data storage. But, NoSQL database fits better for the hierarchical data storage as

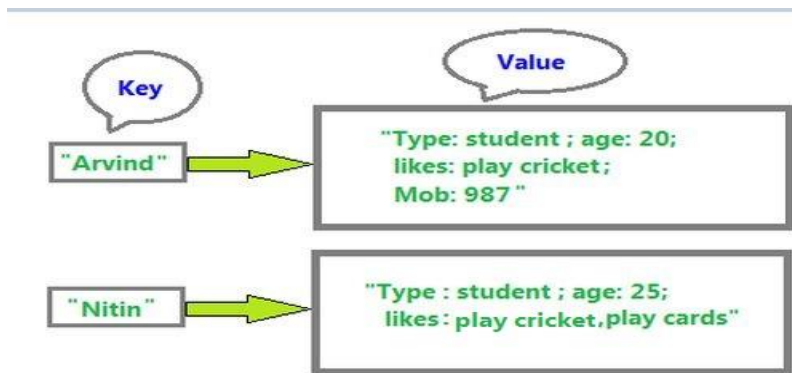
it follows the key-value pair way of storing data similar to JSON data. NoSQL database are highly preferred for large data set (i.e for big data). Hbase is an example for this purpose.

- For scalability: In most typical situations, SQL databases are vertically scalable. You can manage increasing load by increasing the CPU, RAM, SSD, etc, on a single server. On the other hand, NoSQL databases are horizontally scalable. You can just add few more servers easily in your NoSQL database infrastructure to handle the large traffic.
- For high transactional based application: SQL databases are best fit for heavy duty transactional type applications, as it is more stable and promises the atomicity as well as integrity of the data. While you can use NoSQL for transactions purpose, it is still not comparable and stable enough in high load and for complex transactional applications.
- For support: Excellent support are available for all SQL database from their vendors. There are also lot of independent consultations who can help you with SQL database for a very large scale deployments. For some NoSQL database you still have to rely on community support, and only limited outside experts are available for you to setup and deploy your large scale NoSQL deployments.
- For properties: SQL databases emphasizes on ACID properties (Atomicity, Consistency, Isolation and Durability) whereas the NoSQL database follows the Brewers CAP theorem (Consistency, Availability and Partition tolerance)
- For DB types: On a high-level, we can classify SQL databases as either open-source or close-sourced from commercial vendors. NoSQL databases can be classified on the basis of way of storing data as graph databases, key-value store databases, document store databases, column store database and XML databases.

Types of Nosql Databases: There are four general types of NoSQL databases, each with their own specific attributes:

1. **Key-Value storage**

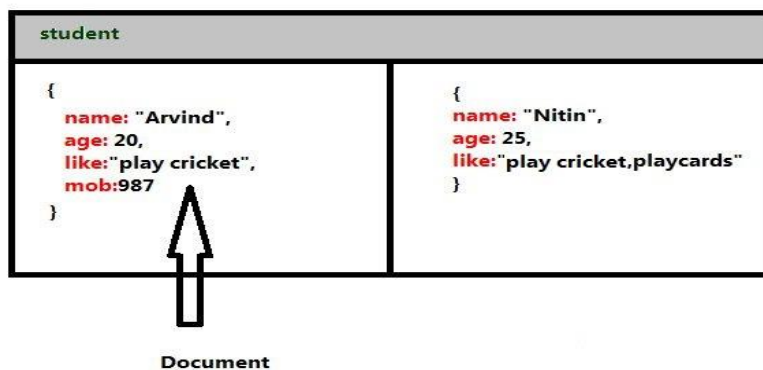
This is the first category of NoSQL database. Key-value stores have a simple data model, which allow clients to put a map/dictionary and request value par key. In the key-value storage, each key has to be unique to provide non-ambiguous identification of values. For example.



2. Document

databases

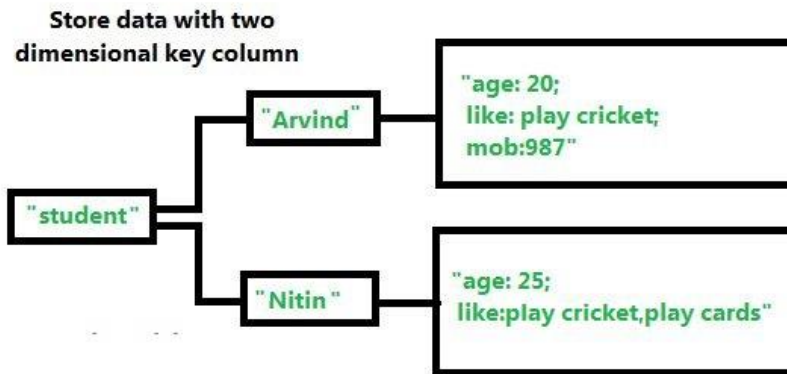
In the document database NoSQL store document in JSON format. JSON-based document are store in completely different sets of attributes can be stored together, which stores highly unstructured data as named value pairs and applications that look at user behavior, actions, and logs in real time.



3. Columns

storage

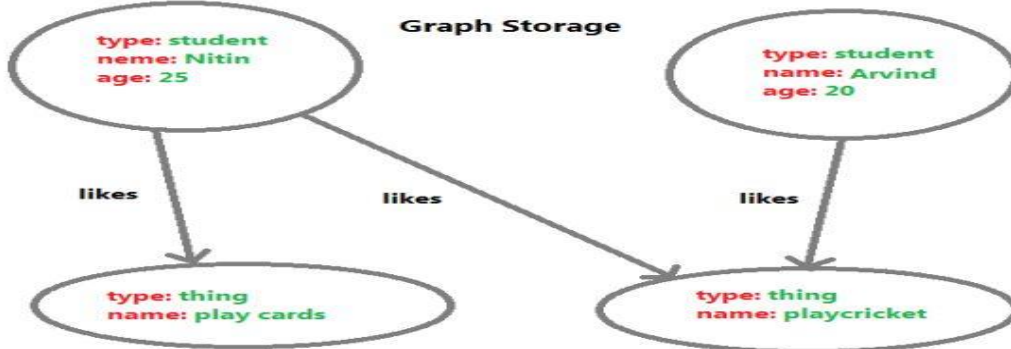
Columnar databases are almost like tabular databases. Thus keys in wide column store scan have many dimensions, resulting in a structure similar to a multi-dimensional, associative array. Shown in below example storing data in a wide column system using a two-dimensional key.



4. Graph

storage

Graph databases are best suited for representing data with a high, yet flexible number of interconnections, especially when information about those interconnections is at least as important as there presented data. In NoSQL database, data is stored in a graph like structures in graph databases, so that the data can be made easily accessible. Graph databases are commonly used on social networking sites. As show in below figure.



Example databases

Data Model	Example Databases
Key-Value	BerkeleyDB LevelDB Memcached Project Voldemort Redis <i>Riak</i>
Document	CouchDB <i>MongoDB</i> OrientDB RavenDB Terrastore
Column-Family	Amazon SimpleDB <i>Cassandra</i> HBase Hypertable
Graph	FlockDB HyperGraphDB Infinite Graph <i>Neo4J</i> OrientDB

Pros and Cons of Relational Databases

- Advantages
 - Data persistence
 - Concurrency – ACID, transactions, etc.
 - Integration across multiple applications
 - Standard Model – tables and SQL
- Disadvantages
 - Impedance mismatch
 - Integration databases vs. application databases
 - Not designed for clustering

Database Impedance mismatch:

Impedance Mismatch means the difference between data model and in memory data structures.

Impedance is the measure of the amount that some object resists (or obstruct, resist) the flow of another object.

Imagine you have a low current flashlight that normally uses AAA batteries. Suppose you could attach your car battery to the flashlight. The low current flashlight will pitifully output a fraction of the light energy that the high current battery is capable of producing. However, match the AAA batteries to the flashlight and they will run with maximum efficiency.

The data representation in RDMS is not matched with the data structure used in memory. In-memory, data structures are lists, dictionaries, nested and hierarchical data structures whereas in Relational database, it stores only atomic values, and there is no lists are nested records. Translating between these representations can be costly, confusing and limits the application development productivity.

Some common characteristics of nosql include:

- Does not use the relational model (mostly)
- Generally open source projects (currently)
- Driven by the need to run on clusters
- Built for the need to run 21st century web properties
- Schema-less
- **Polygot persistence:** The point of view of using different data stores in different circumstances is known as Polyglot Persistence.

Today, most large companies are using a variety of different data storage technologies for different kinds of data. Many companies still use relational databases to store some data, but the persistence needs of applications are evolving from predominantly relational to a mixture of data sources. Polyglot persistence is commonly used to define this hybrid approach. The definition of *polyglot* is “someone who speaks or writes several languages.” The term polyglot is redefined for big data as a set of applications that use several core database technologies.

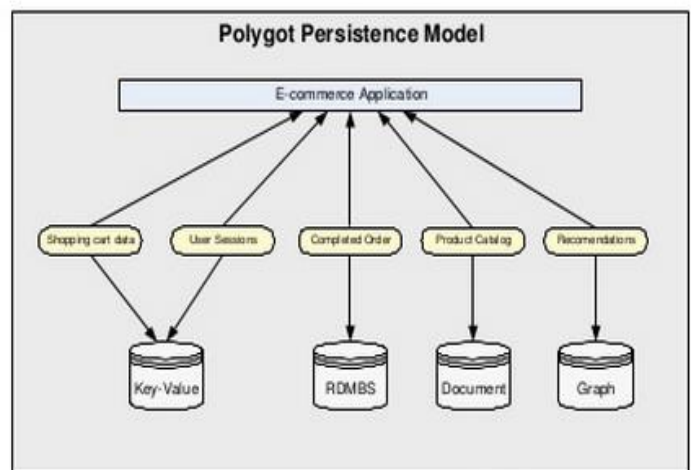
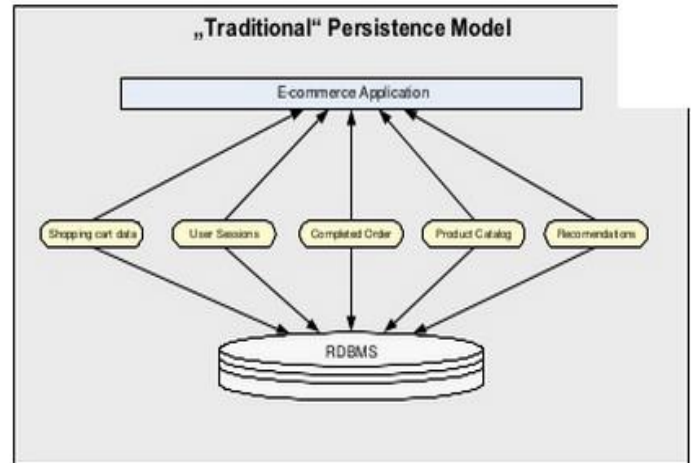
Polyglot Persistence

Today we use the same database for all kind of data

- Business transactions, session management data, reporting, logging information, content information, ...

No need for same properties of availability, consistency or backup requirements

Polyglot Data Storage Usage allows to mix and match Relational and NoSQL data stores

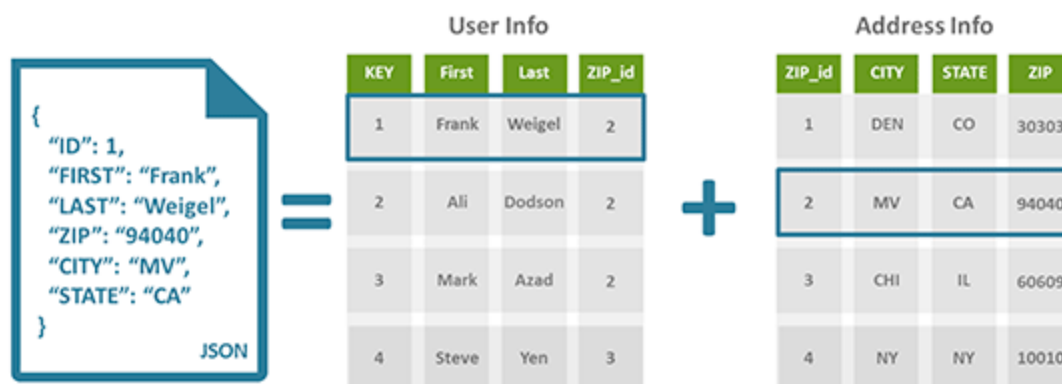


- **Auto Sharding:** NoSQL databases usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool

Nosql data model

Relational and NoSQL data models are very different. The relational model takes data and separates it into many interrelated tables that contain rows and columns. Tables reference each other through foreign keys that are stored in columns as well. When looking up data, the desired information needs to be collected from many tables (often hundreds in today's enterprise applications) and combined before it can be provided to the application. Similarly, when writing data, the write needs to be coordinated and performed on many tables.

NoSQL databases have a very different model. For example, a document-oriented NoSQL database takes the data you want to store and aggregates it into documents using the JSON format. Each JSON document can be thought of as an object to be used by your application. A JSON document might, for example, take all the data stored in a row that spans 20 tables of a relational database and aggregate it into a single document/object. Aggregating this information may lead to duplication of information, but since storage is no longer cost prohibitive, the resulting data model flexibility, ease of efficiently distributing the resulting documents and read and write performance improvements make it an easy trade-off for web-based applications.



Another major difference is that relational technologies have rigid schemas while NoSQL models are schemaless. Relational technology requires strict definition of a schema prior to storing any data into a database. Changing the schema once data is inserted is a big deal, extremely disruptive and frequently avoided – the exact opposite of the behavior desired in the Big Data era, where application developers need to constantly – and rapidly – incorporate new types of data to enrich their apps.

Aggregates data model in nosql

Data Model: A data model is the model through which we perceive and manipulate our data. For people using a database, the data model describes how we interact with the data in the database.

Relational Data Model: The relational model takes the information that we want to store and divides it into tuples.

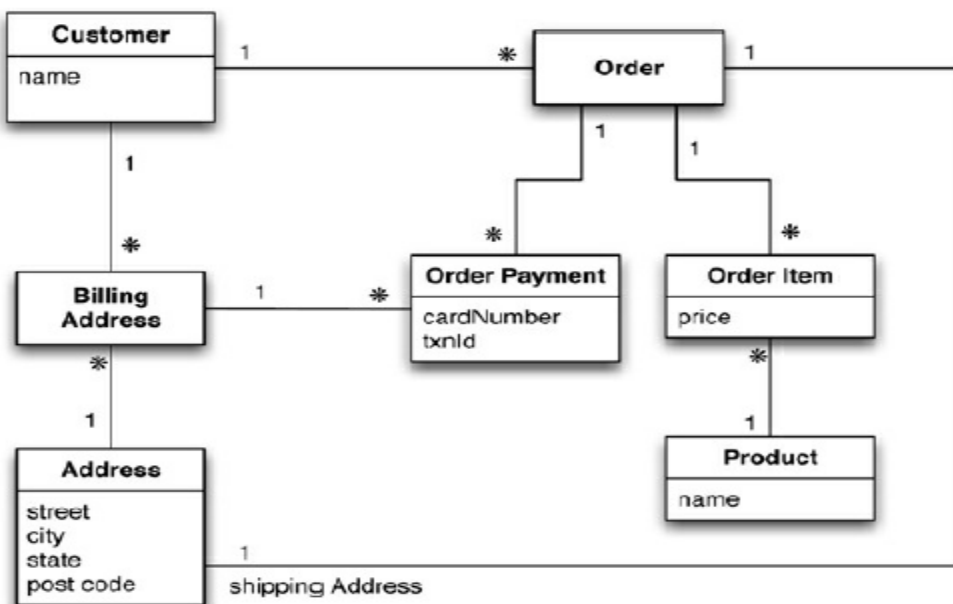
Tuple being a limited Data Structure it captures a set of values and can't be nested. This gives Relational Model a space of development.

Aggregate Model: Aggregate is a term that comes from Domain-Driven Design, an aggregate is a collection of related objects that we wish to treat as a unit, it is a unit for data manipulation and management of consistency.

- Atomic property holds within an aggregate
- Communication with data storage happens in unit of aggregate
- Dealing with aggregate is much more efficient in clusters
- It is often easier for application programmers to work with aggregates

Example of Relations and Aggregates

Let's assume we have to build an e-commerce website; we are going to be selling items directly to customers over the web, and we will have to store information about users, our product catalog, orders, shipping addresses, billing addresses, and payment data. We can use this scenario to model the data using a relation data store as well as NoSQL data stores and talk about their pros and cons. For a relational database, we might start with a data model shown in the following figure.



Data model oriented around a relational database

The following figure presents some sample data for this model.

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

BillingAddress		
Id	CustomerId	AddressId
55	1	77

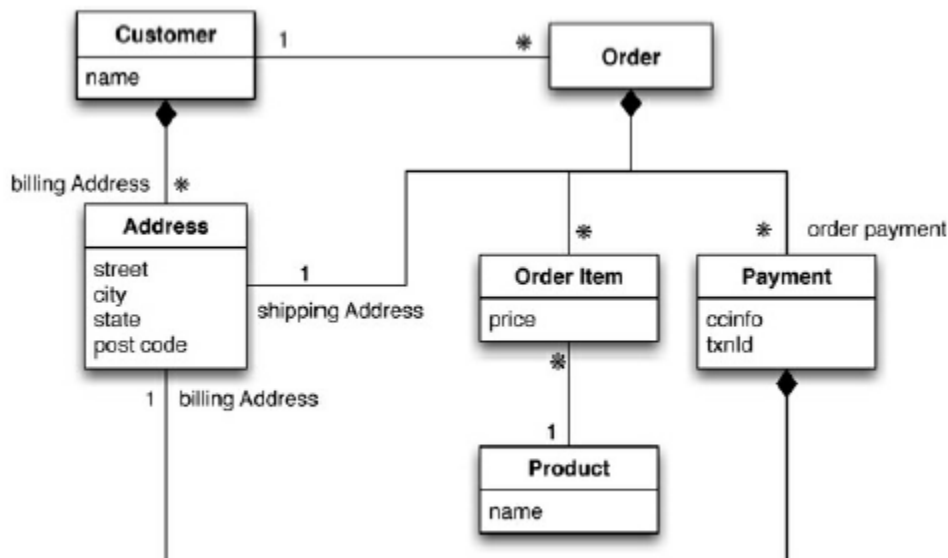
OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

Address	
Id	City
77	Chicago

OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abel1f879rft

Typical data using RDBMS data model

In relational, everything is properly normalized, so that no data is repeated in multiple tables. We also have referential integrity. A realistic order system would naturally be more involved than this. Now let's see how this model might look when we think in more aggregate oriented terms



An aggregate data model

Again, we have some sample data, which we'll show in JSON format as that's a common representation for data in NoSQL.

// in customers

```

{ "
    id":1,
    "name":"Martin",
    "billingAddress":[{"city":"Chicago"}]
}
// in orders
{ "
    id":99,
    "customerId":1,
    "orderItems":[
        {
            "productId":27,
            "price": 32.45,
            "productName": "NoSQL Distilled"
        }
    ],
    "shippingAddress":[{"city":"Chicago"}]
    "orderPayment":[
        {
            "ccinfo":"1000-1000-1000-1000",
            "txnId":"abelif879rft",
            "billingAddress": {"city": "Chicago"}
        }
    ],
}

```

In this model, we have two main aggregates: customer and order. We've used the black-diamond composition marker in UML to show how data fits into the aggregation structure. The customer contains a list of billing addresses; the order contains a list of order items, a shipping address, and payments. The payment itself contains a billing address for that payment.

A single logical address record appears three times in the example data, but instead of using IDs it's treated as a value and copied each time. This fits the domain where we would not want the shipping address, nor the payment's billing address, to change. In a relational database, we would ensure that the address rows aren't updated for this case, making a new row instead. With aggregates, we can copy the whole address structure into the aggregate as we need to.

Aggregate-Oriented Databases: Aggregate-oriented databases work best when most data interaction is done with the same aggregate; aggregate-ignorant databases are better when interactions use data organized in many different formations.

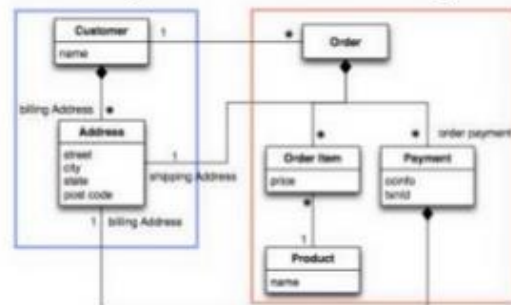
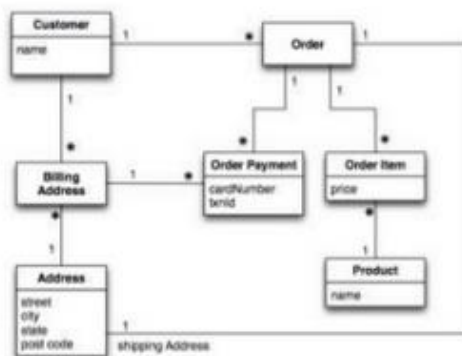
- **Key-value databases**
 - ✓ •Stores data that is opaque to the database
 - ✓ •The database does not see the structure of records
 - ✓ •Application needs to deal with this
 - ✓ •Allows flexibility regarding what is stored (i.e. text or binary data)
- **Document databases**
 - ✓ •Stores data whose structure is visible to the database
 - ✓ •Imposes limitations on what can be stored
 - ✓ •Allows more flexible access to data (i.e. partial records) via querying

Both key-value and document databases consist of aggregate records accessed by ID values

- **Column-family databases**
 - ✓ •Two levels of access to aggregates (and hence, two parts to the "key" to access an aggregate's data)
 - ✓ •ID is used to look up aggregate record
 - ✓ •Column name – either a label for a value (name) or a key to a list entry (order id)
 - ✓ •Columns are grouped into column families

Relational vs. Aggregate Data Models

- The relational model takes the information and divides it into tuples (rows)
- A tuple is a limited data structure
 - no nesting of tuples
 - no list of values
- Aggregate is a term that comes from Domain-Driven Design (Evans)
- An aggregate is a collection of related objects, that should be treated as a unit
 - Unit for data manipulation and management of consistency



Relational Instance

CUSTOMER	
ID	NAME
1	Guido

PRODUCT	
ID	NAME
1000	iPod Touch
1020	Monster Beat

BILLING_ADDRESS		
ID	CUSTOMER_ID	ADDRESS_ID
1	1	55

ADDRESS			
ID	STREET	CITY	POST_CODE
55	Chaumontweg	Spiegel	3095

ORDER		
ID	CUSTOMER_ID	SHIPPING_ADDRESS_ID
90	1	55

ORDER_ITEM			
ID	ORDER_ID	PRODUCT_ID	PRICE
1	90	1000	250.55
1	90	1020	199.55

Aggregate Instance

```
{
  „id”:1,
  „name”:„Guido”,
  „billingAddress”:{„street”:„Chaumontweg”,„city”:„Spiegel”,„postCode”:„3095”}
}

{
  „id”:90,
  „customerid”:1,
  „orderItems”:{
    {
      „productId”:1000,„price”:250.55, „productName”: „iPod Touch”
    },
    {
      „productId”:1020,„price”:199.55, „productName”: „Monster Beat”
    }
  },
  „shippingAddress”:{„street”:„Chaumontweg”,„city”:„Spiegel”,„postCode”:„3095”}
}
```

Schemaless Databases

A common theme across all the forms of NoSQL databases is that they are schemaless. When you want to store data in a relational database, you first have to define a schema—a defined structure for the database which says what tables exist, which columns exist, and what data types each column can hold. Before you store some data, you have to have the schema defined for it.

With NoSQL databases, storing data is much more casual. A key-value store allows you to store any data you like under a key. A document database effectively does the same thing, since it makes no restrictions on the structure of the documents you store. Column-family databases allow you to store any data under any column you like. Graph databases allow you to freely add new edges and freely add properties to nodes and edges as you wish.

why schemaless?

- ✓ A schemaless store also makes it easier to deal with nonuniform data
- ✓ When starting a new development project you don't need to spend the same amount of time on up-front design of the schema.
- ✓ No need to learn SQL or database specific stuff and tools.
- ✓ The rigid schema of a relational database (RDBMS) means you have to absolutely follow the schema. It can be harder to push data into the DB as it has to perfectly fit the schema. Being able to add data directly without having to tweak it to match the schema can save you time
- ✓ Minor changes to the model and you will have to change both your code and the schema in the DBMS. If no schema, you don't have to make changes in two places. Less time consuming
- ✓ With a NoSql DB you have fewer ways to pull the data out
- ✓ Less overhead for DB engine
- ✓ Less overhead for developers related to scalability
- ✓ Eliminates the need for Database administrators or database experts -> fewer people involved and less waiting on experts
- ✓ Save time writing complex SQL joins -> more rapid development

Pros and cons of schemaless data

Pros:

- More freedom and flexibility
- you can easily change your data organization
- you can deal with nonuniform data

Cons:

- A program that accesses data:
 - almost always relies on some form of implicit schema
 - it assumes that certain fields are present
 - carry data with a certain meaning

- ❑ The implicit schema is shifted into the application code that accesses data
- ❑ To understand what data is present you have look at the application code
- ❑ The schema cannot be used to:
 - decide how to store and retrieve data efficiently
 - ensure data consistency
- ❑ Problems if multiple applications, developed by different people, access the same database.
- ❑ Relational schemas can be changed at any time with standard SQL commands

Key-value databases

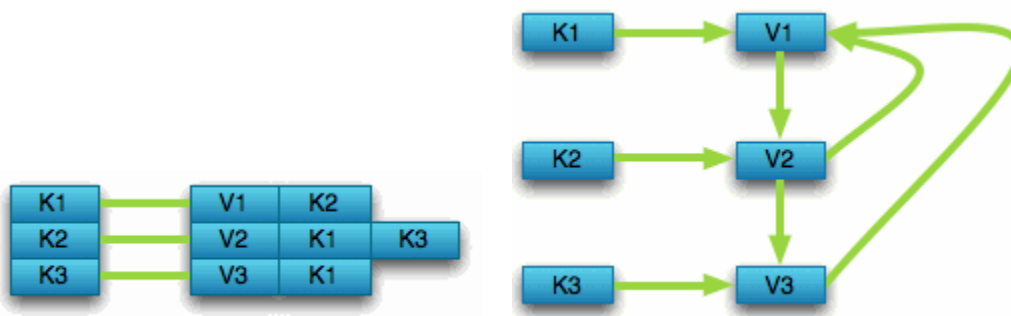
A key-value store is a simple hash table, primarily used when all access to the database is via primary key.

Key-value stores are the simplest NoSQL data stores to use from an API perspective. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a BLOB(Binary Large Object) that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.

It is an associative container such as map, dictionary, and in query processing an index. It is an abstract data type composed of a collection of unique keys and a collection of values, where each key is associated with one value (or set of values). The operation of finding the value associated with a key is called a lookup or indexing. The relationship between a key and its value is sometimes called a mapping or binding.

Some of the popular key-value databases are Riak, Redis, Memcached DB, Berkeley DB, HamsterDB, Amazon DynamoDB.

A Key-Value model is great for lookups of simple or even complex values. When the values are themselves interconnected, you've got a graph as shown in following figure. Lets you traverse quickly among all the connected values.



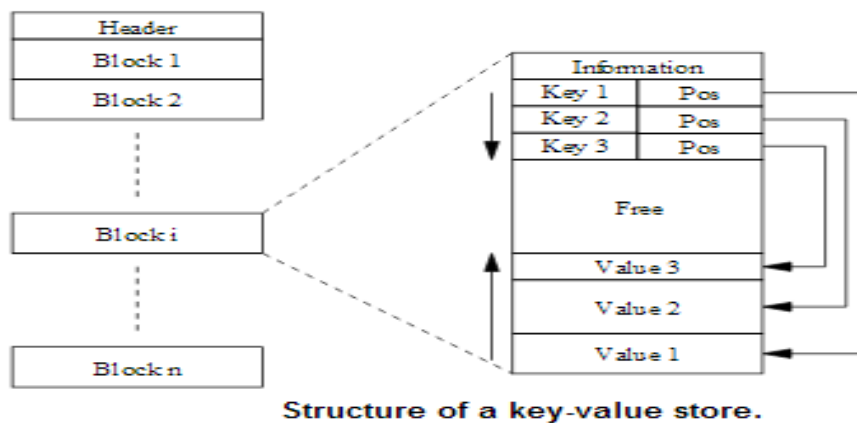
In Key-Value database,

- Data is stored sorted by key.
- Callers can provide a custom comparison function to override the sort order.
- The basic operations are Put(key,value), Get(key), Delete(key).
- Multiple changes can be made in one atomic batch.
- Users can create a transient snapshot to get a consistent view of data.
- Forward and backward iteration is supported over the data.

In key-value databases, a single object that stores all the data and is put into a single bucket. Buckets are used to define a virtual keyspace and provide the ability to define isolated non-default configuration. Buckets might be compared to tables or folders in relational databases or file systems, respectively.

As their name suggest, they store key/value pairs. For example, for search engines, a store may associate to each keyword (the key) a list of documents containing it (the corresponding value).

One approach to implement a key-value store is to use a file decomposed in blocks . As the following figure shows, each block is associated with a number (ranging from 1 to n). Each block manages a set of key-value pairs: the beginning of the block contained, after some information, an index of keys and the position of the corresponding values. These values are stored starting from the end of the block (like a memory heap). The free space available is delimited by the end of the index and the end of the values.



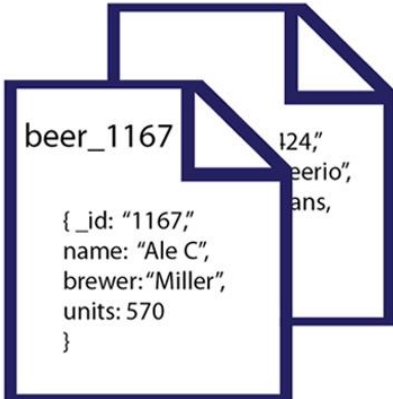
In this implementation, the size of a block is important since it defines the largest value that can be stored (for example the longest list of document identifiers containing a given keyword). Moreover, it supposes that a block number is associated to each key. These block numbers can be assigned in two different ways:

1. The block number is obtained directly from the key, typically by using a hash function. The size of the file is then defined by the largest block number computed by every possible key.
2. The block number is assigned increasingly. When a new pair must be stored, the first block that can hold it is chosen. In practice, a given amount of space is reserved in a block in order to manage updates of existing pairs (a new value can replace an older and smaller one). This limit the size of the file to amount of values to store.

Document Databases

In a relational database system you must define a *schema* before adding records to a database. The schema is the structure described in a formal language supported by the database and provides a blueprint for the tables in a database and the relationships between tables of data. Within a table, you need to define constraints in terms of rows and named columns as well as the type of data that can be stored in each column.

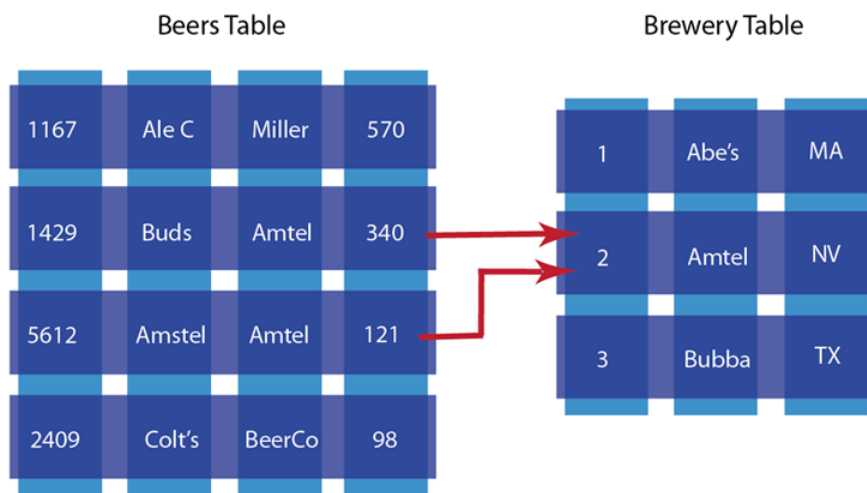
In contrast, a document-oriented database contains *documents*, which are records that describe the data in the document, as well as the actual data. Documents can be as complex as you choose; you can use nested data to provide additional sub-categories of information about your object. You can also use one or more document to represent a real-world object. The following compares a conventional table with document-based objects:

Beers Table				Beer Documents	
1167	Ale C	Miller	570		
3424	Beerio	Ians	340		
5612	Amstel	Amtel	121		
2409	Colt's	BeerCo	98		

In this example we have a table that represents beers and their respective attributes: id, beer name, brewer, bottles available and so forth. As we see in this illustration, the relational model conforms to a schema with a specified number of fields which represent a specific purpose and data type. The equivalent document-based model has an individual document per beer; each document contains the same types of information for a specific beer.

In a document-oriented model, data objects are stored as documents; each document stores your data and enables you to update the data or delete it. Instead of columns with names and data types, we describe the data in the document, and provide the value for that description. If we wanted to add attributes to a beer in a relational mode, we would need to modify the database schema to include the additional columns and their data types. In the case of document-based data, we would add additional key-value pairs into our documents to represent the new fields.

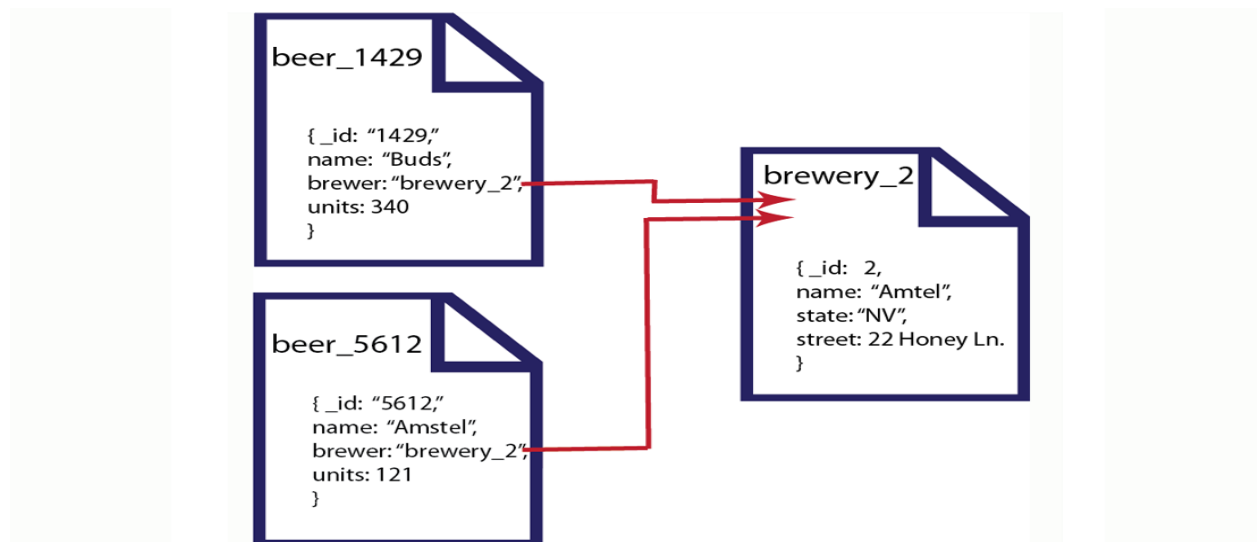
The other characteristic of relational database is *data normalization*; this means you decompose data into smaller, related tables. The figure below illustrates this:



In the relational model, data is shared across multiple tables. The advantage to this model is that there is less duplicated data in the database. If we did not separate beers and brewers into different tables and had one beer table instead, we would have repeated information about breweries for each beer produced by that brewer.

The problem with this approach is that when you change information across tables, you need to lock those tables simultaneously to ensure information changes across the table consistently. Because you also spread information across a rigid structure, it makes it more difficult to change the structure during production, and it is also difficult to distribute the data across multiple servers.

In the document-oriented database, we could choose to have two different document structures: one for beers, and one for breweries. Instead of splitting your application objects into tables and rows, you would turn them into documents. By providing a reference in the beer document to a brewery document, you create a relationship between the two entities:



In this example we have two different beers from the Amtel brewery. We represent each beer as a separate document and reference the brewery in the **brewer** field. The document-oriented approach provides several upsides compared to the traditional RDBMS model. First, because information is stored in documents, updating a schema is a matter of updating the documents for that type of object. This can be done with no system downtime. Secondly, we can distribute the information across multiple servers with greater ease. Since records are contained within entire documents, it makes it easier to move, or replicate an entire object to another server.

Using JSON Documents

JavaScript Object Notation (JSON) is a lightweight data-interchange format which is easy to read and change. JSON is language-independent although it uses similar constructs to JavaScript. The following are basic data types supported in JSON:

- Numbers, including integer and floating point,
- Strings, including all Unicode characters and backslash escape characters,
- Boolean: true or false,
- Arrays, enclosed in square brackets: ["one", "two", "three"]
- Objects, consisting of key-value pairs, and also known as an *associative array* or hash. The key must be a string and the value can be any supported JSON data type.

For instance, if you are creating a beer application, you might want particular document structure to represent a beer:

```
{
  "name":
  "description":
  "category":
  "updated":
```

```
}
```

For each of the keys in this JSON document you would provide unique values to represent individual beers. If you want to provide more detailed information in your beer application about the actual breweries, you could create a JSON structure to represent a brewery:

```
{  
  "name":  
  "address":  
  "city":  
  "state":  
  "website":  
  "description":  
}
```

Performing data modeling for a document-based application is no different than the work you would need to do for a relational database. For the most part it can be much more flexible, it can provide a more realistic representation of your application data, and it also enables you to change your mind later about data structure. For more complex items in your application, one option is to use nested pairs to represent the information:

```
{  
  "name":  
  "address":  
  "city":  
  "state":  
  "website":  
  "description":  
  "geo":  
  {  
    "location": ["-105.07", "40.59"],  
    "accuracy": "RANGE_INTERPOLATED"  
  }  
  "beers": [_id4058, _id7628]  
}
```

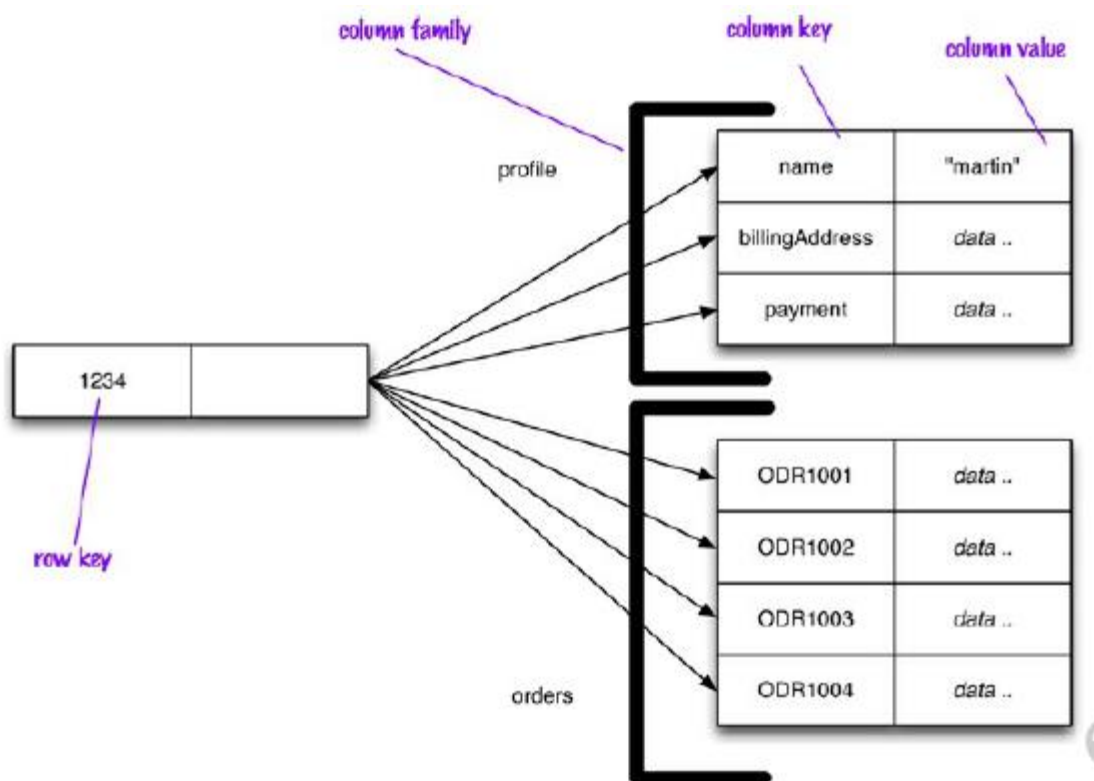
In this case we added a nested attribute for the geo-location of the brewery and for beers. Within the location, we provide an exact longitude and latitude, as well as level of accuracy for plotting it on a map. The level of nesting you provide is your decision; as long as a document is under the maximum storage size for Server, you can provide any level of nesting that you can handle in your application.

In traditional relational database modeling, you would create tables that contain a subset of information for an item. For instance a brewery may contain types of beers which are stored in a separate table and referenced by the beer id. In the case of JSON documents, you use key-values pairs, or even nested key-value pairs.

Column-Family Stores

Its name conjured up a tabular structure which it realized with sparse columns and no schema. The column-family model is as a two-level aggregate structure. As with key-value stores, the first key is often described as a row identifier, picking up the aggregate of interest. The difference with column family structures is that this row aggregate is itself formed of a map of more detailed values. These second-level values are referred to as columns. As well as accessing the row as a whole, operations also allow picking out a particular column, so to get a particular customer's name from following figure you could do something like

`get('1234', 'name').`



Column-family databases organize their columns into column families. Each column has to be part of a single column family, and the column acts as unit for access, with the assumption that data for a particular column family will be usually accessed together.

The data is structured into:

- **Row-oriented:** Each row is an aggregate (for example, customer with the ID of 1234) with column families representing useful chunks of data (profile, order history) within that aggregate.

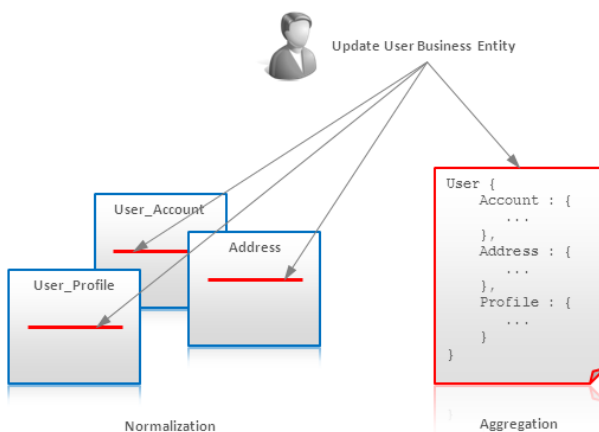
- **Column-oriented:** Each column family defines a record type (e.g., customer profiles) with rows for each of the records. You then think of a row as the join of records in all column families.

Even though a document database declares some structure to the database, each document is still seen as a single unit. Column families give a two-dimensional quality to column-family databases.

Cassandra uses the terms “wide” and “skinny.” **Skinny rows** have few columns with the same columns used across the many different rows. In this case, the column family defines a record type, each row is a record, and each column is a field. A **wide row** has many columns (perhaps thousands), with rows having very different columns. A wide column family models a list, with each column being one element in that list.

Relationships: Atomic Aggregates

Aggregates allow one to store a single business entity as one document, row or key-value pair and update it atomically:



Graph Databases:

Graph databases are one style of NoSQL databases that uses a distribution model similar to relational databases but offers a different data model that makes it better at handling data with complex relationships.

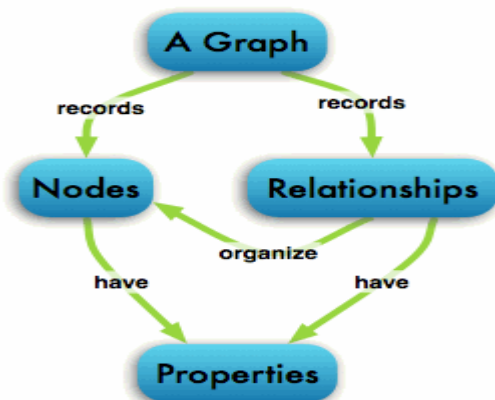
- Entities are also known as nodes, which have properties
- Nodes are organized by relationships which allows to find interesting patterns between the nodes
- The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships

Let's follow along some graphs, using them to express themselves. We'll read" a graph by following arrows around the diagram to form sentences.

A Graph contains Nodes and Relationships

A Graph -[:RECORDS_DATA_IN]-> Nodes -[:WHICH_HAVE]-> Properties.

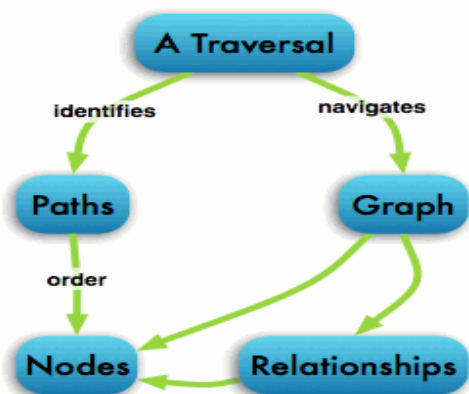
The simplest possible graph is a single Node, a record that has named values referred to as Properties. A Node could start with a single Property and grow to a few million, though that can get a little awkward. At some point it makes sense to distribute the data into multiple nodes, organized with explicit Relationships.



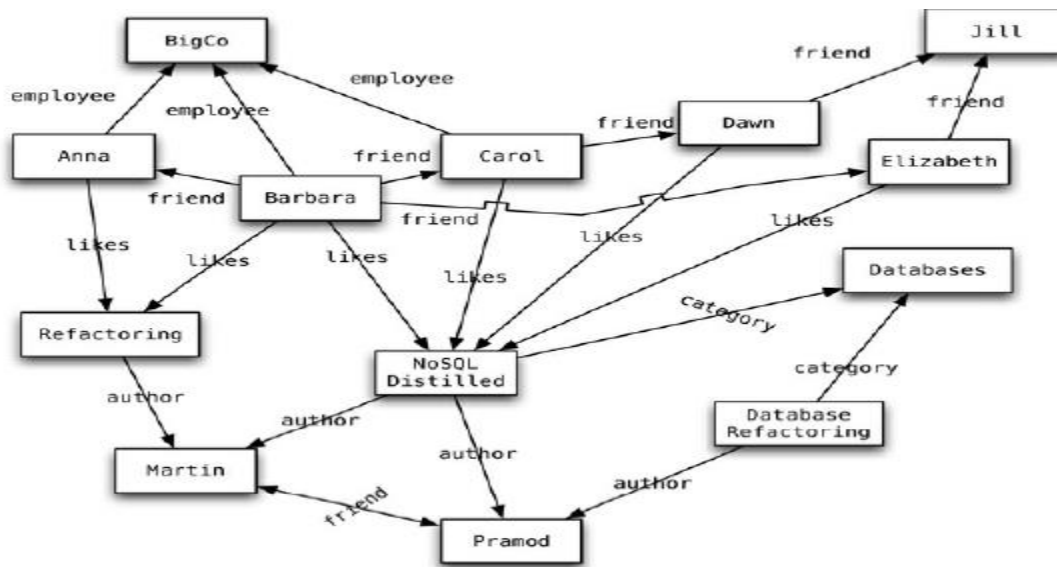
Query a Graph with a Traversal

A Traversal -navigates-> a Graph; it -identifies-> Paths -which order-> Nodes.

A Traversal is how you query a Graph, navigating from starting Nodes to related Nodes according to an algorithm, finding answers to questions like "what music do my friends like that I don't yet own," or "if this power supply goes down, what web services are affected?"



Example



An example graph structure

In this context, a graph refers to a graph data structure of nodes connected by edges. In the above figure we have a web of information whose nodes are very small (nothing more than a name) but there is a rich structure of interconnections between them. With this structure, we can ask questions such as “ **find the books in the Databases category that are written by someone whom a friend of mine likes.**”

Graph databases specialize in capturing this sort of information—but on a much larger scale than a readable diagram could capture. This is ideal for capturing any data consisting of complex relationships such as social networks, product preferences, or eligibility rules.

Materialized Views

In computing, a materialized view is a database object that contains the results of a query. For example, it may be a local copy of data located remotely, or may be a subset of the rows and/or columns of a table or join result, or may be a summary based on aggregations of a table's data. Materialized views can be used within the same aggregate.

Materialized views, which store data based on remote tables, are also known as snapshots. A snapshot can be redefined as a materialized view.

Materialized view is computed in advance and cached on disk.

Strategies to building a materialized view:

Eager approach: the materialized view is updated at the same time of the base data. It is good when you have more frequent reads than writes.

Detached approach: batch jobs update the materialized views at regular intervals. It is good when you don't want to pay an overhead on each update.

NoSQL databases do not have views and have precomputed and cached queries usually called "materialized view".

Distribution Models

Multiple servers: In NoSQL systems, data distributed over large clusters

Single server – simplest model, everything on one machine. Run the database on a single machine that handles all the reads and writes to the data store. We prefer this option because it eliminates all the complexities. It's easy for operations people to manage and easy for application developers to reason about.

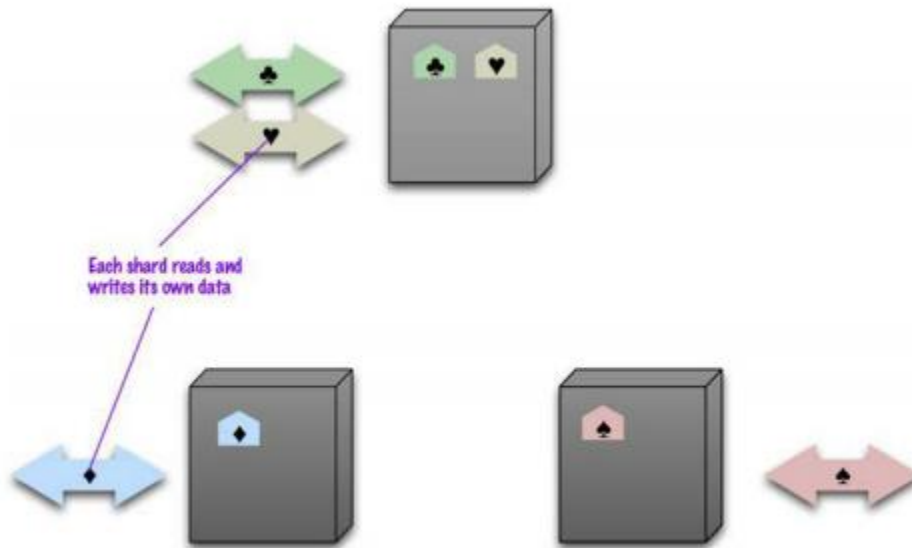
Although a lot of NoSQL databases are designed around the idea of running on a cluster, it can make sense to use NoSQL with a single-server distribution model if the data model of the NoSQL store is more suited to the application. Graph databases are the obvious category here – these work best in a single-server configuration.

If your data usage is mostly about processing aggregates, then a single-server document or key-value store may well be worthwhile because it's easier on application developers.

Orthogonal aspects of data distribution models:

Sharding: DB Sharding is nothing but horizontal partitioning of data. Different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers – a technique that's called sharding.

A table with billions of rows can be partitioned using "Range Partitioning". If the customer transaction date, for an example, based partitioning will partition the data vertically. So irrespective which instance in a Real Application Clusters access the data, it is "not" horizontally partitioned although Global Enqueue Resources are owning certain blocks in each instance but it can be moving around. But in "db shard" environment, the data is horizontally partitioned. For an example: United States customer can live in one shard and European Union customers can be in another shard and the other countries customers can live in another shard but from an access perspective there is no need to know where the data lives. The DB Shard can go to the appropriate shard to pick up the data.



- ☐ Different parts of the data onto different servers
 - Horizontal scalability
 - Ideal case: different users all talking to different server nodes
 - Data accessed together on the same node-aggregate unit!
- ☐ Pros: it can improve both reads and writes
- ☐ Cons: Clusters use less reliable machines-resilience decreases

Many NoSQL databases offer auto-sharding

- ☐ the database takes on the responsibility of sharding

Improving performance:

Main rules of sharding:

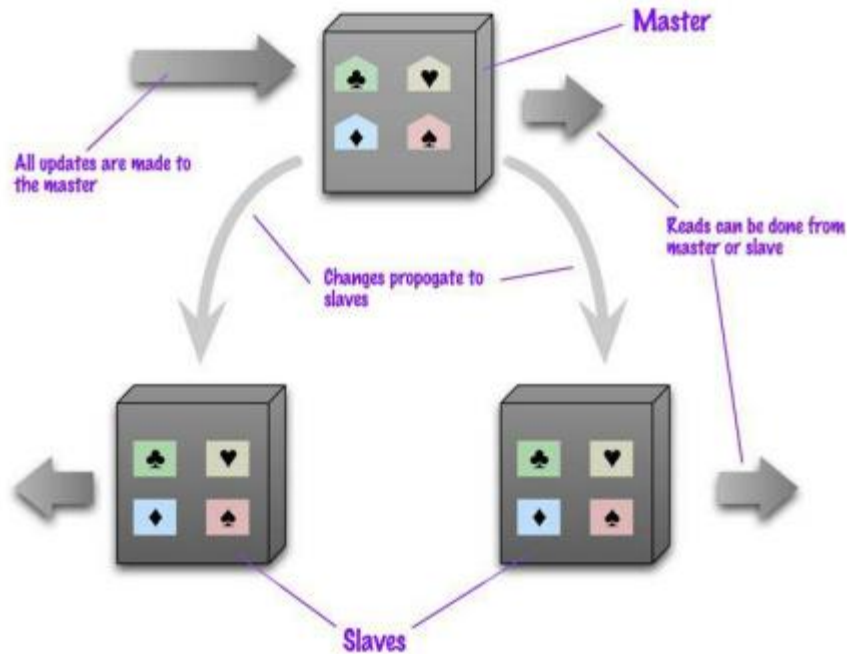
1. Place the data close to where it's accessed
 - ☐ Orders for Boston: data in your eastern US data center
2. Try to keep the load even
 - ☐ All nodes should get equal amounts of the load
3. Put together aggregates that may be read in sequence
 - ☐ Same order, same node

Master-Slave Replication

- ☐ Master
 - ☐ is the authoritative source for the data
 - ☐ is responsible for processing any updates to that data
 - ☐ can be appointed manually or automatically

☐ Slaves

- ☐ A replication process synchronizes the slaves with the master
- ☐ After a failure of the master, a slave can be appointed as new master very quickly



Pros and cons of Master-Slave Replication

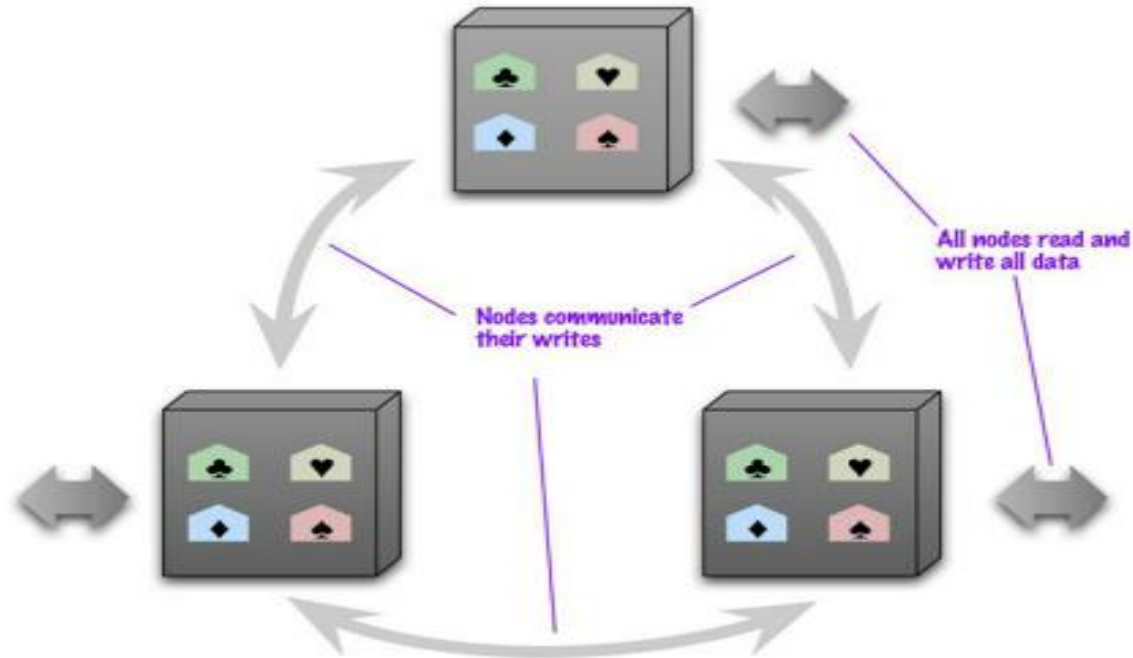
☐ Pros

- ☐ More read requests:
- ☐ Add more slave nodes
- ☐ Ensure that all read requests are routed to the slaves
- ☐ Should the master fail, the slaves can still handle read requests
- ☐ Good for datasets with a read-intensive dataset

☐ Cons

- ☐ The master is a bottleneck
- ☐ Limited by its ability to process updates and to pass those updates on
- ☐ Its failure does eliminate the ability to handle writes until:
 - ☐ the master is restored or
 - ☐ a new master is appointed
- ☐ Inconsistency due to slow propagation of changes to the slaves
- ☐ Bad for datasets with heavy write traffic

Peer-to-Peer Replication



- All the replicas have equal weight, they can all accept writes
- The loss of any of them doesn't prevent access to the data store.

Pros and cons of peer-to-peer replication

□ Pros:

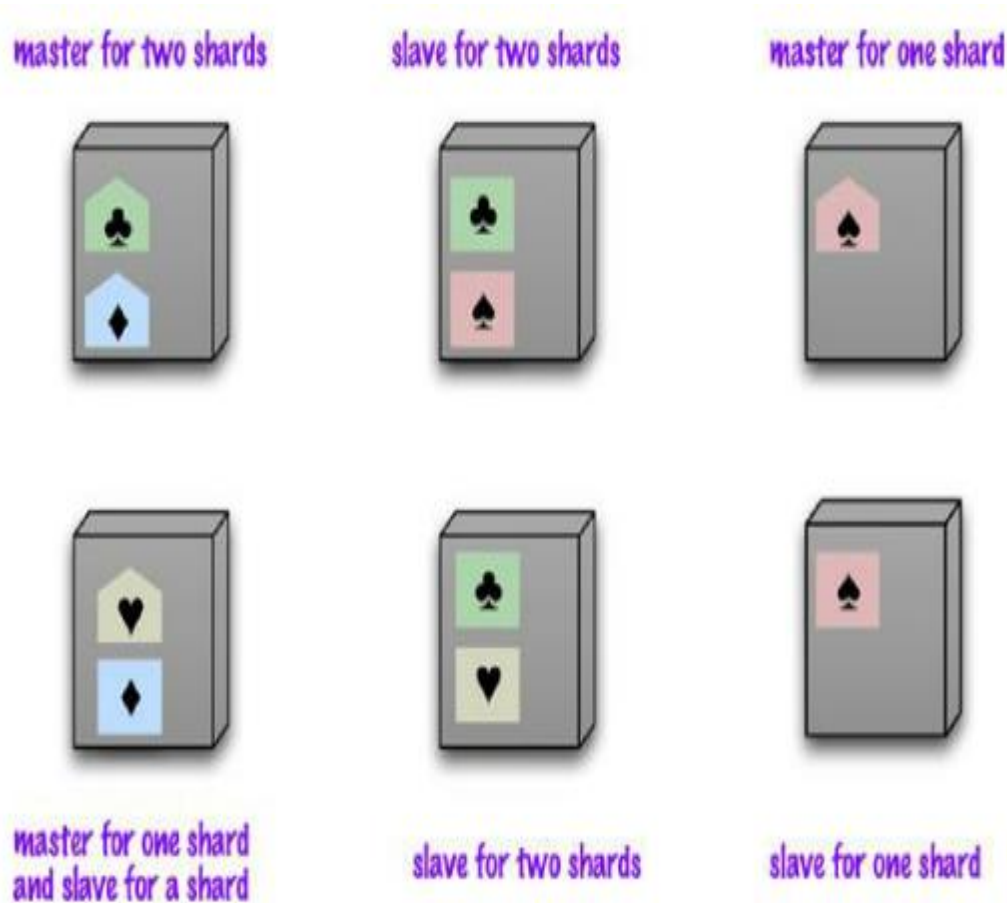
- you can ride over node failures without losing access to data
- you can easily add nodes to improve your performance

□ Cons:

- Inconsistency
- Slow propagation of changes to copies on different nodes
- Inconsistencies on read lead to problems but are relatively transient
- Two people can update different copies of the same record stored on different nodes at the same time - a write-write conflict.
- Inconsistent writes are forever.

Sharding and Replication on Master-Slave

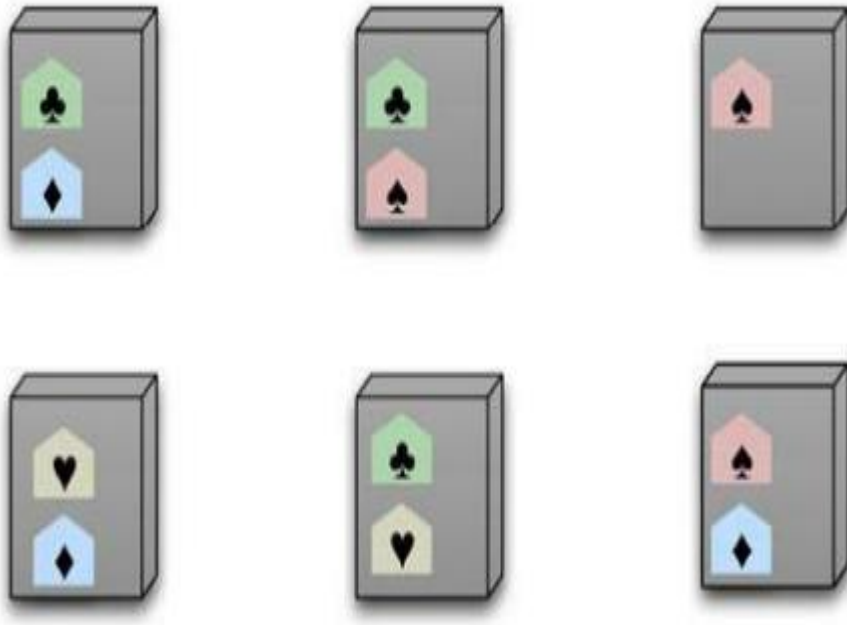
Replication and sharding are strategies that can be combined. If we use both master slave replication and sharding, this means that we have multiple masters, but each data item only has a single master. Depending on your configuration, you may choose a node to be a master for some data and slaves for others, or you may dedicate nodes for master or slave duties.



- ☐ We have multiple masters, but each data only has a single master.
- ☐ Two schemes:
 - ☐ A node can be a master for some data and slaves for others
 - ☐ Nodes are dedicated for master or slave duties

Sharding and Replication on P2P

Using peer-to-peer replication and sharding is a common strategy for column family databases. In a scenario like this you might have tens or hundreds of nodes in a cluster with data sharded over them. A good starting point for peer-to-peer replication is to have a replication factor of 3, so each shard is present on three nodes. Should a node fail, then the shards on that node will be built on the other nodes. (See following figure)



- Usually each shard is present on three nodes
- A common strategy for column-family databases

Key Points

- There are two styles of distributing data:
 - Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
 - Replication copies data across multiple servers, so each bit of data can be found in multiple places.

A system may use either or both techniques.

- 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 point of failure.

Consistency

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.

It is a biggest change from a centralized relational database to a cluster oriented NoSQL. Relational databases has strong consistency whereas NoSQL systems has mostly eventual consistency.

ACID: A DBMS is expected to support “ACID transactions,” processes that are:

- Atomicity: either the whole process is done or none is
- Consistency: only valid data are written
- Isolation: one operation at a time
- Durability: once committed, it stays that way

Various forms of consistency

1. Update Consistency (or write-write conflict):

Martin and Pramod are looking at the company website and notice that the phone number is out of date. Incredibly, they both have update access, so they both go in at the same time to update the number. To make the example interesting, we'll assume they update it slightly differently, because each uses a slightly different format. This issue is called a **write-write conflict**: two people updating the same data item at the same time.

When the writes reach the server, the server will serialize them—decide to apply one, then the other. Let's assume it uses alphabetical order and picks Martin's update first, then Pramod's. Without any concurrency control, Martin's update would be applied and immediately overwritten by Pramod's. In this case Martin's is a lost update. We see this as a failure of consistency because Pramod's update was based on the state before Martin's update, yet was applied after it.

☐ Solutions:

☐ Pessimistic approach

☐ Prevent conflicts from occurring

☐ Usually implemented with write locks managed by the system

☐ Optimistic approach

☐ Lets conflicts occur, but detects them and takes action to sort them out

☐ Approaches:

☐ conditional updates: test the value just before updating

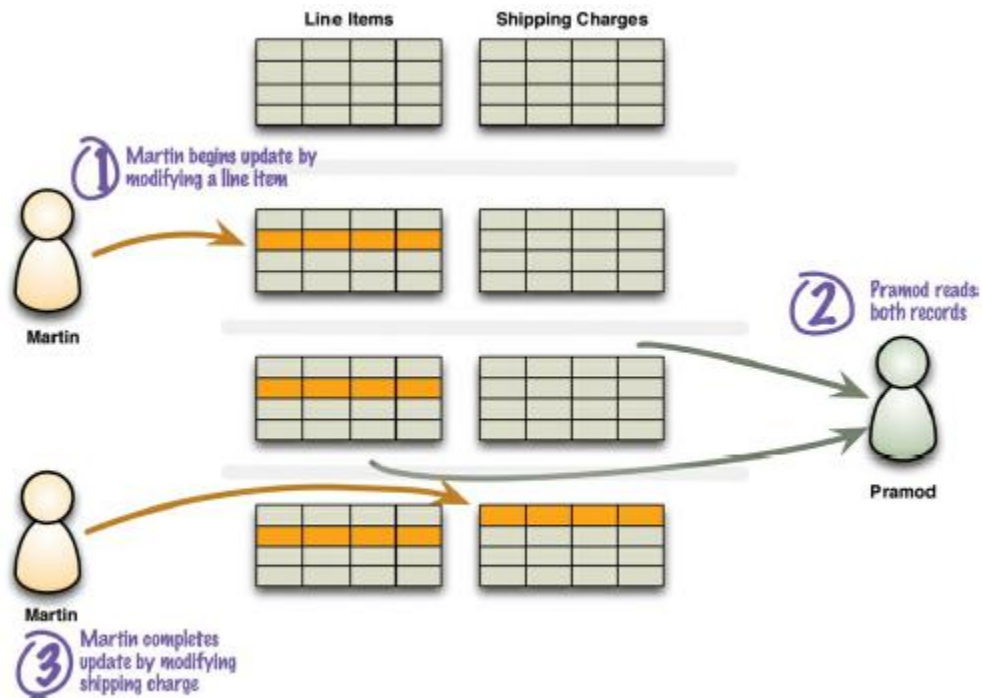
☐ save both updates: record that they are in conflict and then merge them

☐ Do not work if there's more than one server (peer-to-peer replication)

2. Read Consistency (or read-write conflict)

Alice and Bob are using Ticketmaster website to book tickets for a specific show. Only one ticket is left for the specific show. Alice signs on to Ticketmaster first and finds one left, and finds it expensive. Alice takes time to decide. Bob signs on and finds one ticket left, orders it instantly. Bob purchases and logs off. Alice decides to buy a ticket, to find there are no tickets. This is a typical Read-Write Conflict situation.

Another example where Pramod has done a read in the middle of Martin's write as shown in below.



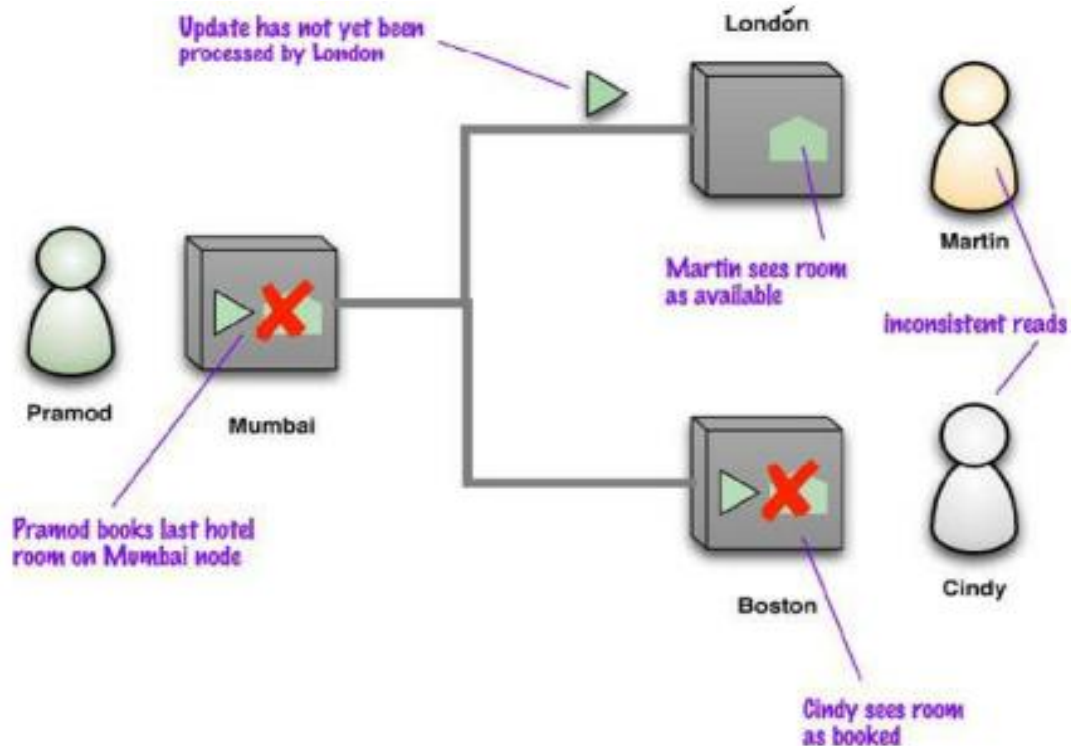
A read-write conflict in logical consistency

We refer to this type of consistency as logical consistency. To avoid a logically inconsistent by providing Martin wraps his two writes in a transaction, the system guarantees that Pramod will either read both data items before the update or both after the update. The length of time an inconsistency is present is called the **inconsistency window**.

Replication consistency

Let's imagine there's one last hotel room for a desirable event. The hotel reservation system runs on many nodes. Martin and Cindy are a couple considering this room, but they are discussing this on the phone because Martin is in London and Cindy is in Boston. Meanwhile Pramod, who is in Mumbai, goes and books that last room. That updates the replicated room availability, but the update gets to Boston quicker than it gets to London. When Martin and Cindy fire up their browsers to see if the room is available, Cindy sees it booked and Martin sees it free. This is another inconsistent read—but it's a breach of a different form of consistency we call

replication consistency: ensuring that the same data item has the same value when read from different replicas.



Eventual consistency: At any time, nodes may have replication inconsistencies but, if there are no further updates, eventually all nodes will be updated to the same value. In other words, Eventual consistency is a consistency model used in nosql database to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Eventually consistent services are often classified as providing BASE (Basically Available, Soft state, Eventual consistency) semantics, in contrast to traditional ACID (Atomicity, Consistency, Isolation, Durability) guarantees.

Basic Availability. The NoSQL database approach focuses on availability of data even in the presence of multiple failures. It achieves this by using a highly distributed approach to database management. Instead of maintaining a single large data store and focusing on the fault tolerance of that store, NoSQL databases spread data across many storage systems with a high degree of replication. In the unlikely event that a failure disrupts access to a segment of data, this does not necessarily result in a complete database outage.

Soft state. BASE databases abandon the consistency requirements of the ACID model pretty much completely. One of the basic concepts behind BASE is that data consistency is the developer's problem and should not be handled by the database.

Eventual Consistency. The only requirement that NoSQL databases have regarding consistency is to require that at some point in the future, data will converge to a consistent state. No guarantees are made, however, about when this will occur. That is a complete departure from the immediate consistency requirement of ACID that prohibits a transaction from executing until the prior transaction has completed and the database has converged to a consistent state.

Version stamp: A field that changes every time the underlying data in the record changes. When you read the data you keep a note of the version stamp, so that when you write data you can check to see if the version has changed.

You may have come across this technique with updating resources with HTTP. One way of doing this is to use etags. Whenever you get a resource, the server responds with an etag in the header. This etag is an opaque string that indicates the version of the resource. If you then update that resource, you can use a conditional update by supplying the etag that you got from your last GET method. If the resource has changed on the server, the etags won't match and the server will refuse the update, returning a 412 (Precondition Failed) error response. In short,

- ☐ It helps you detect concurrency conflicts.
- ☐ When you read data, then update it, you can check the version stamp to ensure nobody updated the data between your read and write
- ☐ Version stamps can be implemented using counters, GUIDs (a large random number that's guaranteed to be unique), content hashes, timestamps, or a combination of these.
- ☐ With distributed systems, a vector of version stamps (a set of counters, one for each node) allows you to detect when different nodes have conflicting updates.

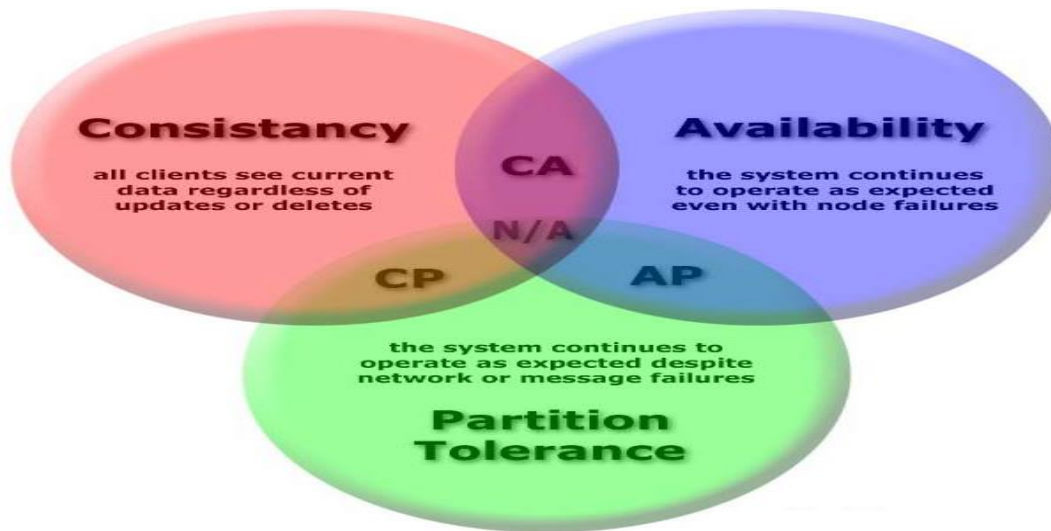
Sometimes this is called a compare-and-set (CAS) operation.

Relaxing consistency

The CAP Theorem: The basic statement of the CAP theorem is that, given the three properties of

Consistency, Availability, and Partition tolerance, you can only get two.

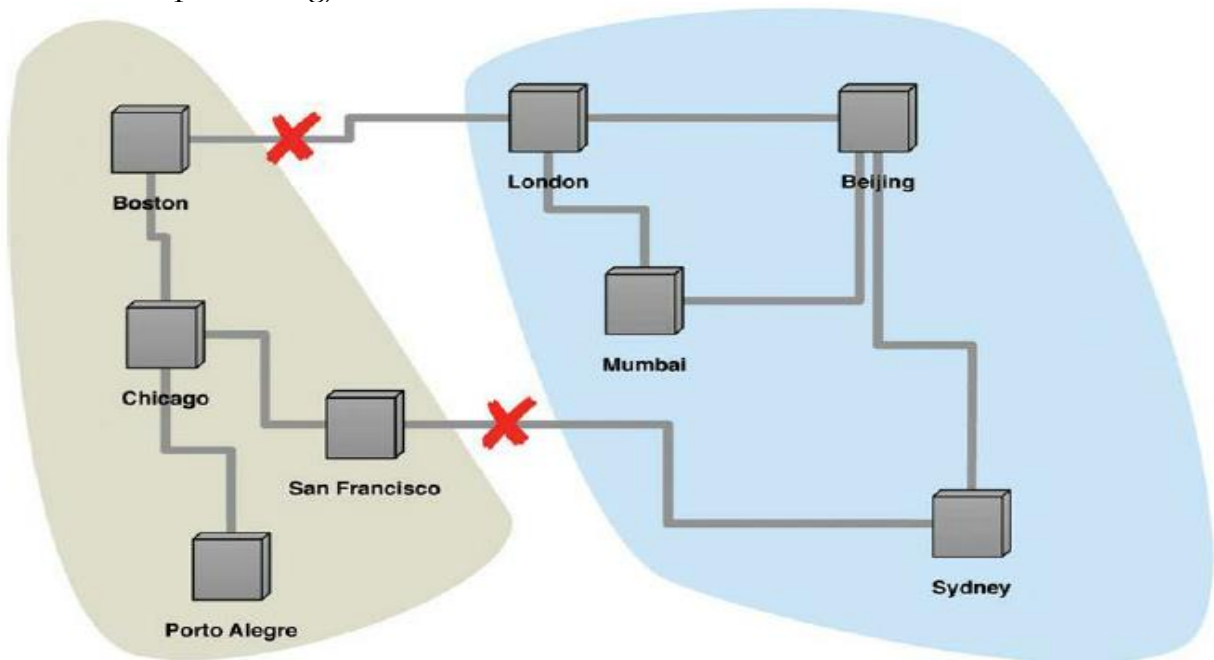
- ☐ Consistency: all people see the same data at the same time
- ☐ Availability: if you can talk to a node in the cluster, it can read and write data
- ☐ Partition tolerance: the cluster can survive communication breakages that separate the cluster into partitions unable to communicate with each other



Network partition: The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency.

Very large systems will “partition” at some point::

- That leaves either C or A to choose from (traditional DBMS prefers C over A and P)
- In almost all cases, you would choose A over C (except in specific applications such as order processing)



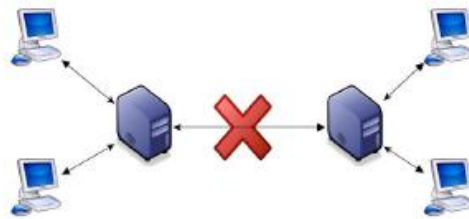
CA systems

- A single-server system is the obvious example of a CA system

- CA cluster: if a partition occurs, all the nodes would go down
- A failed, unresponsive node doesn't infer a lack of CAP availability
- A system that suffer partitions: tradeoff consistency VS availability
- Give up to some consistency to get some availability

An example

- Ann is trying to book a room of the Ace Hotel in New York on a node located in London of a booking system
- Pathin is trying to do the same on a node located in Mumbai
- The booking system uses a peer-to-peer distribution
- There is only a room available
- The network link breaks



Possible solutions

- CP: Neither user can book any hotel room, sacrificing availability
- caP: Designate Mumbai node as the master for Ace hotel
- Pathin can make the reservation
- Ann can see the inconsistent room information
- Ann cannot book the room
- AP: both nodes accept the hotel reservation
- Overbooking!

Map-Reduce

It is a way to take a big task and divide it into discrete tasks that can be done in parallel. A common use case for Map/Reduce is in document database .

A MapReduce program is composed of a **Map()** procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce()** procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

"Map" step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level [tree](#) structure. The worker node processes the smaller problem, and passes the answer back to its master node.

"Reduce" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve.

Logical view

The *Map* function is applied in parallel to every pair in the input dataset. This produces a list of pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, creating one group for each key.

`Map(k1,v1) → list(k2,v2)`

The *Reduce* function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

`Reduce(k2, list (v2)) → list(v3)`

Each *Reduce* call typically produces either one value *v3* or an empty return

Example 1: Counting and Summing

Problem Statement: There is a number of documents where each document is a set of words. It is required to calculate a total number of occurrences of each word in all documents. For instance, there is a log file where each record contains a response time and it is required to calculate an average response time.

Solution:

Let start with simple. The code snippet below shows Mapper that simply emit “1” for each word it processes and Reducer that goes through the lists of ones and sum them up:

```
class Mapper
  method Map(docid id, doc d)
    for all word t in doc d do
      Emit(word t, count 1)
```



```

class Reducer
  method Reduce(word t, counts [c1, c2,...])
    sum = 0
    for all count c in [c1, c2,...] do
      sum = sum + c
    Emit(word t, sum)

```

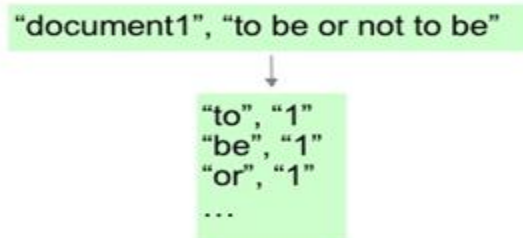
Here, each document is split into words, and each word is counted by the map function, using the word as the result key. The framework puts together all the pairs with the same key and feeds them to the same call to reduce. Thus, this function just needs to sum all of its input values to find the total appearances of that word.

Example 2:

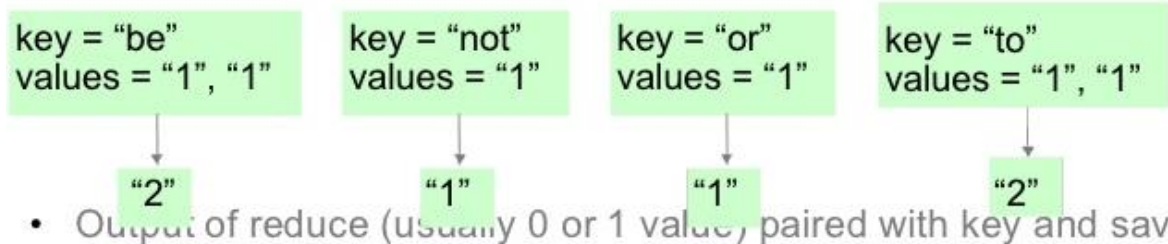
Word Frequencies in Web Pages

A typical exercise for a new engineer in his or her first week

- Input is files with one document per record
- Specify a *map* function that takes a key/value pair
key = document URL
value = document contents
- Output of map function is (potentially many) key/value pairs.
In our case, output (word, "1") once per word in the document



- MapReduce library gathers together all pairs with the same key (shuffle/sort)
- The *reduce* function combines the values for a key
In our case, compute the sum



"be", "2"
"not", "1"
"or", "1"
"to", "2"

Pseudo-code

```
Map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
for each word w in input_values:
EmitIntermediate(w, "1");
Reduce(String key, Iterator intermediate_values):
// key: a word, same for input and output
// intermediate_values: a list of counts
int result = 0;
for each v in intermediate_values:
result += ParseInt(v);
Emit(AsString(result));
```

Multistage map-reduce calculations

Let us say that we have a set of documents and its attributes with the following form:

```
{
  "type": "post",
```

```
"name": "Raven's Map/Reduce functionality",
"blog_id": 1342,
"post_id": 29293921,
"tags": ["raven", "nosql"],
"post_content": "<p>...</p>",
"comments": [
  {
    "source_ip": '124.2.21.2',
    "author": "martin",
    "text": "excellent blog..."
  }
]
```

And we want to answer a question over more than a single document. That sort of operation requires us to use aggregation, and over large amount of data, that is best done using Map/Reduce, to split the work.

Map / Reduce is just a pair of functions, operating over a list of data. Let us say that we want to be about to get a count of comments per blog. We can do that using the following Map / Reduce queries:

```
from post in docs.posts
select new {
  post.blog_id,
  comments_length = comments.length
};
```

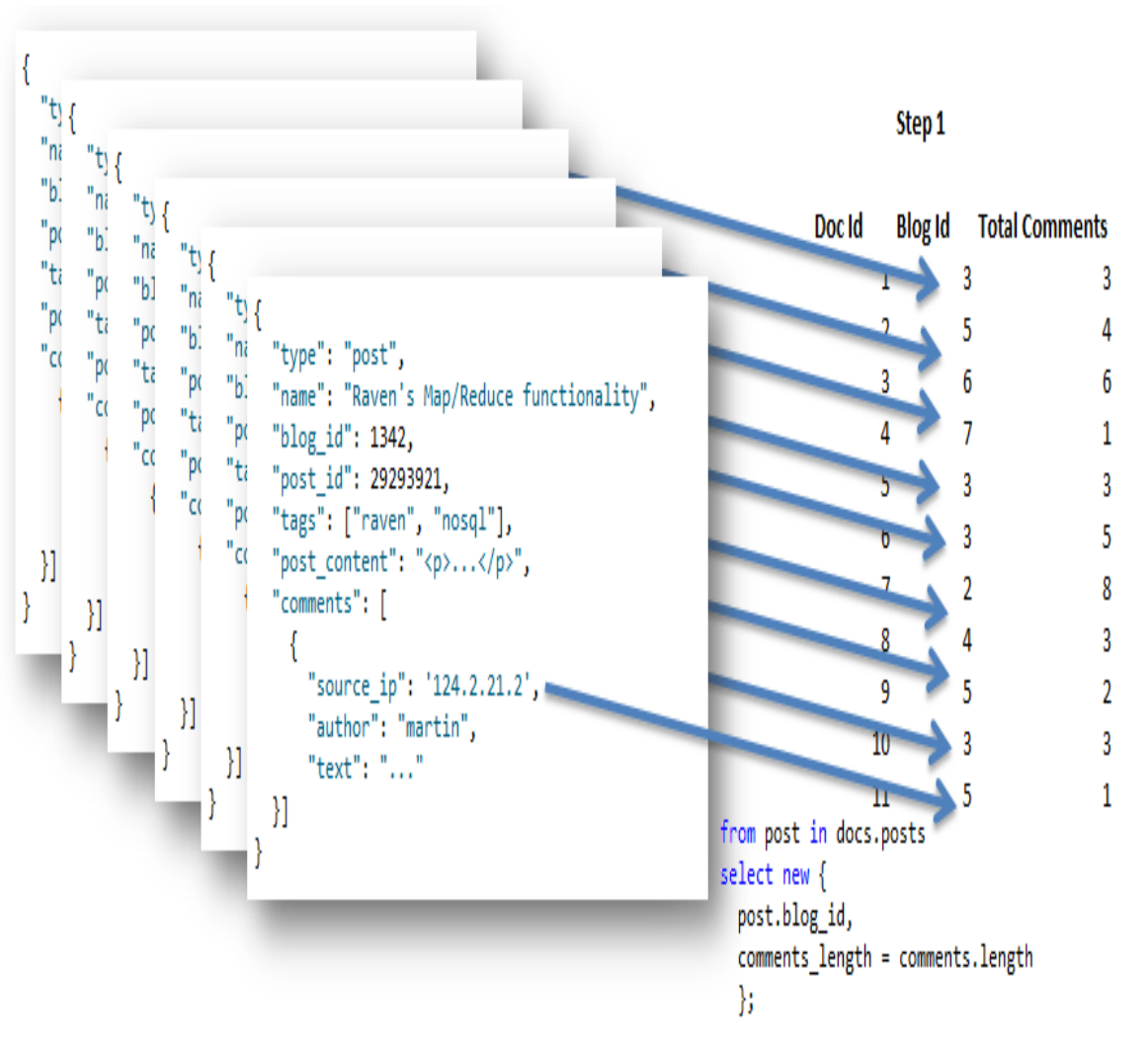
```
from agg in results
group agg by agg.key into g
select new {
  agg.blog_id,
```

```
comments_length = g.Sum(x=>x.comments_length)
};
```

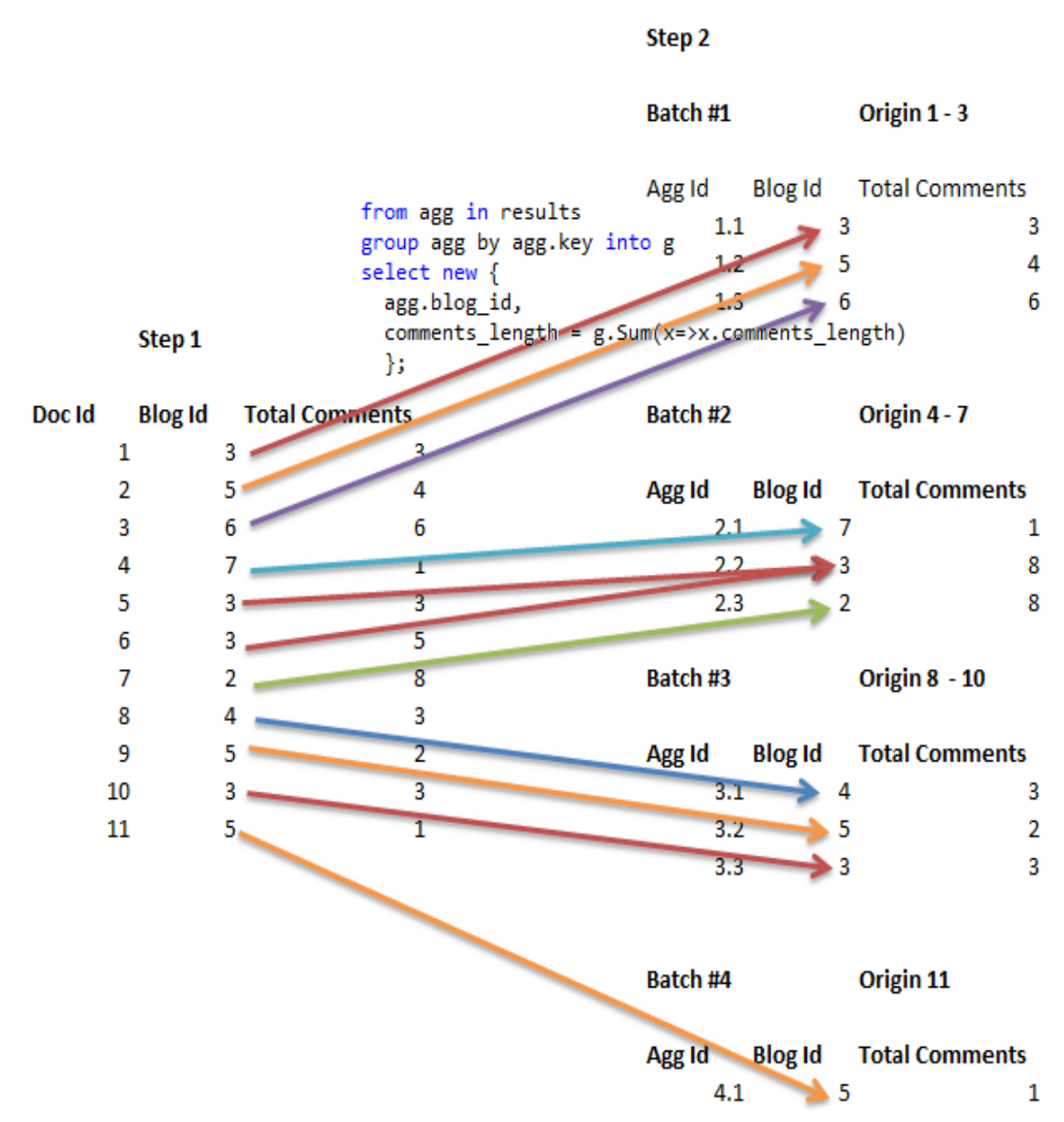
There are a couple of things to note here:

- The first query is the map query, it maps the input document into the final format.
- The second query is the reduce query, it operate over a set of results and produce an answer.
- The first value in the result is the key, which is what we are aggregating on (think the group by clause in SQL).

Let us see how this works, we start by applying the map query to the set of documents that we have, producing this output:



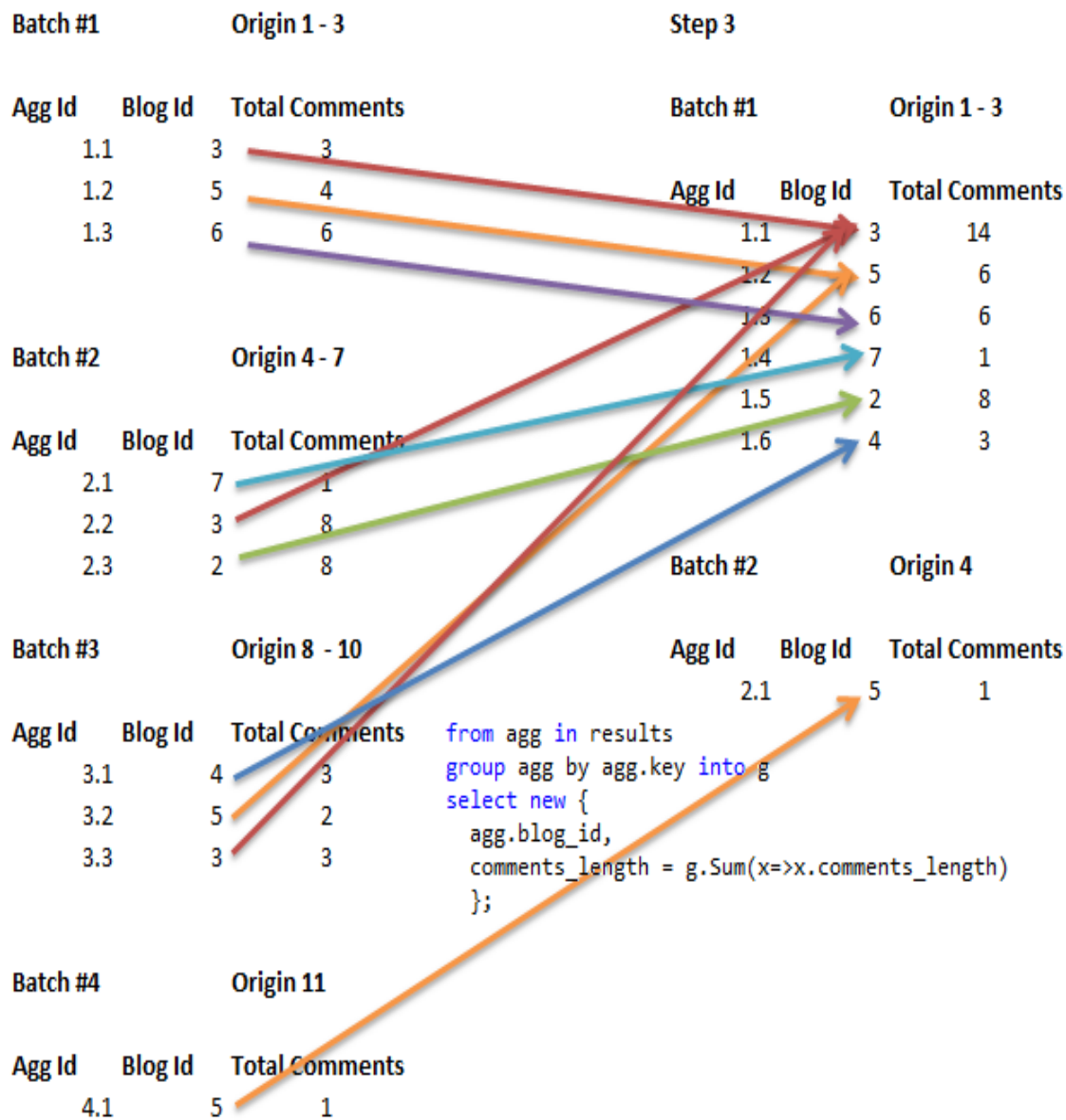
The next step is to start reducing the results, in real Map/Reduce algorithms, we partition the original input, and work toward the final result. In this case, imagine that the output of the first step was divided into groups of 3 (so 4 groups overall), and then the reduce query was applied to it, giving us:



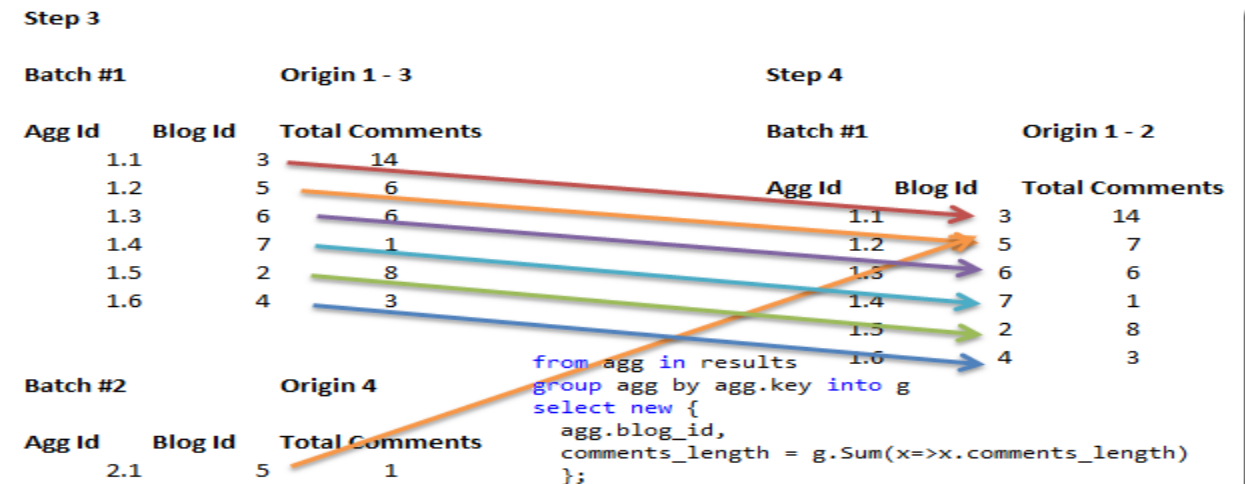
You can see why it was called reduce, for every batch, we apply a sum by blog_id to get a new Total Comments value. We started with 11 rows, and we ended up with just 10. That is where it gets interesting, because we are still not done, we can still reduce the data further.

This is what we do in the third step, reducing the data further still. That is why the input & output format of the reduce query must match, we will feed the output of several the reduce queries as the input of a new one. You can also see that now we moved from having 10 rows to have just 7.

Step 2



And the final step is:



And now we are done, we can't reduce the data any further because all the keys are unique.

RDBMS compared to MapReduce

MapReduce is a good fit for problems that need to analyze the whole dataset, in a batch fashion, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the data is written once, and read many times, whereas a relational database is good for datasets that are continually updated.

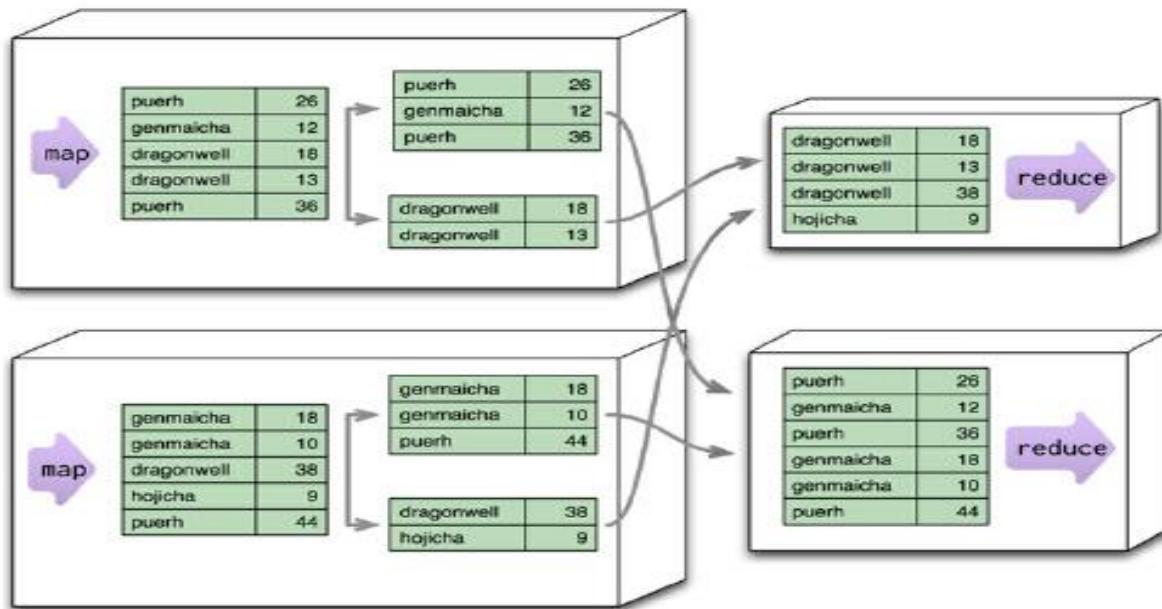
RDBMS compared to MapReduce

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Partitioning and Combining

In the simplest form, we think of a map-reduce job as having a single reduce function. The outputs from all the map tasks running on the various nodes are concatenated together and sent into the reduce. While this will work, there are things we can do to increase the parallelism and to reduce the data transfer(see figure)

Reduce Partitioning Example

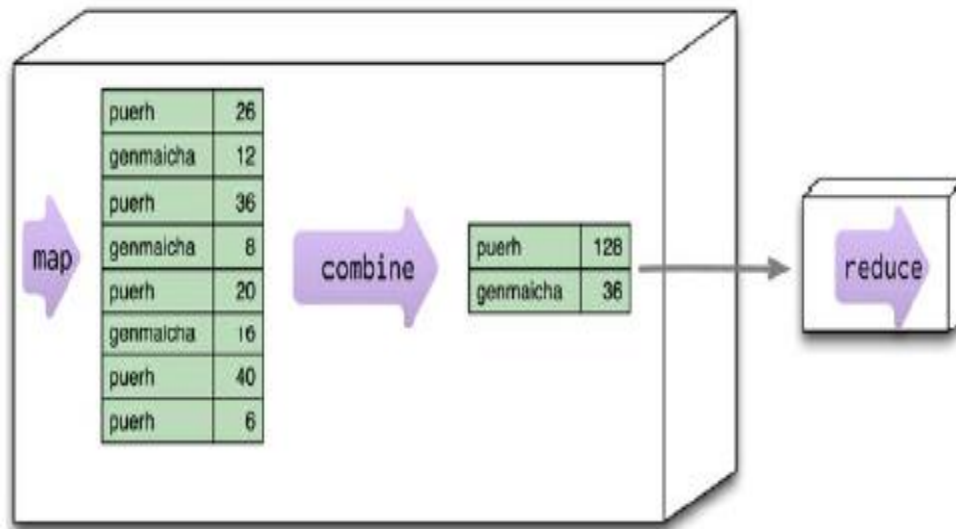


Partitioning allows reduce functions to run in parallel on different keys.

The first thing we can do is increase parallelism by partitioning the output of the mappers. Each reduce function operates on the results of a single key. This is a limitation—it means you can't do anything in the reduce that operates across keys—but it's also a benefit in that it allows you to run multiple reducers in parallel. To take advantage of this, the results of the mapper are divided up based the key on each processing node. Typically, multiple keys are grouped together into partitions. The framework then takes the data from all the nodes for one partition, combines it into a single group for that partition, and sends it off to a reducer. Multiple reducers can then operate on the partitions in parallel, with the final results merged together. (This step is also called "shuffling," and the partitions are sometimes referred to as "buckets" or "regions.")

The next problem we can deal with is the amount of data being moved from node to node between the map and reduce stages. Much of this data is repetitive, consisting of multiple key-value pairs for the same key. A combiner function cuts this data down by combining all the data for the same key into a single value(see fig)

Combinable Reducer Example



Combining reduces data before sending it across the network

A combiner function is, in essence, a reducer function – indeed, in many cases the same function can be used for combining as the final reduction. The reduce function needs a special shape for this to work: Its output must match its input. We call such a function a combinable reducer.

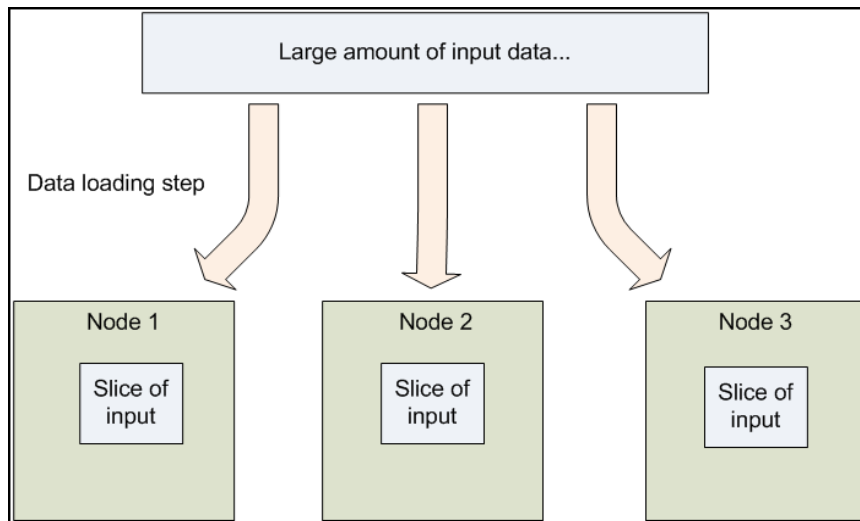
UNIT III BASICS OF HADOOP

Data format - analyzing data with Hadoop - scaling out - Hadoop streaming - Hadoop pipes - design of Hadoop distributed file system (HDFS) - HDFS concepts

Introduction to Hadoop

Performing computation on large volumes of data has been done before, usually in a distributed setting. What makes Hadoop unique is its simplified programming model which allows the user to quickly write and test distributed systems, and its efficient, automatic distribution of data and work across machines and in turn utilizing the underlying parallelism of the CPU cores.

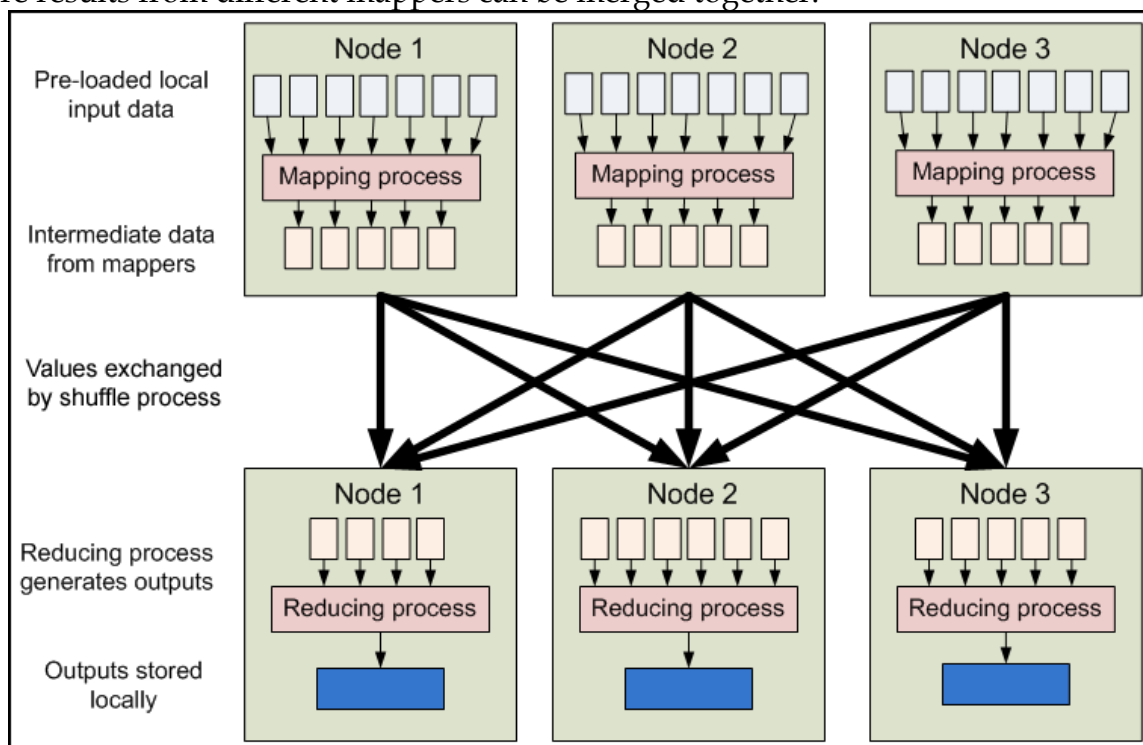
In a Hadoop cluster, data is distributed to all the nodes of the cluster as it is being loaded in. The Hadoop Distributed File System (HDFS) will split large data files into chunks which are managed by different nodes in the cluster. In addition to this each chunk is replicated across several machines, so that a single machine failure does not result in any data being unavailable. An active monitoring system then re-replicates the data in response to system failures which can result in partial storage. Even though the file chunks are replicated and distributed across several machines, they form a single namespace, so their contents are universally accessible.



MAPREDUCE in Hadoop

Hadoop limits the amount of communication which can be performed by the processes, as each individual record is processed by a task in isolation from one another. While this sounds like a major limitation at first, it makes the whole framework much more reliable. Hadoop will not run just any program and distribute it across a cluster. Programs must be written to conform to a particular programming model, named "MapReduce."

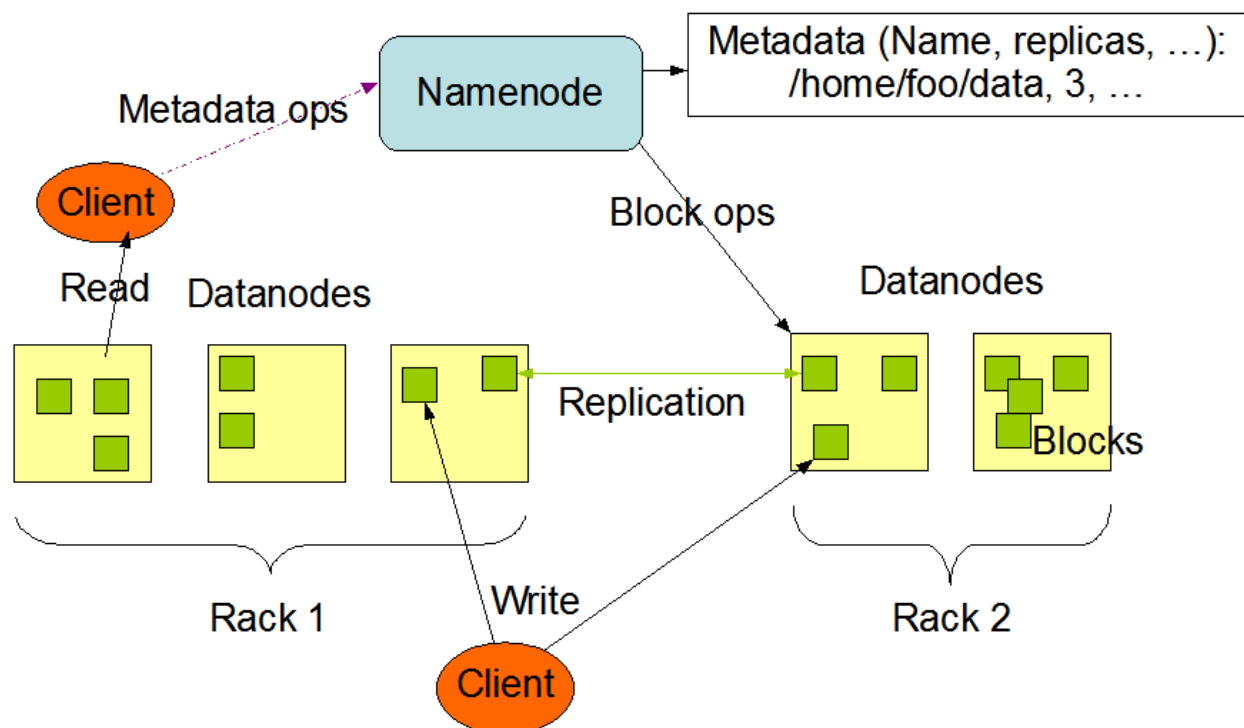
In MapReduce, records are processed in isolation by tasks called Mappers. The output from the Mappers is then brought together into a second set of tasks called Reducers, where results from different mappers can be merged together.



Hadoop Architecture:

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

HDFS Architecture



The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not

preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.

Data Format

InputFormat: How these input files are split up and read is defined by the InputFormat. An InputFormat is a class that provides the following functionality:

- Selects the files or other objects that should be used for input
- Defines the InputSplits that break a file into tasks
- Provides a factory for RecordReader objects that read the file

InputFormat:	Description:	Key:	Value:
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into key, val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

InputFormats provided by MapReduce

OutputFormat: The (key, value) pairs provided to this OutputCollector are then written to output files.

OutputFormat:	Description
TextOutputFormat	Default; writes lines in "key \t value" form
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Disregards its inputs

OutputFormats provided by Hadoop

Hadoop can process many different types of data formats, from flat text files to databases. If it is flat file, the data is stored using a line-oriented ASCII format, in which each line is a record. For example, (National Climatic Data Center) NCDC data as given below, the format supports a rich set of meteorological elements, many of which are optional or with variable data lengths.

Format of a National Climate Data Center record

```
0057
332130 # USAF weather station identifier
99999 # WBAN weather station identifier
19500101 # observation date
0300 # observation time
4
+51317 # latitude (degrees x 1000)
+028783 # longitude (degrees x 1000)
FM-12
+0171 # elevation (meters)
99999
V020
320 # wind direction (degrees)
1 # quality code
N
0072
1
00450 # sky ceiling height (meters)
1 # quality code
C
N
010000 # visibility distance (meters)
1 # quality code
N
9
-0128 # air temperature (degrees Celsius x 10)
1 # quality code
-0139 # dew point temperature (degrees Celsius x 10)
1 # quality code
10268 # atmospheric pressure (hectopascals x 10)
1 # quality code
```

Data files are organized by date and weather station.

Analyzing the Data with Hadoop

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job.

Map and Reduce:

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function.

The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, since these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing,

suspect, or erroneous.

To visualize the way the map works, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):

```
0067011990999991950051507004...9999999N9+00001+9999999999...
0043011990999991950051512004...9999999N9+00221+9999999999...
0043011990999991950051518004...9999999N9-00111+9999999999...
0043012650999991949032412004...0500001N9+01111+9999999999...
0043012650999991949032418004...0500001N9+00781+9999999999...
```

These lines are presented to the map function as the key-value pairs:

```
(0, 0067011990999991950051507004...9999999N9+00001+9999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+9999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+9999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+9999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+9999999999...)
```

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

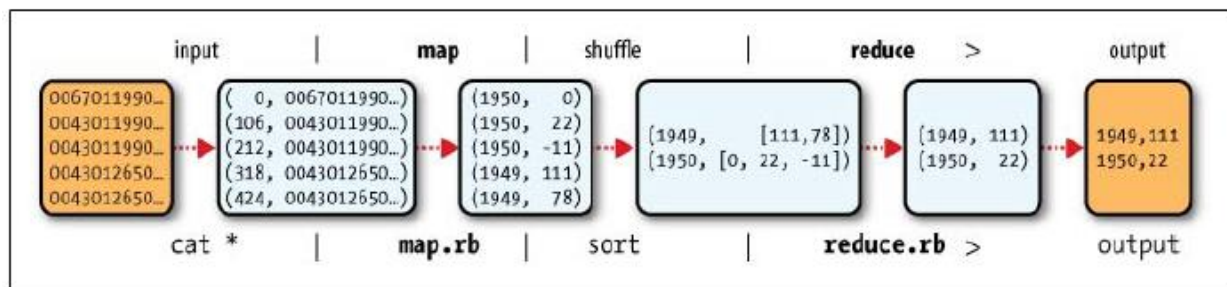
The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:

```
(1949, [111, 78])
(1950, [0, 22, -11])
```

Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

```
(1949, 111)
(1950, 22)
```

This is the final output: the maximum global temperature recorded in each year. The whole data flow is illustrated in the following figure.



Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code. We need three things: a map function, a reduce function, and some code to run the job. The map function is represented by an implementation of the Mapper interface, which declares a map() method.

```
public class MaxTemperatureMapper extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable>
{
    public void map(LongWritable key, Text value, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException
    {
        // statement to convert the input data into string
        // statement to obtain year and temp using the substring method
        // statement to place the year and temp into a set called OutputCollector
    }
}
```

The Mapper interface is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. The map() method is passed a key and a value. We convert the Text value containing the line of input into a Java String, then use its substring() method to extract the columns we are interested in. The map() method also provides an instance of OutputCollector to write the output to.

The reduce function is similarly defined using a Reducer, as illustrated in following figure.

```
public class MaxTemperatureReducer extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable>
{
    public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws
IOException
    {
        // statement to find the maximum temperature of a each year
        // statement to put the max. temp and its year in a set called
        OutputCollector
    }
}
```

The third piece of code runs the MapReduce job.

```
public class MaxTemperature
{
```

```

    public static void main(String[] args) throws IOException
    {
        JobConf conf = new JobConf(MaxTemperature.class);
        conf.setJobName("Max temperature");
        FileInputFormat.addInputPath(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        conf.setMapperClass(MaxTemperatureMapper.class);
        conf.setReducerClass(MaxTemperatureReducer.class);
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        JobClient.runJob(conf);
    }
}

```

A JobConf object forms the specification of the job. It gives you control over how the job is run. Having constructed a JobConf object, we specify the input and output paths. Next, we specify the map and reduce types to use via the setMapperClass() and setReducerClass() methods. The setOutputKeyClass() and setOutputValueClass() methods control the output types for the map and the reduce functions. The static runJob() method on JobClient submits the job and waits for it to finish, writing information about its progress to the console.

Scaling out

You've seen how MapReduce works for small inputs; now it's time to take a bird's-eye view of the system and look at the data flow for large inputs. For simplicity, the examples so far have used files on the local filesystem. However, to scale out, we need to store the data in a distributed filesystem, typically HDFS.

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks.

There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.

Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the userdefined map function for each record in the split.

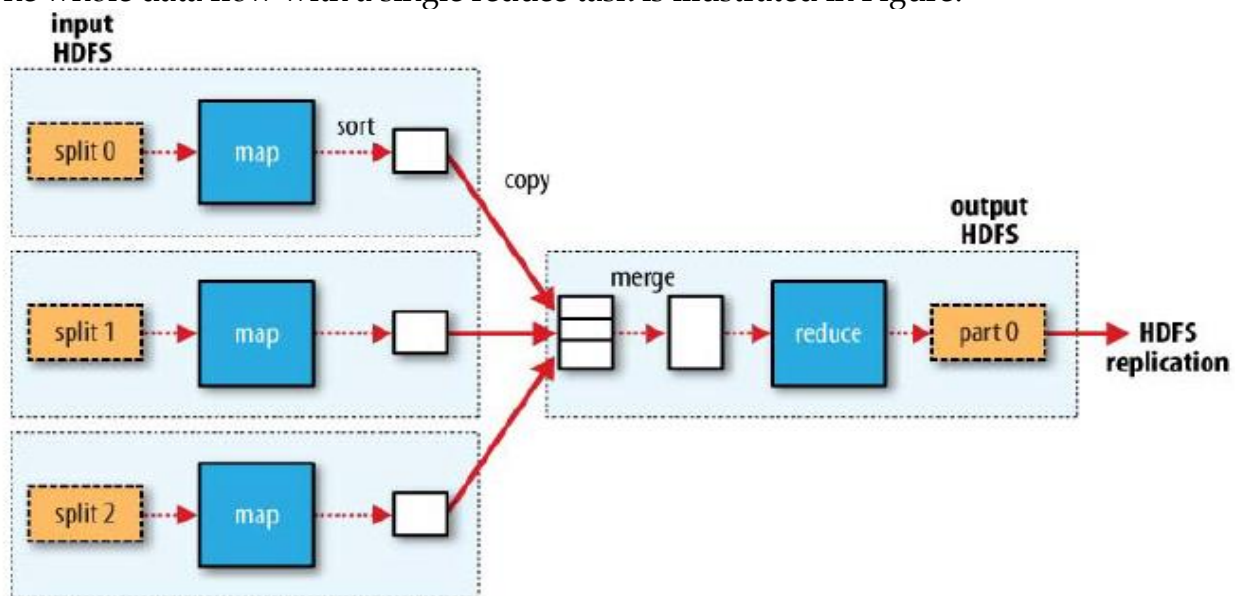
Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be

able to process proportionally more splits over the course of the job than a slower machine.

Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization.

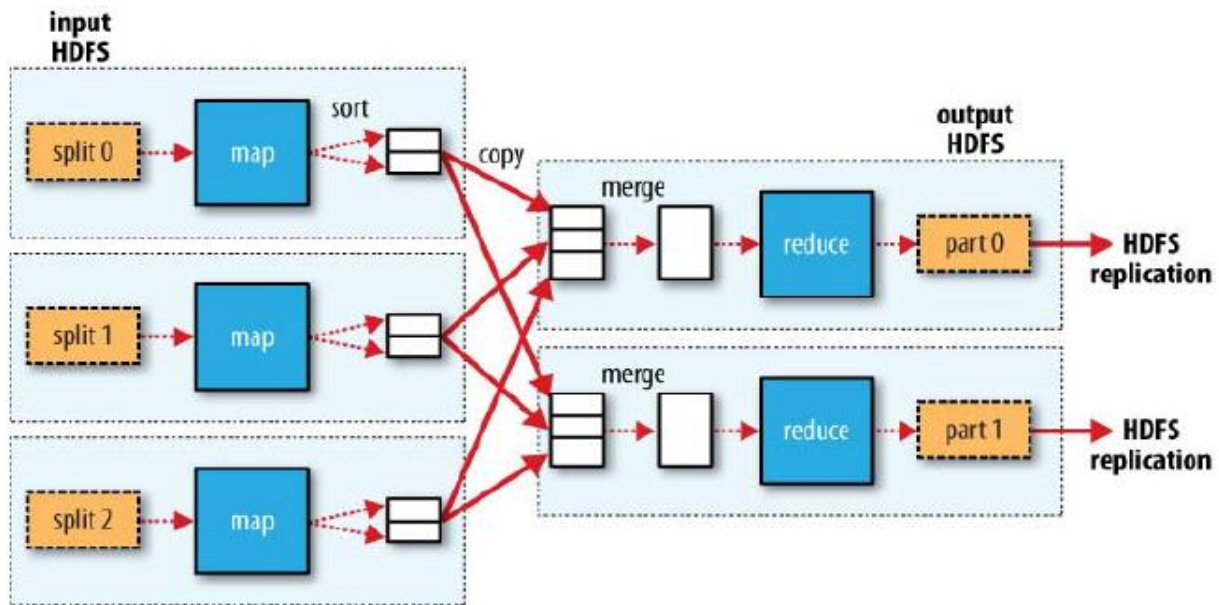
The whole data flow with a single reduce task is illustrated in Figure.



MapReduce data flow with a single reduce task

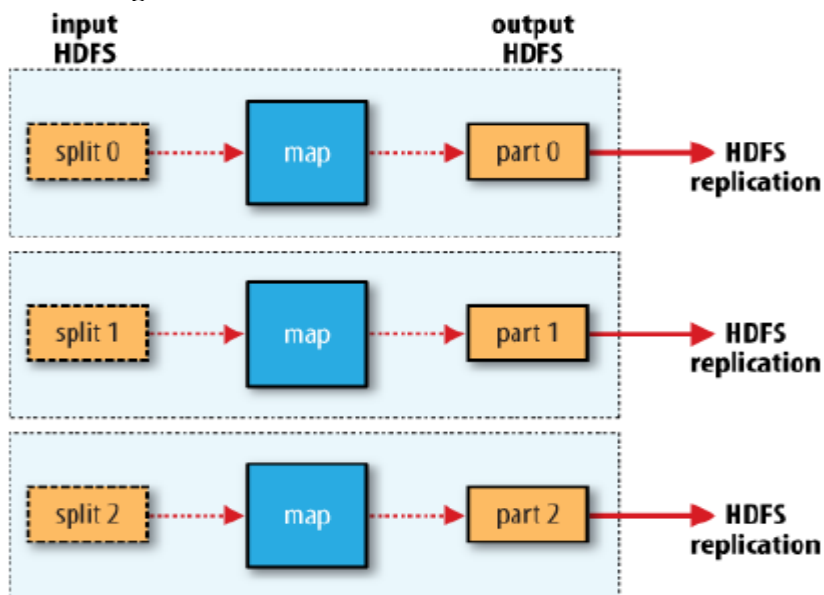
The number of reduce tasks is not governed by the size of the input, but is specified independently. When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure.



MapReduce data flow with multiple reduce tasks

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle since the processing can be carried out entirely in parallel as shown in figure.



MapReduce data flow with no reduce tasks

Combiner Functions

Hadoop allows the user to specify a combiner function to be run on the map output—the combiner function's output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words,

calling the combiner function zero, one, or many times should produce the same output from the reducer.

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

(1950, 0)

(1950, 20)

(1950, 10)

And the second produced:

(1950, 25)

(1950, 15)

The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])

with output:

(1950, 25)

since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce would then be called with:

(1950, [20, 25])

and the reduce would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:

$\text{max}(0, 20, 10, 25, 15) = \text{max}(\text{max}(0, 20, 10), \text{max}(25, 15)) = \text{max}(20, 25) = 25$

```
public class MaxTemperatureWithCombiner
{
    public static void main(String[] args) throws IOException
    {
        JobConf conf = new JobConf(MaxTemperatureWithCombiner.class);
        conf.setJobName("Max temperature");
        FileInputFormat.addInputPath(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        conf.setMapperClass(MaxTemperatureMapper.class);
        conf.setCombinerClass(MaxTemperatureReducer.class);
        conf.setReducerClass(MaxTemperatureReducer.class);
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        JobClient.runJob(conf);
    }
}
```

Hadoop Streaming

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. Hadoop Streaming uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

Streaming is naturally suited for text processing and when used in text mode, it has a line-oriented view of data. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line.

Let's illustrate this by rewriting our MapReduce program for finding maximum temperatures by year in Streaming.

Ruby: The map function can be expressed in Ruby as shown below

```
STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]
  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

Since the script just operates on standard input and output, it's trivial to test the script without using Hadoop, simply using Unix pipes:

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb
1950 +0000
1950 +0022
1950 -0011
1949 +0111
1949 +0078
```

The reduce function shown below

```
last_key, max_val = nil, 0
STDIN.each_line do |line|
  key, val = line.split("\t")
  if last_key && last_key != key
    puts "#{last_key}\t#{max_val}"
    last_key, max_val = key, val.to_i
  else
    last_key, max_val = key, [max_val, val.to_i].max
  end
end
```

```
end
puts "#{last_key}\\t#{max_val}" if last_key
```

We can now simulate the whole MapReduce pipeline with a Unix pipeline, we get

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb | \
sort | ch02/src/main/ruby/max_temperature_reduce.rb
1949 111
1950 22
```

Hadoop Pipes

Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce. Unlike Streaming, which uses standard input and output to communicate with the map and reduce code, Pipes uses sockets as the channel over which the tasktracker communicates with the process running the C++ map or reduce function.

The following example shows the source code for the map and reduce functions in C++.

```
class MaxTemperatureMapper : public HadoopPipes::Mapper
{
    public:
        MaxTemperatureMapper(HadoopPipes::TaskContext& context) { }
        void map(HadoopPipes::MapContext& context)
        {
            // statement to convert the input data into string
            // statement to obtain year and temp using the substring method
            // statement to place the year and temp into a set
        }

};

class MapTemperatureReducer : public HadoopPipes::Reducer
{
    public:
        MapTemperatureReducer(HadoopPipes::TaskContext& context) {}
        void reduce(HadoopPipes::ReduceContext& context)
        {
            // statement to find the maximum temperature of a each year
            // statement to put the max. temp and its year in a set
        }

};

int main(int argc, char *argv[])
```

```
{
return HadoopPipes::runTask( HadoopPipes:: TemplateFactory < MaxTemperatureMapper,
MapTemperatureReducer>());
}
```

The map and reduce functions are defined by extending the Mapper and Reducer classes defined in the HadoopPipes namespace and providing implementations of the map() and reduce() methods in each case.

These methods take a context object (of type MapContext or ReduceContext), which provides the means for reading input and writing output, as well as accessing job configuration information via the JobConf class.

The main() method is the application entry point. It calls HadoopPipes::runTask, which connects to the Java parent process and marshals data to and from the Mapper or Reducer. The runTask() method is passed a Factory so that it can create instances of the Mapper or Reducer. Which one it creates is controlled by the Java parent over the socket connection. There are overloaded template factory methods for setting a combiner, partitioner, record reader, or record writer.

Pipes doesn't run in standalone (local) mode, since it relies on Hadoop's distributed cache mechanism, which works only when HDFS is running.

With the Hadoop daemons now running, the first step is to copy the executable to HDFS so that it can be picked up by tasktrackers when they launch map and reduce tasks:

```
% hadoop fs -put max_temperature bin/max_temperature
```

The sample data also needs to be copied from the local filesystem into HDFS:

```
% hadoop fs -put input/ncdc/sample.txt sample.txt
```

Now we can run the job. For this, we use the Hadoop pipes command, passing the URI of the executable in HDFS using the -program argument:

```
% hadoop pipes \
-D hadoop.pipes.java.recordreader=true \
-D hadoop.pipes.java.recordwriter=true \
-input sample.txt \
-output output \
-program bin/max_temperature
```

The result is the same as the other versions of the same program that we ran previous example.

Design of HDFS

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

- "Very large"

In this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

- Streaming data access

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

- Commodity hardware

Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

- Low-latency data access

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency

- Lots of small files

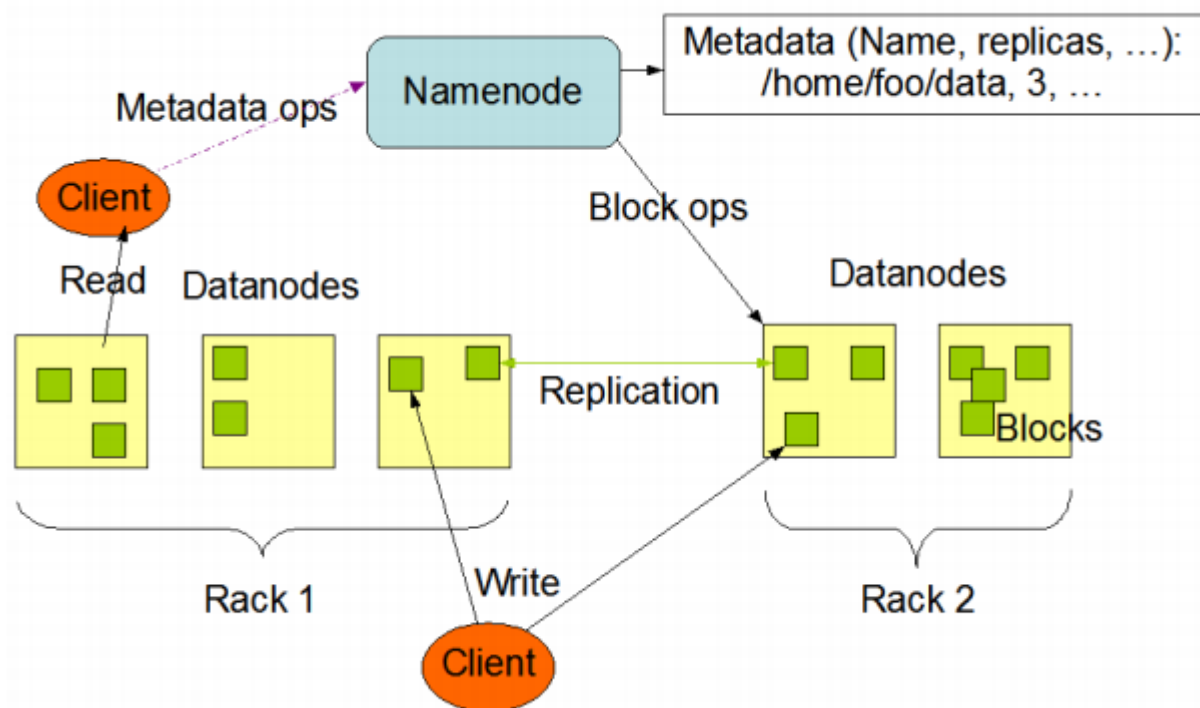
Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware.

- Multiple writers, arbitrary file modifications

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file

HDFS Concepts

The following diagram illustrates the Hadoop concepts



Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes. HDFS, too, has the concept of a block, but it is a much larger unit – 64 MB by default.

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

Benefits of blocks

1. A file can be larger than any single disk in the network. There is nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster.
2. It simplifies the storage subsystem. The storage subsystem deals with blocks, simplifying storage management and eliminating metadata concerns.
3. Blocks fit well with replication for providing fault tolerance and availability.

Namenodes and Datanodes

An HDFS cluster has two types of node operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A client accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to, and they report back to the namenode periodically with lists of blocks that they are storing. Without the namenode, the filesystem cannot be used.

Secondary Namenode

It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

The Command-Line Interface

There are many other interfaces to HDFS, but the command line is one of the simplest and, to many developers, the most familiar. It provides a commandline interface called FS shell that lets a user interact with the data in HDFS. The syntax of this command set is similar to other shells (e.g. bash, csh) that users are already familiar with. Here are some sample action/command pairs:

Action	Command
Create a directory named /foodir	bin/hadoop dfs -mkdir /foodir
Create a directory named /foodir	bin/hadoop dfs -mkdir /foodir
View the contents of a file named /foodir/myfile.txt	bin/hadoop dfs -cat /foodir/myfile.txt

Hadoop Filesystems

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class `org.apache.hadoop.fs.FileSystem` represents a filesystem in Hadoop, and there are several concrete implementations, which are described in Table

Filesystem	Description
Local	A filesystem for a locally connected disk with client-side checksums. Use <code>RawLocalFileSystem</code> for a local filesystem with no checksums.
HDFS	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
HFTP	A filesystem providing read-only access to HDFS over HTTP.
HSFTP	A filesystem providing read-only access to HDFS over HTTPS.
HAR	A filesystem layered on another filesystem for archiving files. Hadoop Archives are typically used for archiving files in HDFS to reduce the namenode's memory usage.
KFS (Cloud-Store)	CloudStore (formerly Kosmos filesystem) is a distributed filesystem like HDFS or Google's GFS, written in C++.
FTP	A filesystem backed by an FTP server.
S3 (native)	A filesystem backed by Amazon S3.
S3 (block-based)	A filesystem backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5-GB file size limit.

Thrift

The Thrift API in the “thriftfs” module expose Hadoop filesystems as an Apache Thrift service, making it easy for any language that has Thrift bindings to interact with a Hadoop filesystem, such as HDFS.

C

Hadoop provides a C library called `libhdfs` that mirrors the Java `FileSystem` interface (it was written as a C library for accessing HDFS, but despite its name it can be used to access any Hadoop filesystem). It works using the Java Native Interface (JNI) to call

a Java filesystem client.

FUSE

Filesystem in Userspace (FUSE) allows filesystems that are implemented in user space to be integrated as a Unix filesystem. Hadoop's Fuse-DFS contrib module allows any Hadoop filesystem (but typically HDFS) to be mounted as a standard filesystem.

WebDAV

WebDAV is a set of extensions to HTTP to support editing and updating files. WebDAV shares can be mounted as filesystems on most operating systems, so by exposing HDFS (or other Hadoop filesystems) over WebDAV, it's possible to access HDFS as a standard filesystem.

File patterns

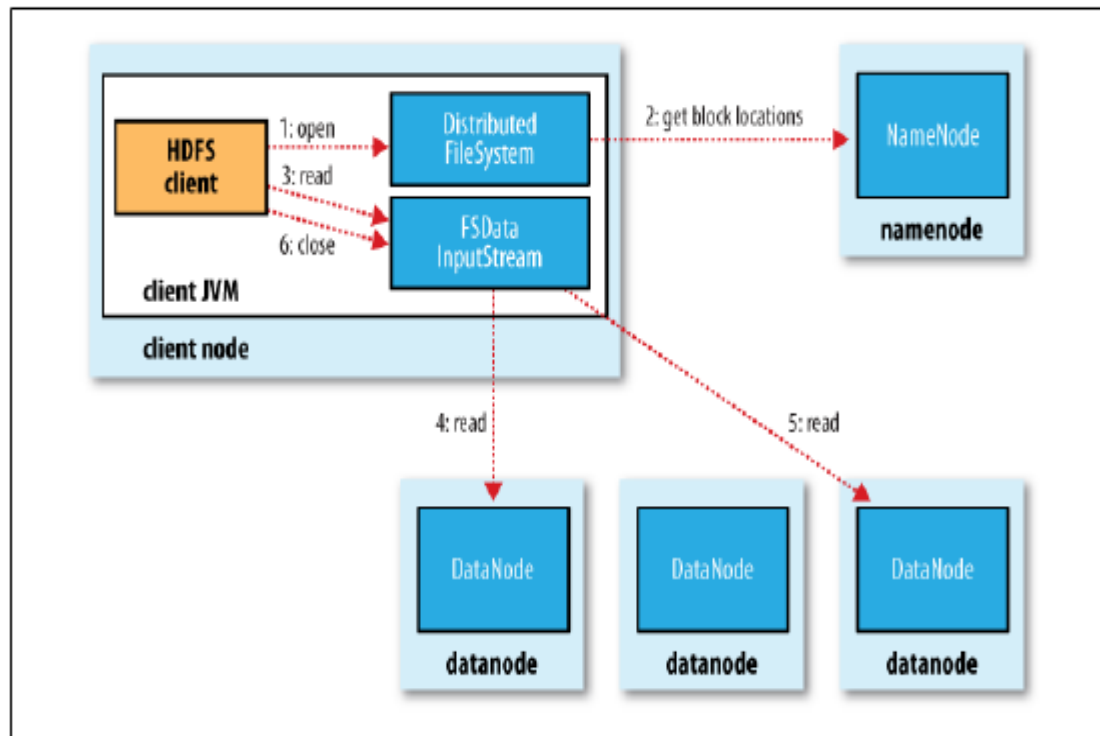
It is a common requirement to process sets of files in a single operation. Hadoop supports the same set of glob characters as Unix bash as shown in Table

Glob characters and their meanings

Glob	Name	Matches
*	<i>asterisk</i>	Matches zero or more characters
?	<i>question mark</i>	Matches a single character
[ab]	<i>character class</i>	Matches a single character in the set {a, b}
[^ab]	<i>negated character class</i>	Matches a single character that is not in the set {a, b}
[a-b]	<i>character range</i>	Matches a single character in the (closed) range [a, b], where a is lexicographically less than or equal to b
[^a-b]	<i>negated character range</i>	Matches a single character that is not in the (closed) range [a, b], where a is lexicographically less than or equal to b
{a,b}	<i>alternation</i>	Matches either expression a or b
\c	<i>escaped character</i>	Matches character c when it is a metacharacter

Anatomy of a File Read

To get an idea of how data flows between the client interacting with HDFS, the namenode and the datanodes, consider Figure, which shows the main sequence of events when reading a file.



A client reading data from HDFS

The client opens the file it wishes to read by calling `open()` on the `FileSystem` object, which for HDFS is an instance of `DistributedFileSystem` (step 1).

`DistributedFileSystem` calls the `namenode`, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2). For each block, the `namenode` returns the addresses of the `datanodes` that have a copy of that block.

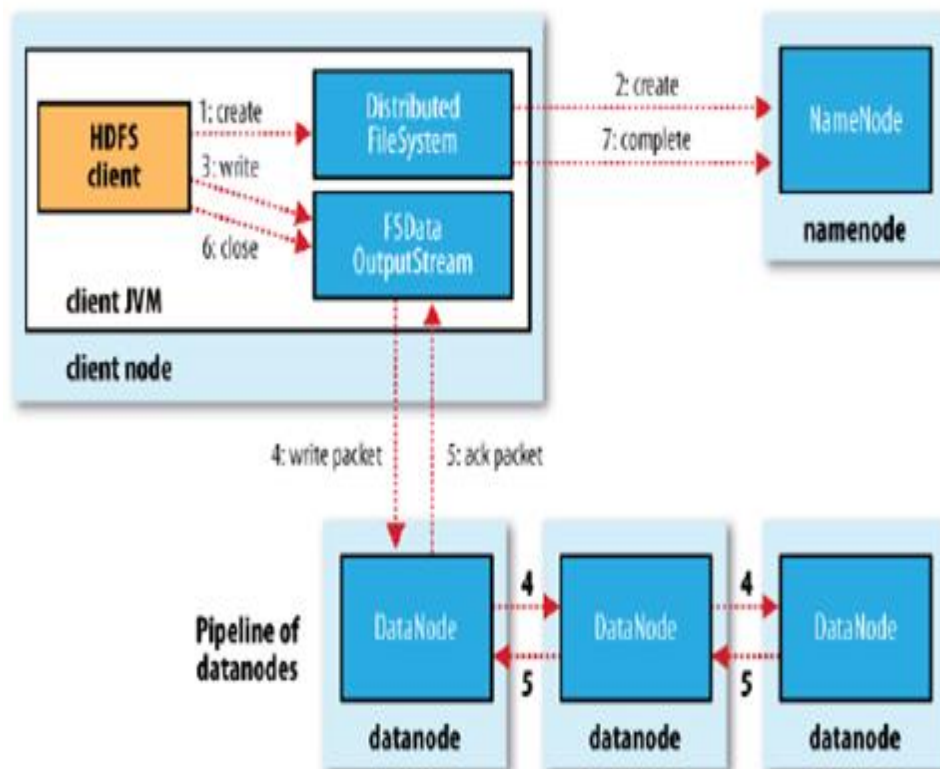
The `DistributedFileSystem` returns an `FSDataInputStream` (an input stream that supports file seeks) to the client for it to read data from. `FSDataInputStream` in turn wraps a `DFSInputStream`, which manages the `datanode` and `namenode` I/O.

The client then calls `read()` on the stream (step 3). `DFSInputStream`, which has stored the `datanode` addresses for the first few blocks in the file, then connects to the first (closest) `datanode` for the first block in the file. Data is streamed from the `datanode` back to the client, which calls `read()` repeatedly on the stream (step 4). When the end of the block is reached, `DFSInputStream` will close the connection to the `datanode`, then find the best `datanode` for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream. Blocks are read in order with the `DFSInputStream` opening new connections to `datanodes` as the client reads through the stream. It will also call the `namenode` to retrieve the `datanode` locations for the next batch of blocks as needed. When the client has finished reading, it calls `close()` on the `FSDataInputStream` (step 6).

Anatomy of a File Write

The client creates the file by calling `create()` on `DistributedFileSystem` (step 1). `DistributedFileSystem` makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist, and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an `IOException`. The `DistributedFileSystem` returns an `FSDDataOutputStream` for the client

to start writing data to. Just as in the read case, `FSDDataOutputStream` wraps a `DFSOutputStream`, which handles communication with the datanodes and namenode. As the client writes data (step 3), `DFSOutputStream` splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline – we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).



A client writing data to HDFS

DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5). If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. When the client has finished writing data, it calls `close()` on the stream (step 6).