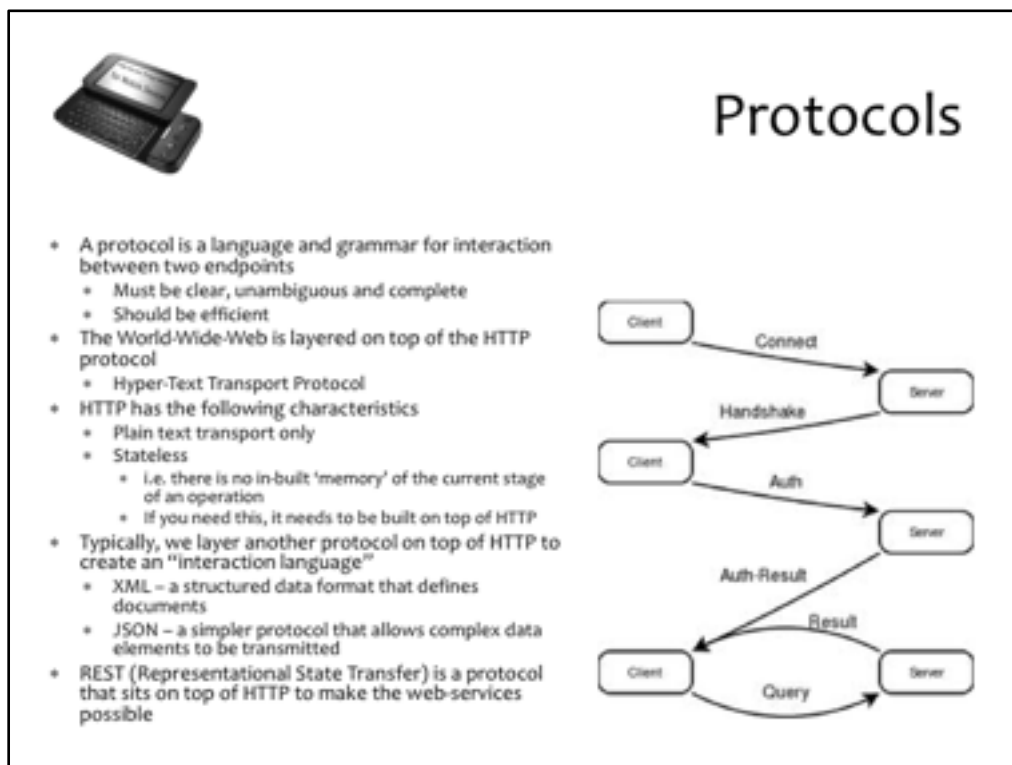


This lecture is about keeping data for mobile apps on a web-server or in the cloud. The core principle of cloud services is “location independent, high-availability” It turns out that most companies have come to the same conclusions about how to achieve this aim.



We need to start with the protocol.

How will your (mobile) application ask for information?

How will servers respond?

There are a variety of protocols available:

Some are data-centric – e.g. SQL, Windows Active Directory

Some are application centric – e.g. MS Exchange

Some are network centric – e.g. Berkeley Sockets, HTTP, WebSockets

Some are resource centric – e.g. REST (REpresentational State Transfer), SOAP (Simple Object Access Protocol)

Generally, our mobile apps would benefit from a bit of each:

Data structure that makes it easy to query data on the other hand, one that allows data to be stored in clusters on different servers


Application structure that makes it easy to retrieve data in the right sort of chunkse.g. all of one user's bank transactions, whether these range across different servers or not

Network structure to make sure that requests for data are responded to quickly this suggests a stateless protocol, for speed and so that a set of data can be organised across servers

Resource-centric (particularly REST) turns out to be a very good way to manage data in web applications.

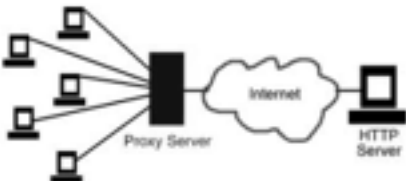
For a single, small app, we can use HTTP and a simple packaging protocol effectively
e.g. XML or JSON – document centric (may not conform to how data is stored, but who cares)
servers smart enough to assemble the information

For larger apps, we tend to need an abstraction of some sort to de-couple the data that mobile devices work with from a much more complex central data structure that copes with the data for many devices. REST has been shown to be a very good match for this (much more so than SOAP – the other way) because data items can be managed like resources by the device, regardless what form they take and because each item of data can be referenced by a simple URL – like a bookmark. All that is needed is a server to agree on the format that data will be uploaded and downloaded in.



Stateless Data Interchange

- State is the combination of all of the data variables in a system at a point in time
 - An operation that changes a variable changes the system state
- A stateless protocol is a protocol that treats each interaction independently of any other
 - HTTP communicates statelessly because there is no 'sequence' of operations
 - The server does not need to retain any information of previous operations
 - Systems built on top of HTTP can change the state of the server (e.g. by adding items to a database), but this does not affect how HTTP works
- It is important for the effectiveness of the web that HTTP is stateless
 - Otherwise, servers would always be tied up keeping track of prior operations
 - Request→Response operations are fast because of statelessness
 - Server farms rely on this



Stateless turns out to be very important

Highly distributed systems would not work well if there had to be a single, all-seeing master that knew where everything was stored

Request \leftrightarrow Response behaviour would be much messier if every request had to query a master data-store

The trick turns out to be to manage statelessness across a large cluster of nodes using a map-reduce algorithm. We don't need a master index for a database, because each node indexes its own data and can respond very quickly to "do you have this data" queries

Organize the server nodes hierarchically (or as a grid so that each node can pass on a query to several other nodes) and we can distribute a search job (almost any data retrieval) across the whole network.

Assume a big "tree" structure – trunk has branches, branches have more branches etc.

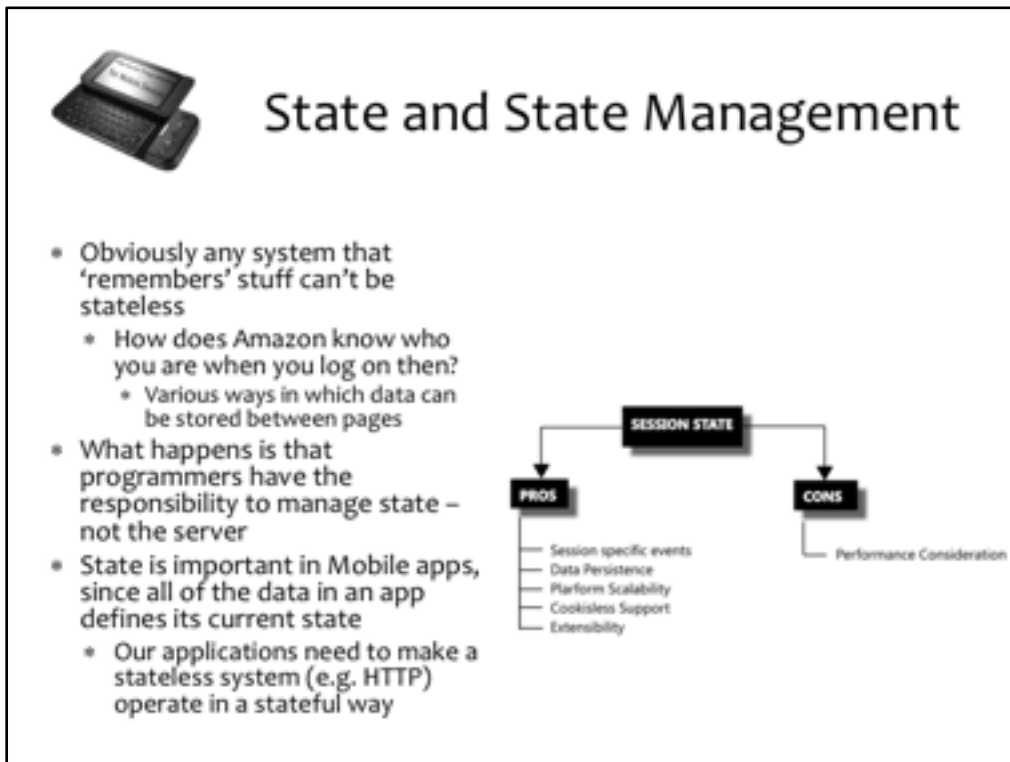
The trunk is the master node to which queries are sent – e.g.

:get me a list of elements that include the term "Big Data"

This node sends the same request out to all of the nodes it is directly attached to - some respond (fairly quickly because they actually contain some of the data) some pass the query on to nodes they are directly connected to, etc.


The requests are simple mappings – given a key value, each server simply has

to find the nodes with this key - this can be implemented in just about any technology
The responses could come from any of the range of server nodes as results are passed back
the trunk, each node that issued or re-issued a request filters out duplicate responses (the I
The set of responses that reaches the trunk is the result – a MapReduce



State and State Management

- Obviously any system that 'remembers' stuff can't be stateless
 - How does Amazon know who you are when you log on then?
 - Various ways in which data can be stored between pages
- What happens is that programmers have the responsibility to manage state – not the server
- State is important in Mobile apps, since all of the data in an app defines its current state
 - Our applications need to make a stateless system (e.g. HTTP) operate in a stateful way



```

graph TD
    SS[SESSION STATE] --> P[PROS]
    SS --> C[CONS]
    P --- P_list["Session specific events<br/>Data Persistence<br/>Platform Scalability<br/>Cookiesless Support<br/>Extensibility"]
    C --- C_list["Performance Consideration"]
  
```


The use of the HTTP stateless protocol for communications is important

If a server node receives a request, the “internal” state of a node can be checked using any of a variety of mechanisms – e.g. SQL query, simple string table search etc.

There is no ‘blocking up’ the communications bandwidth with state data. Instead, what is needed is a distributable/scalable query system within each node that can deal with simple requests very quickly

e.g. you visit Amazon.co.uk – the Cookie they planted on your system automatically identifies you, because the server node that your amazon home page request reached can use MapReduce to find your data across the entire Amazon system very quickly. All data pertaining to your current Amazon status is returned to that node and assembled (along with the usual Amazon page-furniture) as your personalized page


At no point did HTTP have to remember what you did last, because that is stored across the Amazon system’s log files(for searches), or their orders database (for purchases you’re in the middle of making). None of this data goes away – prospective sales simply turn into abandoned sales or completed sales (stick some things in your basket and then go away from the Amazon pages – the next time you go back, they will still be there). YOUR state information is distributed across a multitude of individual server nodes – data tables, log files etc. MapReduce makes it all accessible.



State and Scalability

- * The state of a session can be managed at the client OR the server
 - * Client→
 - * Cookies
 - * sessionStorage
 - * localStorage
 - * Hidden form variables
 - * Server→
 - * S-S-Session storage (e.g. ASP.NET)
- * In general, prefer client-side state management
 - * Otherwise, server is not stateless, and performance suffers
 - * In particular – **scalability**
 - * The ability of a server to respond to big increases in demand


Client-side state management



Client: I am Mary, I am shopping for books. I have three titles in my cart totaling \$36.

Web server

Server-side state management

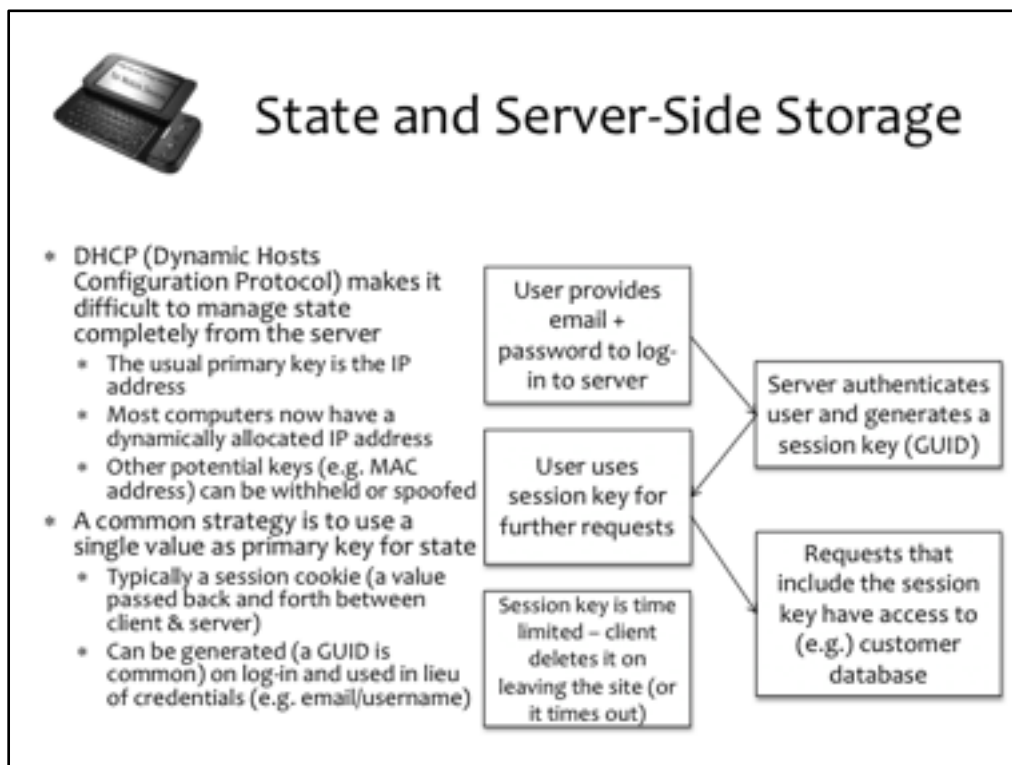


Client: I am session \$56.

Web server: Session \$56 is Mary. Mary is shopping for books. There are three titles in Mary's cart totaling \$36.

Statelessness also means that no one server (or master node or whatever) needs to keep track of a growing number of transactions. As Amazon (Google, John Lewis etc.) gains more customers, new servers can be added to cope with the increasing workload. The scalability of MapReduce means that the search operations do not need to take any longer provided the overall number of new customers is matched by an added number of servers. The system organization is scalable to a very high degree

Scalability – a system is scalable if a large increase in the number of requests (queries, transactions, searches etc.) does not result in any part of the system becoming a bottleneck. Fairly obviously, if you have a single database server (say MySQL, for example), then the more simultaneous users there are, the more work that server has to do. Eventually, the increase in users will become noticeable in the time it takes for one user to get a response. Adding a new server will only help if the workload can be shared evenly, but that is not easy to manage with a normal database, where there is a server running the database queries and other servers simply store data for the master server to locate. A scalable system would be one that distributed the queries fairly across all server nodes. This is not completely manageable, but some systems can minimize the work that the ‘master’ node or nodes have to do.



State management at the server is ideal for systems that need to exercise control
 Security
 Shopping sites (need to keep customers)

Basic principle is that state variables are stored permanently in a server-side database
 e.g. go to Amazon, start a purchase and leave the site before completing it Go back and check – stuff is still in your shopping basket. Amazon maintains your shopping basket permanently

Access is easier to manage in this way –

User logs-in with id and password, and receives a session key in response
 A GUID (Globally Unique Identifier) is normal - (basically a big random number)

The client app can store this in `sessionStorage` or `localStorage`

From now – client can move from page to page on the site and will be recognised by the GUID

At some stage the session key will become out of date. `sessionStorage` on client is deleted when the user leaves the site. `sessionStorage` on the server can be removed by a cron task (a task set to run periodically on the server)

From then, client can only continue with the site by re-logging-in



- Relational databases have long been considered the standard for structured data storage
 - Loads of history behind this topic
 - Loads of expertise in IT industry
 - Very efficient for data storage in a single (master) database)
- The model breaks down as the database size gets very large (or the database gets very distributed)
 - Petabytes across many servers
 - No guarantee that all of the data items a query refers to are on the same server
 - Data Warehouses
- SQL becomes a problem at these scales – normalized is not efficient

Relational data is based on the idea that any piece of data should be stored only ONCE.

For example, if you place an order with Amazon, the Amazon system **could** organise this by adding your name, address, credit details etc. to the order. Then if you placed a second order, it would add this same information to that as well.

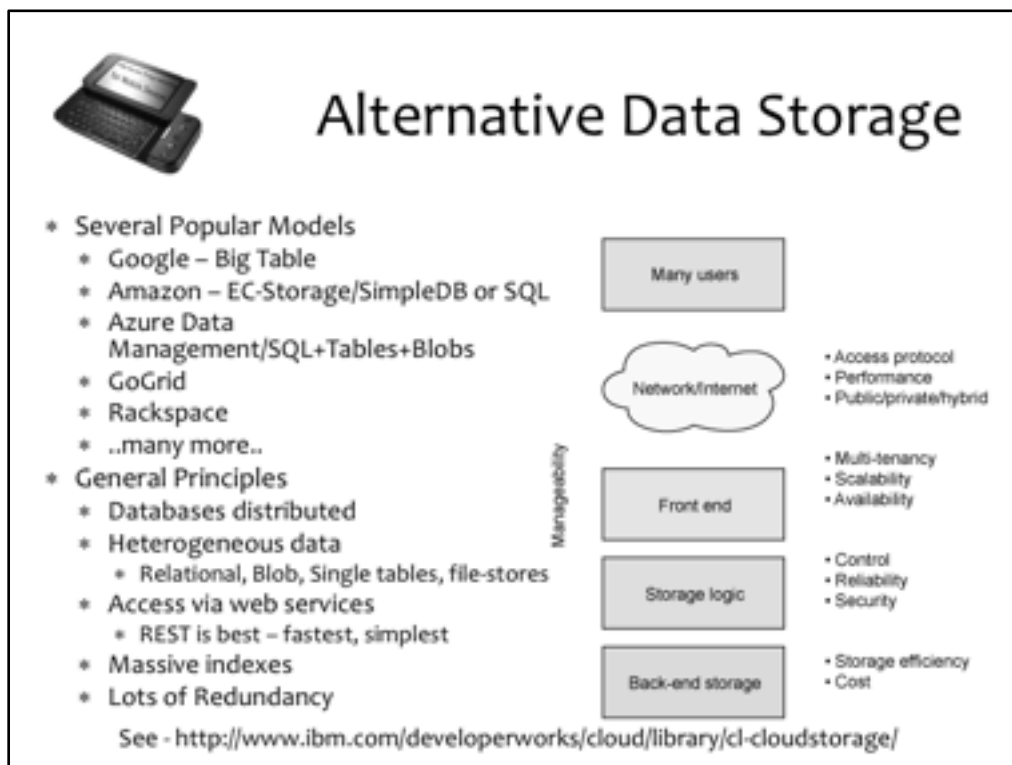
1. How does your personal info get added to the order? If you re-enter it every time, that gets tedious and you get tired of filling in the same form every time.
2. If the amazon system looked it up, then why store it alongside your order details (the book you bought)? All it needs to store is a **reference** to your personal data – e.g. your email address would work, since that is unique. Now, when you place a dozen orders, each simply refers to the bit of the Amazon database that holds your personal data
3. If your personal data WAS stored with every purchase order, Amazon would be storing a lot of duplicated info. If you changed your address or phone number, which record would we update? All of them? (that's a bit of overkill). If only one, how would the system be sure which order held the most up to date data?

Data integrity is the main reason for the relational database model. By storing a single copy of each entity (where an entity is an indivisible unit—like your full name,

or home address), there is less chance of a) getting the wrong one b) updating the wrong one), c) c by entering wrong personal data into an order.

Relational databases are often said to be *normalized*. This simply means that different tables are different types of entities (the real-world things that the data describes), with the result that there is less redundancy (storing more than one copy of a thing), and providing more certain operations. The four basic operations are CREATE, READ, UPDATE and DELETE (CRUD). In a relational system, these are easier to perform.

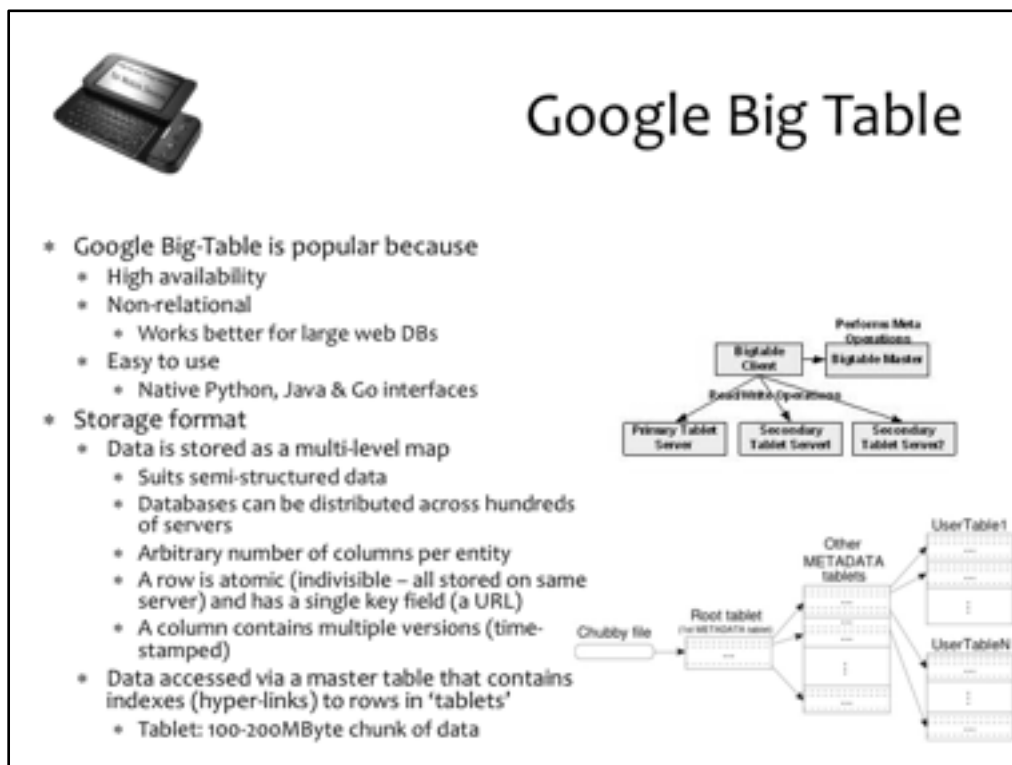
However, the relational system storing different data items on different tables can be a problem in a distributed database system. For example, your Amazon data records are likely to be stored fairly far apart (e.g. in the UK). Records of the items you bought are likely to be stored on servers close to the warehouse where they were bought. If you ask Amazon to show you the list of things you have bought, the query that retrieves this list will need to run across a lot of different databases. If each of these was an isolated copy of Oracle or SQL Server, the query could take some time. Not only that, if one server failed, it could be that the query results would be wrong. For this reason, the relational model breaks down over large distributed systems. What is the solution? A degree of data redundancy, and less splitting-up of the related data elements. However, the redundancy creates a data integrity problem – if there are two copies of your customer record, and they both contain different information, which one is correct? High Replication data servers (like App Engine, Amazon etc.) manage this by guaranteeing that a single key value that might exist on a number of different servers will be updated in a very short time, thus avoiding the possibility of different versions of your data placing you at different addresses, but not for long.



Since we've established that the relational database model is not a perfect fit for large, scalable web applications, what are the alternatives?

I do expect you'll be wondering "what has mobile got to do with large-scalable web apps?" Absolutely everything – almost all successful mobile apps rely on a web presence. It could be simply for distributing updated versions (for some reasons, mobile apps seem to need that a lot), but more and more apps rely on a web presence for accessing working data. For example, Evernote – one of the most popular organiser apps, is nothing without a large wider-area storage network that lets you access the same task list on any machines – phones, laptops, desktops, tablets. Even games need a storage network for in-game purchases – levels, new characters etc.

So – the models depicted here are various companies' solutions to their widely accessed, highly redundant storage mechanisms. Although they all have idiosyncrasies and (at the lowest level) very different actual storage mechanisms, they all rely on a single family of algorithms for managing widely distributed queries. The way data is actually stored (in flat files, tables, relational DBs or large binary objects) is immaterial compared to the use of MapReduce and the indexing schemes needed to make that work.

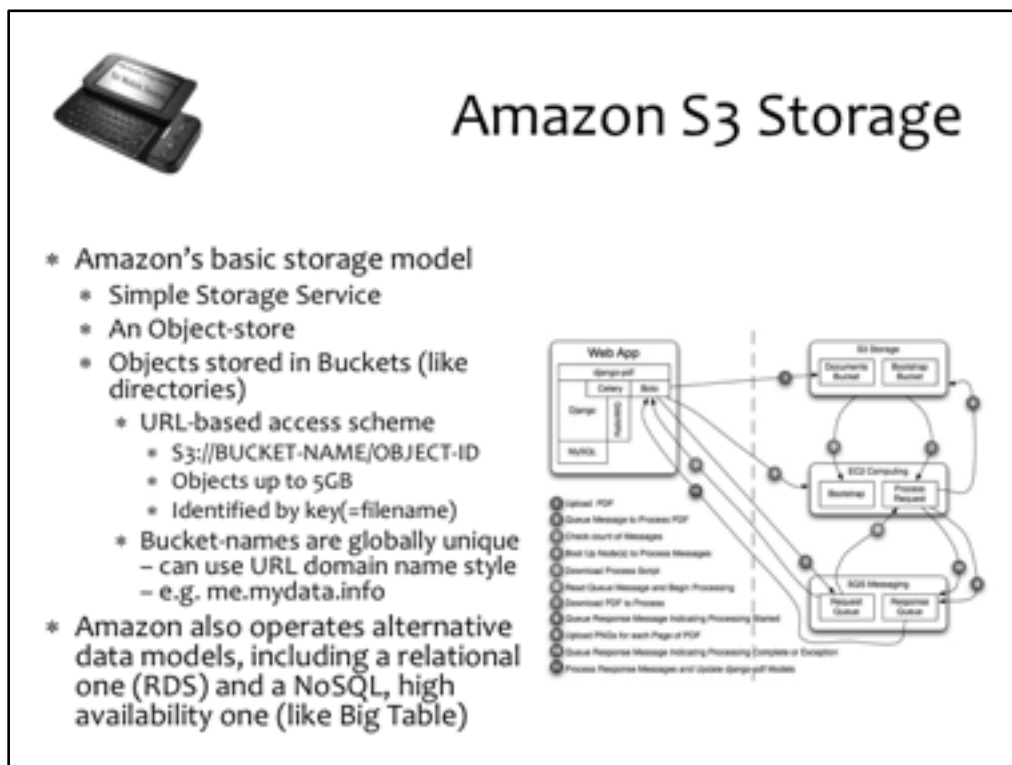


Big Table was developed by Google to allow their search engines to store and manage data over a large number of separate server farms. The idea was that losing any single farm would not result in data loss, but also that it should be as quick to access information for many location around the world as any other. Google invented the MapReduce algorithm as a way of concentrating large amounts of intelligent storage nodes to make them searchable as a single entity. The data is stored in a raw text format (for most situations) that makes the development and maintenance of the search algorithms simple and direct. The entire database (all of Google's data) is searchable as a single entity, although since all of the data is replicated in several places, many parts of a search operation are likely to be redundant.

Google Data is stored in "tablets", blocks of around 100-200MB of data (i.e. a smallish hard disk drive from about 15 years ago – indicating the origins of BigTable in a farm of cheap simple machines) that contain the actual data plus indexes for it. The MapReduce algorithm farms out requests to large numbers of nodes, waits for responses to come back and then concentrates these responses (removing the redundant data).

One significant feature of BigTable is that it is not relational. While the GQL language is structured similarly to SQL, there is no equivalent of a JOIN operation (normally used to incorporate data from more than one table into a single result set). You CAN work with relational operations in Google Big Table, but the joins need to be done in

program code (e.g. query for a customer's record, then use the key from this to query for the custc




Amazon provides several forms of cloud storage: SimpleDB – a non-relational database model based on simple object serialization. This results in a schemaless system (no table definitions) that you can store any object in.

Amazon RDS is a relational data model using MySQL or Oracle databases that can be replicated and mirrored over a number of servers

Amazon DynamoDB is a NoSQL database model, more like Google's Big Table (no joins or transactions)

Amazon, by providing a range of self-configurable virtual machines, also allows you to set up and host your own database engines (MySQL, SQL Server etc.). They also support CouchBase (a non-relational high-performance key-value store [think localStorage style]) and 10Gen – a system for accessing MongoDB (a high performance document-oriented data store)




Azure Storage Services

- * Microsoft provides a hierarchy of data storage mechanisms
 - * SQL Server across multiple server instances
 - * Blob storage (Binary Large Objects)
 - * Table storage
 - * Row-like entities
 - * Up to 255 columns
 - * No schema – therefore each row can have a different set of columns
 - * This is very like Big Table
 - * Sensibly, Microsoft are playing to their strengths (SQL Server) and but also expanding into high-availability

Storage Account

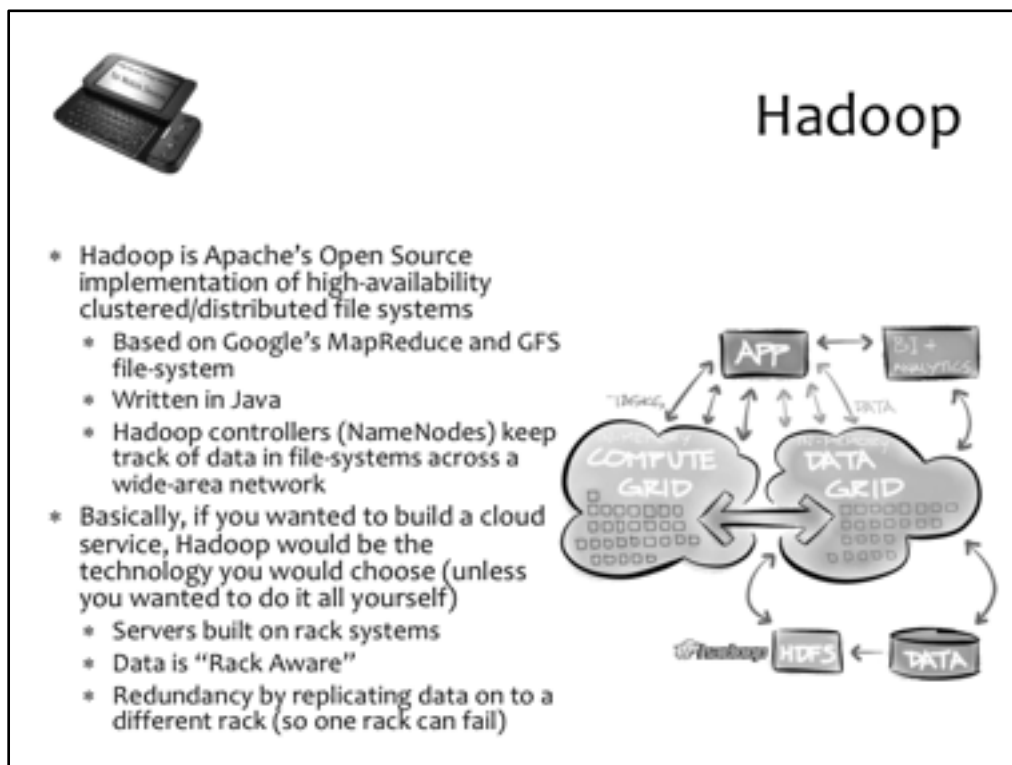
Storage Account



<http://social.technet.microsoft.com/wiki/contents/articles/1674.data-storage-offerings-on-the-windows-azure-platform.aspx>

Azure provides a wide range of database machinery. You can host self-administered SQL Servers on virtual machines, use Table storage (a bit like Big Table) or simply store binary large objects by key – these can be any type of data you like – even traditional relational databases.

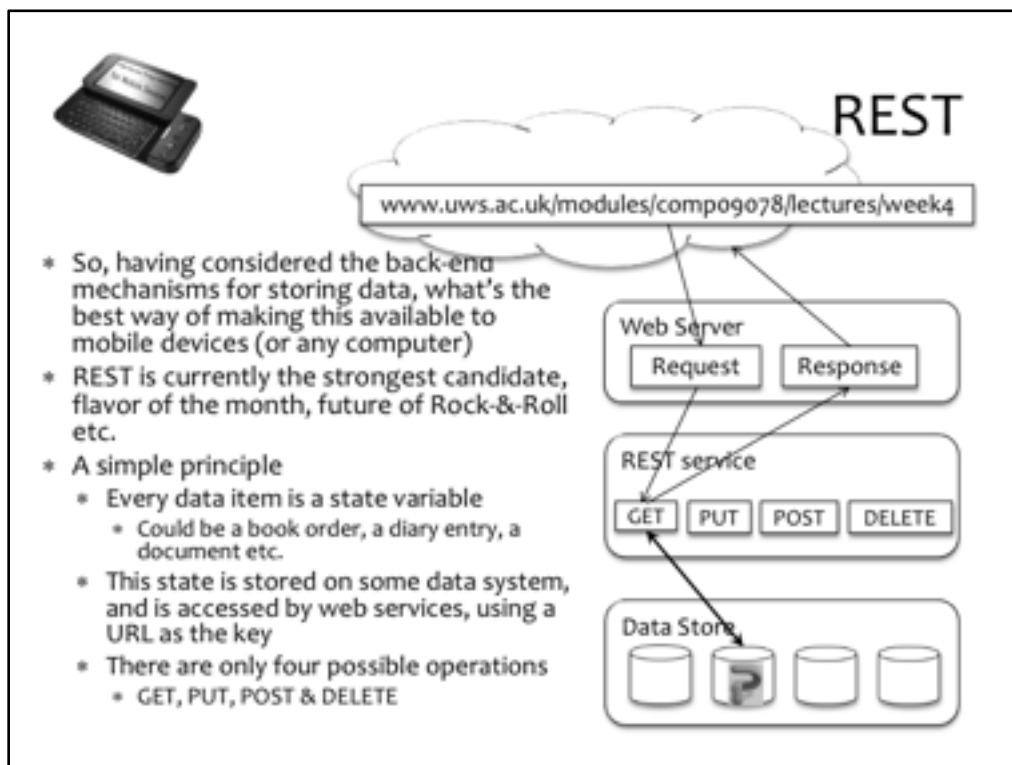
Blob storage uses a high-availability key-value system where the indexes are queried using a MapReduce algorithm. This provides very quick access to big data structures.



Hadoop is nothing more than an open source version of the indexing system used by Google in Big Table.

Indexes are collections of keys, each a url leading to a server and file that stores the data. The rest of Hadoop is based around a MapReduce algorithm to provide quick access to the indexes.

Hadoop also includes the notion of 'Rack Awareness', so that different servers instances can be replicated on different machines (racks) so that if a rack fails for any reason (power failure, dead network switch) then the data is still available on a different rack. This is like a big version of RAID (Redundant Array of Inexpensive Disks) that you can have on a PC to make the file system fault tolerant, except that the principle is extended to whole servers and server farms



REST started as the doctoral thesis of Roy Fielding in 2000. It is described as an “architectural style” for application infrastructure.

REST systems consist of Clients and Servers. A Client makes requests, a Server responds to them. In this respect, it is no different from the HTTP protocol used by the WWW.

REST primarily is a method for managing Resources between the client and server.

A resource is represented by a defined piece of data on a server – a document, document fragment that captures the current state of some data that is considered in the application to be an entity. For example, an Amazon order, a Customer record, a PowerPoint presentation, three numbers that represent temperature, barometric pressure and the probability of rainfall, or anything else that is a significant object in an application.

Four things you can do in a REST service:

GET a resource – in which case the server extracts and returns the data referenced by the URL.

PUT a resource into the server – the data is sent to the server (in HTTP headers – usually in some structured format such as JSON or XML) and stored,

using the URL as a key. For example, in the slide, we could add a new document to the data at www.uws.ac.uk/modules/comp09078/week5.

POST a resource, which essentially updates or replaces the current resource at a given URL

DELETE a resource, which instructs the server to remove it.

Using only these four operations (verbs), you can completely manage the overall server state.

Typically, REST is operated by a library of code that interacts with the HTTP services handled by a web server. Microsoft web services are implemented as a layer of the software stack somewhere between the underlying data services. In Google App Engine, a very simple (but functionally complete) REST library manages the underlying Big Table data-store. The REALLY GOOD thing about either of these, is that data-store interactions – all the developer needs to do is to format URLs to match the required resource and the operation is needed, the data for the new or updated resource needs to be added to the request headers. This is a fairly straightforward job.

To see a defined rest protocol, have a look at <http://code.google.com/p/appengine-rest-server/>, which defines the REST protocol for ANY interactions with the Google BigTable data layer.

Also see: <http://mcm-share-prices-hrd.appspot.com/>, which is the home page for a REST server I've built to share prices. This gives some examples you can test by typing them into the URL bar of a server.



Accessing a REST Service

- * You don't need much in the way of specialized equipment for this
 - * You can access a GET request using the URL bar of any browser
 - * For POST, PUT and DELETE requests, you need to be able to manipulate headers
 - * You have access to a useful piece of kit that let you do this:
 - * PostMan
 - * A Chrome plug-in that lets you make REST requests and receive the responses



REST is based on only four of these, which are used in a way that is fully compatible with HTTP's definitions.

GET – this operation requires only a URL (e.g. <http://mcm-share-prices-hrd.appspot.com/rest/shareprice/> which will return the first 50 shareprice instances). Generally, in a GET operation, we include some other information at the end of the URL to narrow the query. This can be a specific key... (e.g. <http://mcm-share-prices-hrd.appspot.com/rest/shareprice/ahZzfm1jbS1zaGFyZS1wcmlljXMTaHJkchELEgpTaGFyZVByaWNIGP8HDA>)

which will return one shareprice instance with a matching key:

```
<?xml version="1.0" encoding="utf-8"?>
<shareprice>
```

```
<key>ahZzfm1jbS1zaGFyZS1wcmljZXMTaHJkchELEgpTaGFyZVByaWNlGP8HDA</key>
  <updated>2009-12-28T00:00:00</updated>
  <last>623.4</last>
  <name>Google Inc.</name>
```

```

    <removed>false</removed>
    <ticker>GOOG</ticker>
    <change>4.92</change>
  </shareprice>

```

...or a **predicate** which is a term added to the query to indicate the target of a request. Predicate terms, like in Javascript where you can do `if(name=="Fred")`. We can't use terminology like the REST service I'm using provides a different way. To add a predicate, we need to include a the required type of comparison (equality, less than etc.) and the part of the record we want updated, ticker etc.). For example:

`http://mcm-share-prices-hrd.appspot.com/rest/shareprice?feq_ticker=GOOG`

requests all of the shareprice records where the ticker is GOOG (Google) The other predicate implementation of REST accepts are:

<code>?flt_</code>	for less-than
<code>?fgt_</code>	for greater than
<code>?feq_</code>	equality
<code>?fne_</code>	not equal
<code>?fle_</code>	less or equal
<code>?fge_</code>	greater or equal.

POST – this operation is used to create or update a record. It overlaps a bit with PUT, which is the method to use to create a new record, is used by some REST servers to update all of the one.

This problem with using POST (or PUT) is that we need a way to include the data that we are sending to the service.

HTTP includes a very flexible way of including data using Headers – additional information that

request. To add a new record to the shareprices service (not sensible, since this service gives prices – not ones you make up as you go along, but the example might be useful), we need which is just `http://mcm-share-prices-hrd.appspot.com/rest/shareprice/` and the data that which is a shareprice record in XML format but with no key value. The easiest way to get there is to copy an existing shareprice and update the values – e.g. :

```
<shareprice>
  <updated>2009-12-28T00:00:00</updated>
  <last>101.01</last>
  <name>UWS Trading Inc.</name>
  <removed>false</removed>
  <ticker>UWST</ticker>
  <change>10.1</change>
</shareprice>
```

This block of XML is pasted into the header of an HTTP request (you'll find a neat little utility on Moodle that lets you do this, or you can use the cRESTClient extension for Google Chrome) and the response is the key of the newly added record – e.g. :

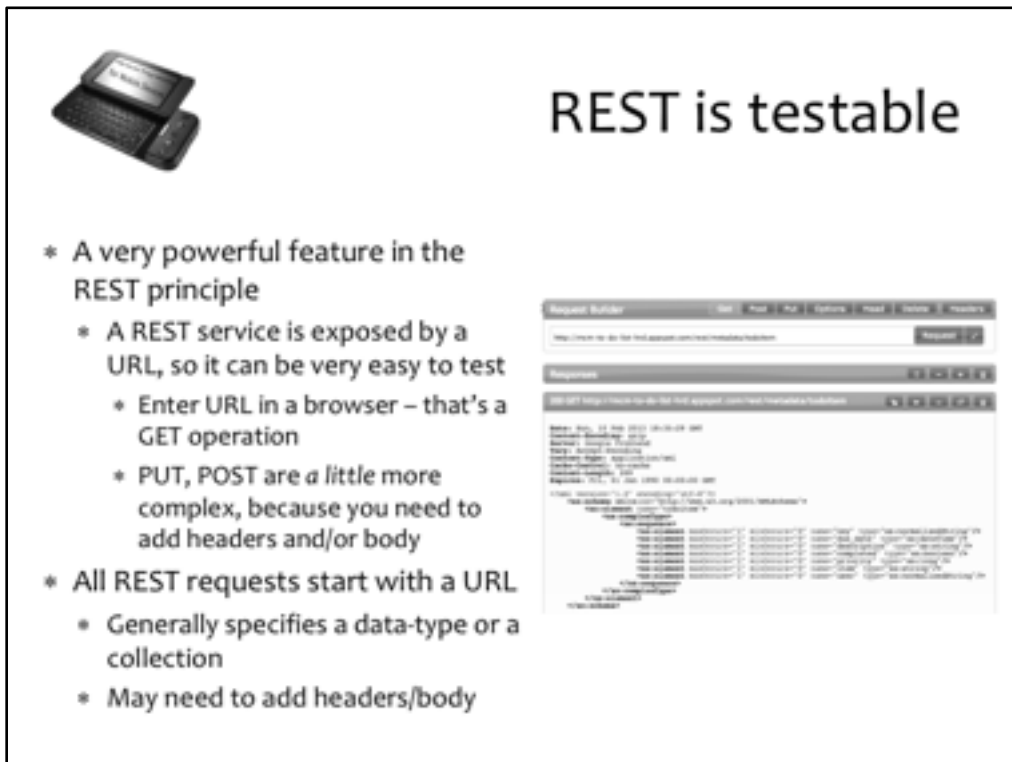
```
ahZzfm1jbS1zaGFyZS1wcmljZXMtaHJkchMLEgpTaGFyZVByaWNlGIPj2QEM
```

PUT - operations are similar to POST, except that you include a key value of an existing record and the record is updated.

DELETE operations require a key to indicate the element to delete, e.g.

```
http://mcm-share-prices-hrd.appspot.com/rest/shareprice/ahZzfm1jbS1zaGFyZS1wcmljZXMtaHJkchMLEgpTaGFyZVByaWNlGIPj2QEM
```

which would delete the record added in the above POST operation.



REST is testable

- * A very powerful feature in the REST principle
 - * A REST service is exposed by a URL, so it can be very easy to test
 - * Enter URL in a browser – that's a GET operation
 - * PUT, POST are a little more complex, because you need to add headers and/or body
- * All REST requests start with a URL
 - * Generally specifies a data-type or a collection
 - * May need to add headers/body

HTTP headers are a core part of the protocol. Headers are data packages associated with a request (URL).

Headers carry info about: the client/browser, the requested page/element, the server etc.


```
GET /tutorials/other/top-20-mysql-best-practices/ HTTP/1.1
Host: net.tutsplus.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q7ib3vtdjq120
Pragma: no-cache
Cache-Control: no-cache
```

In the above, first line is the request (GET + URL + protocol [HTTP/1.1]). The rest is headers. The response from the server also includes headers:

```
HTTP/1.x 200 OK
Transfer-Encoding: chunked
Date: Sat, 28 Nov 2009 04:36:25 GMT
Server: LiteSpeed
Connection: close
X-Powered-By: W3 Total Cache/0.8
Pragma: public
Expires: Sat, 28 Nov 2009 05:36:25 GMT
Etag: "pub1259380237;gz"
Cache-Control: max-age=3600, public
Content-Type: text/html; charset=UTF-8
Last-Modified: Sat, 28 Nov 2009 03:50:37 GMT
X-Pingback: http://net.tutsplus.com/xmlrpc.php
Content-Encoding: gzip
Vary: Accept-Encoding, Cookie, User-Agent
```

First line is the Status line (indicating the nature of the response).

For more on this, look at <http://net.tutsplus.com/tutorials/other/http-headers-for-dummies/>



Lab Work

- * There is no specific lab work for this week
 - * If you didn't complete last week's lab exercise, you should do so today
 - * If you did, you'll be in a position to start creating your own web services – next week's lab exercise
 - * By all means, feel free to start this today (lab 4 is on Moodle)
- * Remember, your project will involve creating both client and service parts of a mobile application
 - * If you're already able to create web services reliably, good for you – you can probably skip the next lab session
 - * If you THINK you can and get it wrong, your chances of passing this module are low

Basically, the assessment is a test of whether you can create a client app for a simple service:

- no storage needed (client or server)

- little or no formatting of data from service

In the lab you'll want to test a request to the service. You can use a browser for this. If you want more data than that, use one of the REST clients described in the slides – either my RESTClient.exe or one of the available browser plug-ins – I recommend the POSTMAN plug-in for Google Chrome.

The idea of the lab is to get used to using the protocol in a client app, because in next week's lab, you'll be building a web service (using App Engine developer kit or some other). If you use the GAE, then there is a drop-in REST toolkit available for it and you can concentrate on creating suitable models for whatever data you want your service to host.