Cloud Computing

Big Data

Dell Zhang
Birkbeck, University of London
2015/16
The Challenges and Opportunities of Big Data
Why Big Data?

640K ought to be enough for anybody.
Big Data Everywhere!

- Lots of data are being collected and warehoused
  - Web data, e-commerce
  - Purchases at department/grocery stores
  - Bank/Credit Card transactions
  - Social Network
Big Data Challenges (5Vs)

- **Volume**: amount of data
- **Velocity**: speed of data in/out
- **Variety**: range of data types, sources
- **Value**: low density of valuable information
- **Veracity**: uncertainty, imprecision, missing values, and mis-statements or untruths
How Much Data?

• Google processes 20 PB a day (2008)
• Wayback Machine has 3 PB + 100 TB/month (3/2009)
• Facebook has 2.5 PB of user data + 15 TB/day (4/2009)
• eBAY has 6.5 PB of user data + 50 TB/day (5/2009)
• CERN’s Large Hydron Collider (LHC) generates 15 PB a year
A TIDAL WAVE OF DATA

Twitter
8 TB per day

Facebook
100 TB per day

Global Businesses
1.8 ZB in 2011

US Library of Congress
235 TB

Boeing
640 TB per flight

Wal-Mart
2.5 PB stored
The Large Hadron Collider
The Earthscope Project

- Annual budget: $25,000,000
- Construction cost: $197,000,000
- Staff: 110
- Physical size: 3.8 million square miles
- Scientific utility: 10
- WIIFY: 10
- Wow factor: 10
The PRISM

Current Providers
- Microsoft (Hotmail, etc.)
- Google
- Yahoo!
- Facebook
- PalTalk
- YouTube
- Skype
- AOL
- Apple

What Will You Receive in Collection (Surveillance and Stored Comms)?
It varies by provider. In general:
- E-mail
- Chat – video, voice
- Videos
- Photos
- Stored data
- VoIP
- File transfers
- Video Conferencing
- Notifications of target activity – logins, etc.
- Online Social Networking details
- Special Requests

Complete list and details on PRISM web page:
Go PRISMFAA

TOP SECRET//SI//ORCON//NOFORN
Boundless Informant

- It is a big data analysis and data visualization system used by the United States National Security Agency (NSA). [Wikipedia]
  - According to published slides, Boundless Informant leverages Free and Open Source Software — and is therefore "available to all NSA developers" — and corporate services hosted in the cloud.
  - The tool uses HDFS, MapReduce, and Accumulo (formerly Cloudbase) for data processing.
What Kinds of Data?

• Structured Data
  – Tables/Transactions/Legacy Data, ...

• Semi-Structured Data
  – XML, ...

• Unstructured Data
  – Text Data: the Web, ...
  – Graph Data: Social Networks, ...
What to Do with Big Data?

• Aggregation and Statistics
  – Data Warehouse and OLAP

• Indexing, Searching, and Querying
  – Keyword based Search
  – Pattern Matching (XML/RDF)

• Knowledge Discovery
  – Data Mining
  – Statistical Modeling
What to Do with Big Data?

• Example: Answering factoid questions
  – Pattern matching on the Web
  – Works amazingly well

Who shot Abraham Lincoln? → search “* shot Abraham Lincoln”

(Brill et al., TREC 2001; Lin, ACM TOIS 2007)
What to Do with Big Data?

- **Example: Learning relations**
  - Start with seed instances
  - Search for patterns on the Web
  - Using patterns to find more instances

Wolfgang Amadeus Mozart (1756 - 1791)
Einstein was born in 1879
Birthday-of(Mozart, 1756)
Birthday-of(Einstein, 1879)
PERSON (DATE –
PERSON was born in DATE

(Agichtein and Gravano, DL 2000; Ravichandran and Hovy, ACL 2002; ...)
IBM’s Watson
No Data Like More Data!

(Banko and Brill, ACL 2001)
(Brants et al., EMNLP 2007)

How do we get here if we’re not Google?
The Law of Large Numbers

• In probability theory, the law of large numbers (LLN) is a theorem that describes the result of performing the same experiment a large number of times. According to the law, the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed. [Wikipedia]
## Average household income by programming language

<table>
<thead>
<tr>
<th>Language</th>
<th>Average Household Income ($)</th>
<th>Data Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Puppet</td>
<td>87,589.29</td>
<td>112</td>
</tr>
<tr>
<td>Haskell</td>
<td>89,973.82</td>
<td>191</td>
</tr>
<tr>
<td>PHP</td>
<td>94,031.19</td>
<td>978</td>
</tr>
<tr>
<td>CoffeeScript</td>
<td>94,890.80</td>
<td>435</td>
</tr>
<tr>
<td>VimL</td>
<td>94,967.11</td>
<td>532</td>
</tr>
<tr>
<td>Shell</td>
<td>96,930.54</td>
<td>979</td>
</tr>
<tr>
<td>Lua</td>
<td>96,930.69</td>
<td>101</td>
</tr>
<tr>
<td>Erlang</td>
<td>97,306.55</td>
<td>168</td>
</tr>
<tr>
<td>Clojure</td>
<td>97,500.00</td>
<td>269</td>
</tr>
<tr>
<td>Python</td>
<td>97,578.87</td>
<td>2314</td>
</tr>
<tr>
<td>JavaScript</td>
<td>97,598.75</td>
<td>3443</td>
</tr>
<tr>
<td>Emacs Lisp</td>
<td>97,774.65</td>
<td>355</td>
</tr>
<tr>
<td>C#</td>
<td>97,823.31</td>
<td>665</td>
</tr>
<tr>
<td>Ruby</td>
<td>98,238.74</td>
<td>3242</td>
</tr>
<tr>
<td>C++</td>
<td>99,147.93</td>
<td>845</td>
</tr>
<tr>
<td>CSS</td>
<td>99,881.40</td>
<td>527</td>
</tr>
<tr>
<td>Perl</td>
<td>100,295.45</td>
<td>990</td>
</tr>
<tr>
<td>C</td>
<td>100,766.51</td>
<td>2120</td>
</tr>
<tr>
<td>Go</td>
<td>101,158.01</td>
<td>231</td>
</tr>
<tr>
<td>Scala</td>
<td>101,460.91</td>
<td>243</td>
</tr>
<tr>
<td>ColdFusion</td>
<td>101,536.70</td>
<td>109</td>
</tr>
<tr>
<td>Objective-C</td>
<td>101,801.60</td>
<td>562</td>
</tr>
<tr>
<td>Groovy</td>
<td>102,650.86</td>
<td>116</td>
</tr>
<tr>
<td>Java</td>
<td>103,179.39</td>
<td>1402</td>
</tr>
<tr>
<td>XSLT</td>
<td>106,199.19</td>
<td>123</td>
</tr>
<tr>
<td>ActionScript</td>
<td>108,119.47</td>
<td>113</td>
</tr>
<tr>
<td>Rank</td>
<td>Kicker</td>
<td>Make %</td>
</tr>
<tr>
<td>------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>Garrett Hartley</td>
<td>87.7</td>
</tr>
<tr>
<td>2</td>
<td>Matt Stover</td>
<td>86.8</td>
</tr>
<tr>
<td>3</td>
<td>Robbie Gould</td>
<td>86.2</td>
</tr>
<tr>
<td>4</td>
<td>Rob Bironas</td>
<td>86.1</td>
</tr>
<tr>
<td>5</td>
<td>Shayne Graham</td>
<td>85.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>Dave Rayner</td>
<td>72.2</td>
</tr>
<tr>
<td>52</td>
<td>Nick Novak</td>
<td>71.9</td>
</tr>
<tr>
<td>53</td>
<td>Tim Seder</td>
<td>71.0</td>
</tr>
<tr>
<td>54</td>
<td>Jose Cortez</td>
<td>70.7</td>
</tr>
<tr>
<td>55</td>
<td>Wade Richey</td>
<td>66.1</td>
</tr>
</tbody>
</table>
Meaningfulness of Answers

• A big data-mining risk is that you may “discover” patterns that are meaningless
  – Bonferroni’s principle: (roughly) if you look in more places for interesting patterns than your amount of data will support, you are bound to find crap.
Meaningfulness of Answers

• Example: Rhine Paradox
  – Joseph Rhine was a parapsychologist in the 1950’s who hypothesized that some people had Extra-Sensory Perception (ESP)
  – He devised an experiment where subjects were asked to guess 10 hidden cards: red or blue
  – He discovered that almost 1 in 1000 had ESP: they were able to get all 10 right!
Meaningfulness of Answers

• Example: Rhine Paradox
  – He told these people they had ESP and called them in for another test of the same type
  – Alas, he discovered that almost all of them had lost their ESP
  – What did he conclude?
How do we get data for computation?

What’s the problem here?
How do we get data for computation?

- Don’t move data to computation…, move computation to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the programs on the node that has the data local

- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
Distributed File Systems

• A Distributed File System (that provides global file namespace) is the answer
  – GFS (Google File System) for Google’s MapReduce
  – HDFS (Hadoop Distributed File System) for Hadoop
    • HDFS = GFS clone (same basic ideas)

• Typical usage pattern
  – Huge files (100s of GB to TB)
  – Data is rarely updated in place
  – Reads and appends are common
Distributed File Systems

- The problem of **reliability**: if nodes fail, how to store data persistently?
  - Data kept in “chunks” spread across machines
  - Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure
GFS: History

- Google File System, a paper published in 2003 by Google Labs at OSDI
  - Explains the design and architecture of a distributed system apt for serving very large data files (internally used by Google for storing documents collected from the Web).
  - Open Source versions have been developed at once: Hadoop File System (HDFS), Kosmos File System (KFS).
GFS: Motivation

• Why do we need a distributed file system in the first place?
  – Traditional Network File System (NFS) does not meet scalability requirements
    • What if file1 gets really big?
  – A Large-Scale Distributed File System requires
    • A virtual file namespace
    • Partitioning of files in “chunks”.
GFS: Motivation

A traditional network file system

A large scale distributed file system
GFS: Assumptions

• Commodity hardware over “exotic” hardware
  – Scale “out”, not “up”
• High component failure rates
  – Inexpensive commodity components fail all the time
• “Modest” number of huge files
  – Multi-gigabyte files are common, if not encouraged
• Files are write-once, mostly appended to
  – Perhaps concurrently
• Large streaming reads over random access
  – High sustained throughput over low latency
GFS: Design Decisions

• Chunk servers
  – File is split into equal-size contiguous chunks: typically 64MB
  – Each chunk is replicated (default 3x)
  – Try to keep replicas in different racks

• Master node
  – Stores metadata and coordinates access
  – Simple centralized management
  – Might be replicated
GFS: Design Decisions

• Client library for file access
  – Talks to master node to find chunk servers
  – Connects directly to chunk servers to access data
  – Maintains a cache with the chunks’ locations (but not the chunks themselves)
    • No data caching: little benefit due to large datasets, streaming reads.
  – Simplify the API: some of the issues are pushed onto the client (e.g., data layout)
GFS: Architecture

1. Client cache reads file `/dirB/file1` from cache.
2. Master node sends chunk locations to client cache.
3. Client cache sends file `/dirB/file1` to servers.
4. Client cache reads file `/dirB/file1` from server (indicated by `(...)`) for read operations.

GFS structure and messages are indicated by arrows.
GFS Architecture
GFS: Technical Details

• The architecture works best for very large files (e.g., several GBs), divided in large (e.g., 64 MBs) chunks
  – This limits the metadata information served by the Master

• Each server implements recovery & replication techniques
  – Default: 3 replicas
GFS: Technical Details

• Availability
  – The Master sends heartbeat messages to servers, and initiates a replacement when a failure occurs.

• Scalability
  – The Master is a potential single point of failure; its protection relies on distributed recovery techniques for all changes that affect the file namespace.
Workflow of a `write()` operation (simplified)

1. `write(/dirB/file1, blob)`
2. List of chunk servers
3. `append(blob)`
4. `ack`

Write (append) in GFS (simplified to non-concurrent operations)
Namespace Updates: Distributed Recovery Protocol
From GFS to HDFS

• Terminology differences:
  – GFS master = Hadoop namenode
  – GFS chunkservers = Hadoop datanodes

• Functional differences:
  – No file appends in HDFS (planned feature)
  – HDFS performance is (likely) slower

For the most part, we’ll use the Hadoop terminology...
HDFS: Architecture

Adapted from (Ghemawat et al., SOSP 2003)
HDFS: NameNode Metadata

• Meta-data in Memory
  – The entire metadata is in main memory
  – No demand paging of meta-data

• Types of Metadata
  – List of files
  – List of blocks for each file
  – List of DataNodes for each block
  – File attributes, e.g. creation time, replication factor

• A Transaction Log
  – Records file creations, file deletions. etc
HDFS: Namenode Responsibilities

• Managing the file system namespace:
  – Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.

• Coordinating file operations:
  – Directs clients to datanodes for reads and writes
  – No data is moved through the namenode

• Maintaining overall health:
  – Periodic communication with the datanodes
  – Block re-replication and rebalancing
  – Garbage collection
HDFS: DataNode

• Block Server
  – Stores data in the local file system (e.g. ext3)
  – Stores meta-data of a block (e.g. CRC)
  – Serves data and meta-data to Clients

• Block Report
  – Periodically sends a report of all existing blocks to the NameNode

• Facilitates Pipelining of Data
  – Forwards data to other specified DataNodes
HDFS: Block Placement

• Current Strategy
  – One replica on local node
  – Second replica on a remote rack
  – Third replica on the same remote rack
  – Additional replicas are randomly placed

• Clients read from nearest replica

• Would like to make this policy pluggable
HDFS: Data Correctness

• Use checksums to validate data
  – CRC32

• File creation
  – Client computes checksum per 512 byte
  – DataNode stores the checksum

• File access
  – Client retrieves the data and checksum from DataNode
  – If validation fails, Client tries other replicas
HDFS: NameNode Failure

• A single point of failure
• Transaction Log stored in multiple directories
  – A directory on the local file system
  – A directory on a remote file system (NFS/CIFS)
• Need to develop a real high availability solution
Distributed Access Structures

• Indexing
  – Hashing based techniques
    • Consistent Hashing
  – Tree based techniques
    • Distributed B-Tree
Indexing

• We assume a (very) large collection $C$ of pairs $(k,v)$, where $k$ is a key and $v$ is the value of an object (seen as row data).

• An index on $C$ is a structure that associates the key $k$ with the (physical) address of $v$. 
Indexing

- An index supports *dictionary operations*:
  - insertion $\text{insert}(k, v)$
  - deletion $\text{delete}(k)$
  - key search $\text{search}(k): v$
  - range search $\text{range}(k_1, k_2): \{v\}$ [optional]

- In a distributed index, one should also consider:
  - (node) leave and (node) join operations
Hashing (Centralised)

Hash directory

Suzan
h('Suzan')=3

John
h('John')=3

Hash file
b0
b1
b3
b2
Hashing (Distributed)

• The straightforward idea
  – Everybody uses the same hash function, and buckets are replaced by servers.

• Issues with distribution
  – Dynamicity: at Web scale, we must be able to add or remove servers at any moment.
  – Inconsistencies: it is very hard to ensure that all participants share an accurate view of the system (e.g., the hash function)
Hashing (Distributed)

• A naïve solution
  – Let $N$ be the number of servers.
  – The function $\text{modulo}(h(k), N) = i$ maps a pair $(k,v)$ to server $S_i$.
  – Fact: if $N$ changes, or if a client uses an invalid value for $N$, the mapping becomes inconsistent.
Consistent Hashing

• With consistent hashing, addition or removal of an instance does not significantly change the mapping of keys to servers.

  – A simple, non-mutable hash function \( h \) maps both the keys and the servers’ IPs to a large address space \( A \) (e.g., \([0, 2^{64}−1]\)) organized as a ring in clockwise order.

  – If \( S \) and \( S' \) are two adjacent servers on the ring: all the keys in range \([h(S),h(S')]\) are mapped to \( S \).
Some (Really Useful) Refinements

• What if a server fails?
  – Use replication:
    • put a copy on the next machine (on the ring),
    • then on the next after the next,
    • and so on.
Some (Really Useful) Refinements

• How can we balance the load?
  – Map a server to several points on the ring (virtual nodes)
    • the more points, the more load received by a server;
    • also useful if the server fails: data relocation is more evenly distributed.
    • also useful in case of heterogeneity (the rule in large-scale systems).
Where is the Hash Directory?

• On a specific (“Master") node, acting as a load balancer (e.g., in caching systems)
  – raises scalability issues

• Each node records its successor on the ring
  – may require $O(N)$ messages for routing queries:
    not resilient to failures
Where is the Hash Directory?

• Each node records $\log N$ carefully chosen other nodes (e.g., in P2P)
  – ensures $O(\log N)$ messages for routing queries
  – convenient trade-off for highly dynamic networks

• Full duplication of the hash directory at each node (e.g., in Dynamo)
  – ensures 1 message for routing
  – heavy maintenance protocol which can be achieved through gossiping
Amazon Dynamo

• A distributed storage system that targets high availability
  – Your shopping cart is stored there!

• An open-source version: Voldemort
  – It is used by LinkedIn
Amazon Dynamo

• Features
  – Duplicates and maintains the hash directory at each node via gossiping: queries can be routed to the correct server with 1 message.
  – The hosting server replicates $N$ (application parameter) copies of its objects on the $N$ distinct nodes that follow $S$ on the ring.
  – Propagates updates asynchronously: may result in update conflicts, solved by the application at read-time.
  – Use a fully distributed failure detection mechanism: failure are detected by individual nodes when then fail to communicate with others.
**B-Tree (Centralised)**

![B-Tree Diagram]

**Branch Blocks**

- B
- C
- Cr
- Di
- Lu
- Rh
- F
- H
- Kar
- N
- P
- Ph
- Sam
- St
- Su

**Leaf Blocks**

- Karl, ROWID
- Kathy, ROWID
- Kim, ROWID
- Lance, ROWID
- Luis, ROWID
- Mark, ROWID
- Many, ROWID
- Mike, ROWID
- Mike, ROWID
- Nancy, ROWID
- Nancy, ROWID
- Nancy, ROWID
- Nicole, ROWID
- Norm, ROWID
- Pablo, ROWID
- Paula, ROWID
- Paula, ROWID
- Peter, ROWID
- Phil, ROWID
- Pierre, ROWID
- Rajiv, ROWID
- Raoul, ROWID
B-Tree (Distributed)
B-Tree (Distributed)

• Issues with distribution
  – All operations follow a top-down path: potential factor of non-scalability

• Possible solutions
  – *caching* of the tree structure, or part of it, on the Client node
  – *replication* of the upper levels of the tree
  – *routing tables*, stored at each node, enabling horizontal and vertical navigation in the tree
BigTable

• Can be seen as a distributed *map* structure, with features taken from
  – B-trees
  – non-dense indexed files
BigTable

• Context
  – A controlled environment, with homogeneous servers located in a data centre;
  – A stable organization, with long-term storage of large structured data;
  – A data model (column-oriented tables with versioning)
BigTable

• Design
  – Close to a B-tree, with large capacity leaves
  – Scalability is achieved by a cache maintained by Client nodes
BigTable

• Structure Overview
  – The table is partitioned in “tablets”
    • Tablets are indexed by upper levels
    • Full tablets are split, with upward adjustment
  – Leaf level:
    • A “tablet” organized in “rows” indexed by a key
    • Rows are stored in lexicographic order on the key values
BigTable

• Architecture: one Master - many Servers
  – The Master maintains the root node and carries out administrative tasks
  – Scalability is obtained with Client cache that stores a (possibly outdated) image of the tree
    • A Client request may fail, due to an out-of-date image of the tree
    • An adjustment requires at most \textit{height}(Tree) rounds of message
a) A new client contacts a distributed system

b) Using its image, the client directly contacts N
Client

root node

search(856)

answer + image adjustment
Apache Cassandra

• BigTable data model + Dynamo infrastructure
  – Initially developed by Facebook
  – Now used by Twitter and Digg, etc.
Distributed Access Structures

• Key points that you should remember
  – Reliability: always be ready to face a failure somewhere
    • Detect failures and deal with it
    • Use replication
Distributed Access Structures

• Key points that you should remember
  – Scalability: no single point of failure; even load distribution over all the nodes.
  • Distribute (and maintain) routing information: trade-off between maintenance cost and operations cost.
  • Cache an image of the structure (e.g., in the Client): design a convergence protocol if the image gets outdated.
Distributed Access Structures

- Key points that you should remember
  - Efficiency: clearly depends on the amount of information replicated at each node or at the Client
    - Stable systems: the structure can be duplicated at each node, and allows $O(1)$ cost – low maintenance
    - Highly dynamic systems: very hard to maintain a consistent view of the structure for each participant
NoSQL Databases

• NoSQL ("not only SQL") is a broad class of database management systems identified by non-adherence to the widely used relational model
  – Relational tables ➔ structured storage (typically key-value pairs)
  – Scaling horizontally:
    • Breaking some of the fundamental guarantees of SQL in order to reach massive scale
Relational Tables

• A Relational Database Management System (RDBMS) stores data as a collection of tables, and provides relational operators to manipulate the data in tabular form.

• All modern commercial relational databases employ SQL as their query language, therefore they are also called SQL databases.
# Relational Tables

<table>
<thead>
<tr>
<th>Car</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>CarKey</td>
<td>MakeKey</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MakeModel</th>
<th>Make</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelKey</td>
<td>MakeKey</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Make</th>
<th>MakeKey</th>
<th>Make</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nissan</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Honda</td>
<td></td>
</tr>
</tbody>
</table>
Relational Tables

• The challenge is *scalability*: SQL databases are hard to be expanded by adding more servers as data have to be replicated across the new servers or partitioned between them
  – Data shipping: time-consuming and expensive
  – Nearly impossible to guarantee maintenance of **referential integrity** (any field in a table that’s declared a foreign key can contain only values from a parent table's primary key or a candidate key)
Structured Storage

• May not require fixed table schemas
• Usually avoid *join* operations
  – if a join is needed that depends on shared tables, then replicating the data is hard and blocks easy scaling
• Often provide weak consistency guarantees
  – such as eventual consistency and transactions restricted to single data items
  – in most cases, you can impose full ACID guarantees by adding a supplementary middleware layer
Key-Value Pairs

• Key-value databases are item-oriented
  – All relevant data relating to an item are stored in that item. Therefore data are commonly duplicated between items in a table
  – A table can contain very different items
  – It radically improves scalability by eliminating the need to join data from multiple tables
# Key-Value Pairs

<table>
<thead>
<tr>
<th>Key</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Make</strong>: Nissan</td>
</tr>
<tr>
<td></td>
<td><strong>Model</strong>: Pathfinder</td>
</tr>
<tr>
<td></td>
<td><strong>Color</strong>: Green</td>
</tr>
<tr>
<td></td>
<td><strong>Year</strong>: 2003</td>
</tr>
<tr>
<td>2</td>
<td><strong>Make</strong>: Nissan</td>
</tr>
<tr>
<td></td>
<td><strong>Model</strong>: Pathfinder</td>
</tr>
<tr>
<td></td>
<td><strong>Color</strong>: Blue</td>
</tr>
<tr>
<td></td>
<td><strong>Year</strong>: 2005</td>
</tr>
<tr>
<td></td>
<td><strong>Trans</strong>: Automatic</td>
</tr>
<tr>
<td>API call</td>
<td>API functional description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CreateDomain</td>
<td>Creates a domain that contains your dataset.</td>
</tr>
<tr>
<td>DeleteDomain</td>
<td>Deletes a domain.</td>
</tr>
<tr>
<td>ListDomains</td>
<td>Lists all domains.</td>
</tr>
<tr>
<td>DomainMetadata</td>
<td>Retrieves information about creation time for the domain, storage information both as counts of item names and attributes, and total size in bytes.</td>
</tr>
<tr>
<td>PutAttributes</td>
<td>Adds or updates an item and its attributes, or adds attribute-value pairs to items that exist already. Items are automatically indexed as they’re received.</td>
</tr>
<tr>
<td>BatchPutAttributes</td>
<td>For greater overall throughput of bulk writes, performs up to 25 PutAttribute operations in a single call.</td>
</tr>
<tr>
<td>DeleteAttributes</td>
<td>Deletes an item, an attribute, or an attribute value.</td>
</tr>
<tr>
<td>GetAttributes</td>
<td>Retrieves an item and all or a subset of its attributes and values.</td>
</tr>
<tr>
<td>Select</td>
<td>Queries the data set in the familiar “Select target from domain_name where query_expression” syntax. Supported value tests are =, !=, &lt;, &gt;, &lt;=, &gt;=, like, not like, between, is null, isn’t null, and every(). Example: select * from mydomain where every(keyword) = &quot;Book&quot;. Orders results using the SORT operator, and counts items that meet the condition(s) specified by the predicate(s) in a query using the Count operator.</td>
</tr>
</tbody>
</table>
# SQL vs NoSQL

<table>
<thead>
<tr>
<th>Database use</th>
<th>Challenges faced with a cloud database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactional support and referential integrity</td>
<td>Applications using cloud databases are largely responsible for maintaining the integrity of transactions and relationships between tables.</td>
</tr>
<tr>
<td>Complex data access</td>
<td>Cloud databases (and ORM in general) excel at single-row transactions: get a row, save a row, and so on. But most nontrivial applications have to perform joins and other operations that cloud databases can’t.</td>
</tr>
<tr>
<td>Business Intelligence</td>
<td>Application data has value not only in terms of powering applications but also as information that drives business intelligence. The dilemma of the pre-relational database, in which valuable business data was locked inside impenetrable application data stores, isn’t something to which business will willingly return.</td>
</tr>
</tbody>
</table>
Take Home Messages

- The challenges and opportunities of big data
- Large-scale distributed file systems
  - GFS (HDFS)
- Large-scale distributed access structures
  - Dynamo (Voldemort), BigTable (HBase)
- NoSQL databases