NoSQL Technologies and Introduction to ElasticSearch

COMP9313: Big Data Management

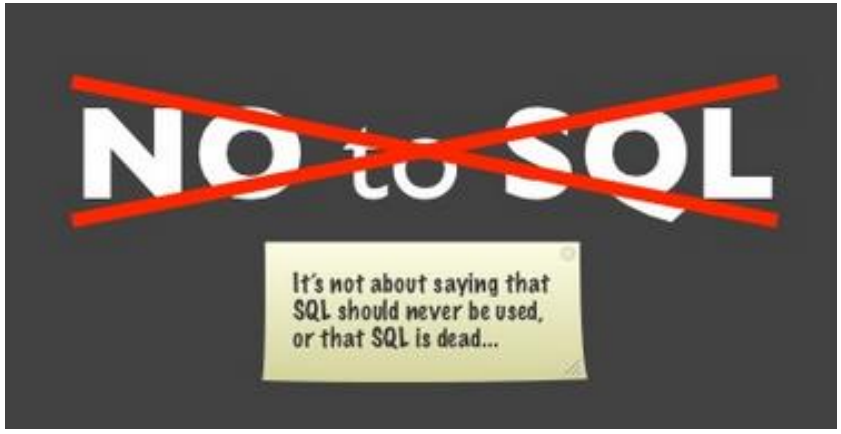
Introduction to NoSQL

What does RDBMS provide?

- Relational model with schemas
- Powerful, flexible query language (SQL)
- Transactional semantics: ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability)
- Rich ecosystem, lots of tool support (MySQL, PostgreSQL, etc.)

What is NoSQL?

- The name stands for Not Only SQL
- Does not use SQL as querying language
- Class of non-relational data storage systems
- The term NOSQL was introduced by Eric Evans when an event was organized to discuss open source distributed databases
- It's not a replacement for a RDBMS but compliments it
- All NoSQL offerings relax one or more of the ACID properties (will talk about the CAP theorem)



What is NoSQL?

- Key features (advantages):
  - non-relational
  - doesn't require strict schema
  - data are replicated to multiple nodes (so, identical & fault-tolerant) and can be partitioned
  - down nodes easily replaced
  - no single point of failure
  - horizontal scalability
  - cheap, easy to implement (for open-source solutions)
  - massive write performance
  - fast key-value access

Why NoSQL?

- Web apps have different needs (than the apps that RDBMS were designed for)
  - Low and predictable response time (latency)
  - Scalability & elasticity (at low cost!)
  - High availability
  - Flexible schemas / semi-structured data
  - Geographic distribution (multiple datacenters)

- Web apps can (usually) do without
  - Transactions / strong consistency / integrity
  - Complex queries

Who are using NoSQL?

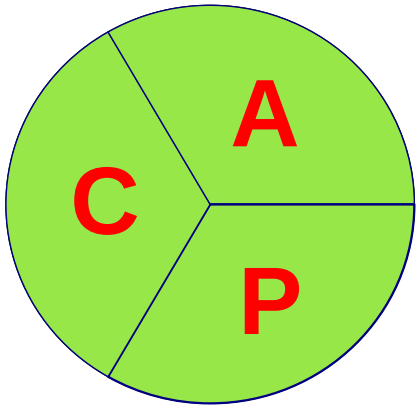
- Google (BigTable)
- LinkedIn (Voldemort)
- Facebook (Cassandra)
- Twitter (HBase, Cassandra)
- Baidu (HyperTable)

Three Major Papers for NoSQL

- Three major papers were the seeds of the NoSQL movement
  - BigTable, 2006 (Google)
  - Dynamo, 2007 (Amazon)
    - Ring partition and replication
    - Gossip protocol (discovery and error detection)
    - Distributed key-value data store
    - Eventual consistency
  - CAP Theorem, 2002 (discuss in the next few slides)

CAP Theorem

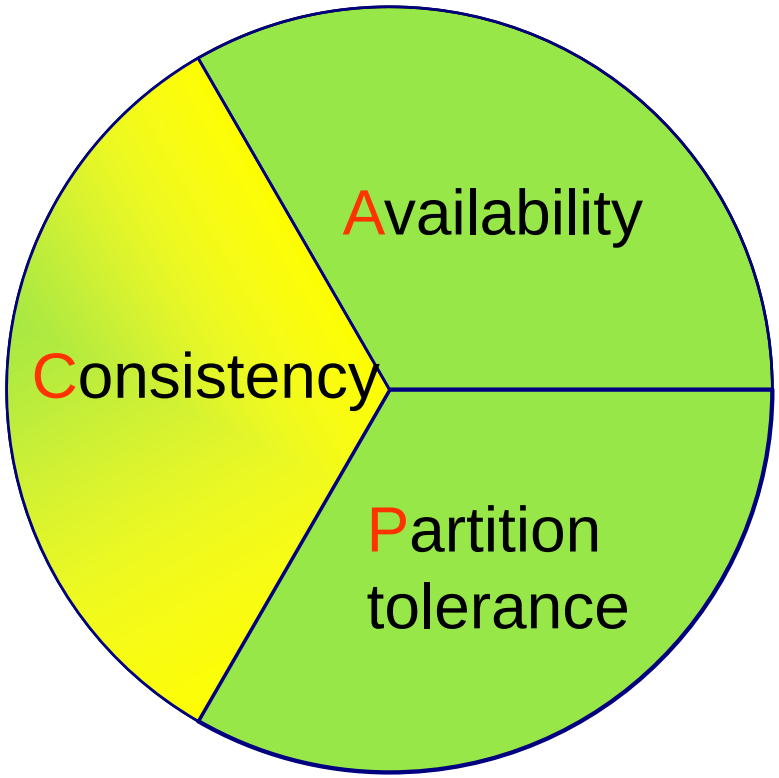
- Consider these three properties of a distributed system (sharing data)
  - Consistency:
    - all copies have same value
  - Availability:
    - reads and writes always succeed
  - Partition-tolerance:
    - system properties (consistency and/or availability) hold even when network failures prevent some machines from communicating with others



CAP Theorem

- Brewer's CAP Theorem:
  - *For any system sharing data, it is "impossible" to guarantee simultaneously all of these three properties*
  - You can have at most two of these three properties for any shared-data system
- 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)

CAP Theorem: Consistency



All clients always have the same view of the data

Once a writer has written data, all readers will see that data

- Two kinds of consistency:
  - strong consistency – **ACID** (Atomicity Consistency Isolation Durability)
  - weak consistency – **BASE** (Basically Available Soft-state Eventual consistency)

ACID & CAP

- **ACID**
  - A DBMS is expected to support “ACID transactions,” processes that are:
  - **A**tomicity: either the whole process is done or none is
  - **C**onsistency: only valid data are written
  - **I**solation: one operation at a time
  - **D**urability: once committed, it stays that way

- **CAP**
  - **C**onsistency: all data on cluster has the same copies
  - **A**vailability: cluster always accepts reads and writes
  - **P**artition tolerance: guaranteed properties are maintained even when network failures prevent some machines from communicating with others

Consistency Model

- A consistency model determines rules for visibility and apparent order of updates
- Example:
  - Row *X* is replicated on nodes *M* and *N*
  - Client *A* writes row *X* to node *N*
  - Some period of time *t* elapses
  - Client *B* reads row *X* from node *M*
  - **Does client *B* see the write from client *A*?**
  - Consistency is a continuum with tradeoffs
  - **For NOSQL, the answer would be: "maybe"**
  - CAP theorem states: *"strong consistency can't be achieved at the same time as availability and partition-tolerance"*

Eventual Consistency

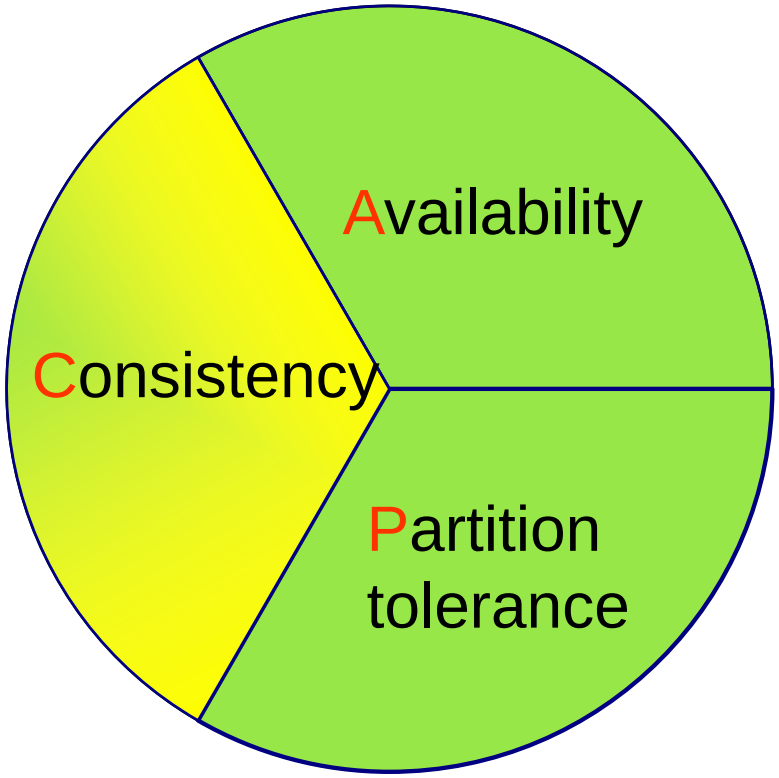
- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency)
- http://en.wikipedia.org/wiki/Eventual_consistency

Eventual Consistency

- The types of large systems based on **CAP** aren't **ACID** they are **BASE** (http://queue.acm.org/detail.cfm?id=1394128):
  - Basically Available - system seems to work all the time
  - Soft State - it doesn't have to be consistent all the time
  - Eventually Consistent - becomes consistent at some later time

- Everyone who builds big applications builds them on **CAP** and **BASE**: Google, Yahoo, Facebook, Amazon, eBay, etc.

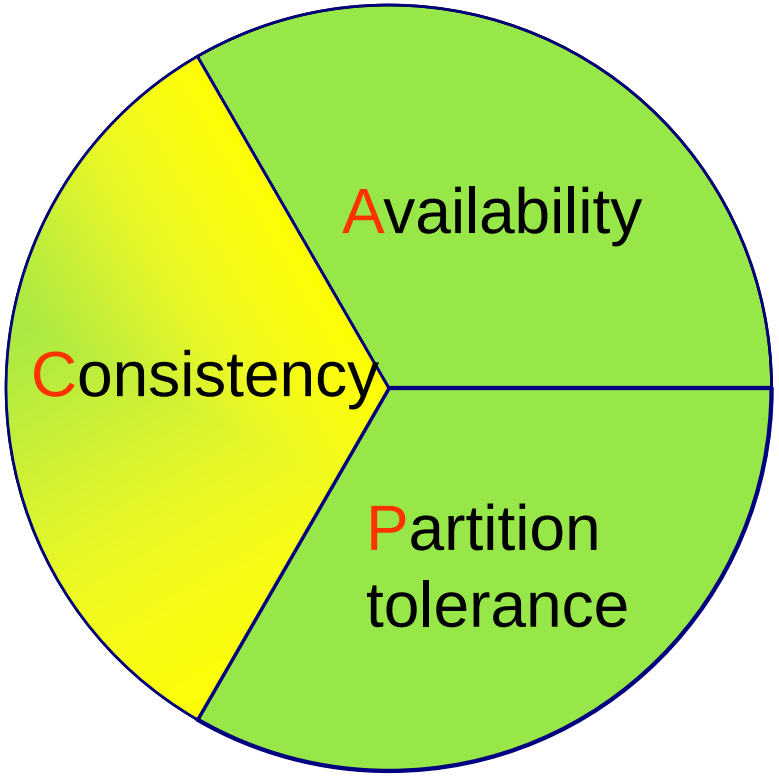
CAP Theorem: Availability



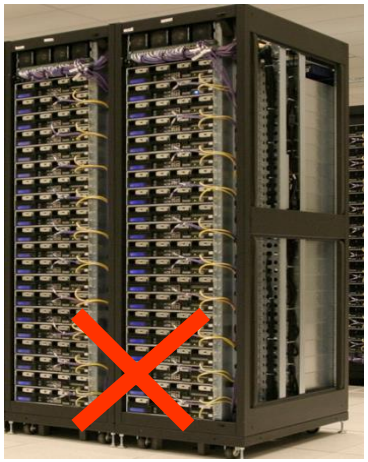
System is available during software and hardware upgrades and node failures

- Traditionally, thought of as the server/process available five 9's (99.999 %).
  - However, for large node system, at almost any point in time there's a good chance that a node is either down or there is a network disruption among the nodes.
  - Want a system that is resilient in the face of network disruption

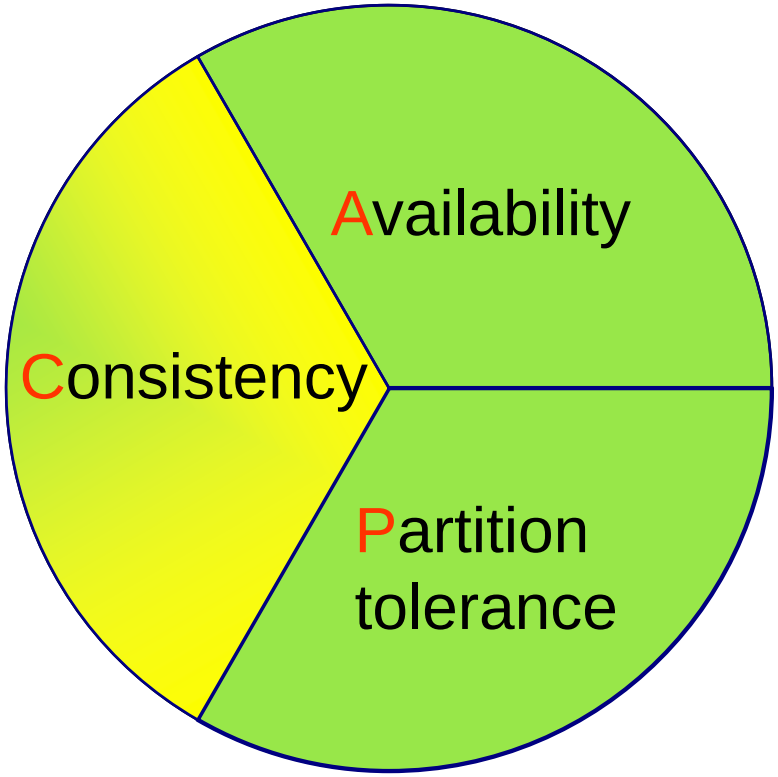
CAP Theorem: Partition-Tolerance



A system can continue to operate in the presence of a network partitions.



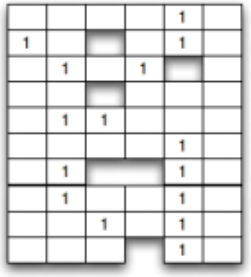
CAP Theorem

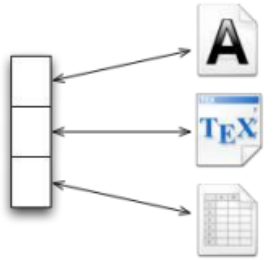


CAP Theorem: You can have at most **two** of these properties for any shared-data system

No SQL Taxonomy

- Key-Value stores
  - Simple K/V lookups (Distributed Hash Table (DHT))
- Column stores
  - Each key is associated with many attributes (columns)
  - NoSQL column stores are actually hybrid row/column stores
    - Different from “pure” relational column stores!
- Document stores
  - Store semi-structured documents (JSON)
- Graph databases
  - Neo4j, etc.
  - Not exactly NoSQL
    - can’t satisfy the requirements for High Availability and Scalability/Elasticity very well





Key-value

- Focus on scaling to huge amounts of data
- Designed to handle massive load
- Based on Amazon's Dynamo paper
- Data model: (global) collection of Key-value pairs
- Example: (DynamoDB)
  - *items* having one or more attributes (name, value)
  - An *attribute* can be single-valued or multi-valued like set
  - items are combined into a *table*

Key-value

- Basic API access:
  - get(key): extract the value given a key
  - put(key, value): create or update the value given its key
  - delete(key): remove the key and its associated value
  - execute(key, operation, parameters): invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc.)

Key-value

- Pros:
  - very fast
  - very scalable (horizontally distributed to nodes based on key)
  - simple data model
  - eventual consistency
  - fault-tolerant
- Cons
  - Can’t model more complex data structure such as objects

Key-value

Name	Producer	Data model	Querying
SimpleDB	Amazon	set of pairs (key, {attribute}), where attribute is a pair (name, value)	restricted SQL; select, delete, GetAttributes, and PutAttributes operations
Redis	Salvatore Sanfilippo	set of pairs (key, value), where value is simple typed value, list, ordered (according to ranking) or unordered set, hash value	primitive operations for each value type
Dynamo	Amazon	like SimpleDB	simple get operation and put in a context
Voldemort	LinkeIn	like SimpleDB	similar to Dynamo

Document-based

- Can model more complex objects
- Inspired by Lotus Notes
- Data model: collection of documents
- Document: **JSON** (**J**ava**S**cript **O**bject **N**otation is a data model, key-value pairs, which supports objects, records, structs, lists, array, maps, dates, Boolean with **nesting**), **XML**, other semi-structured formats.
- Example: (MongoDB) document

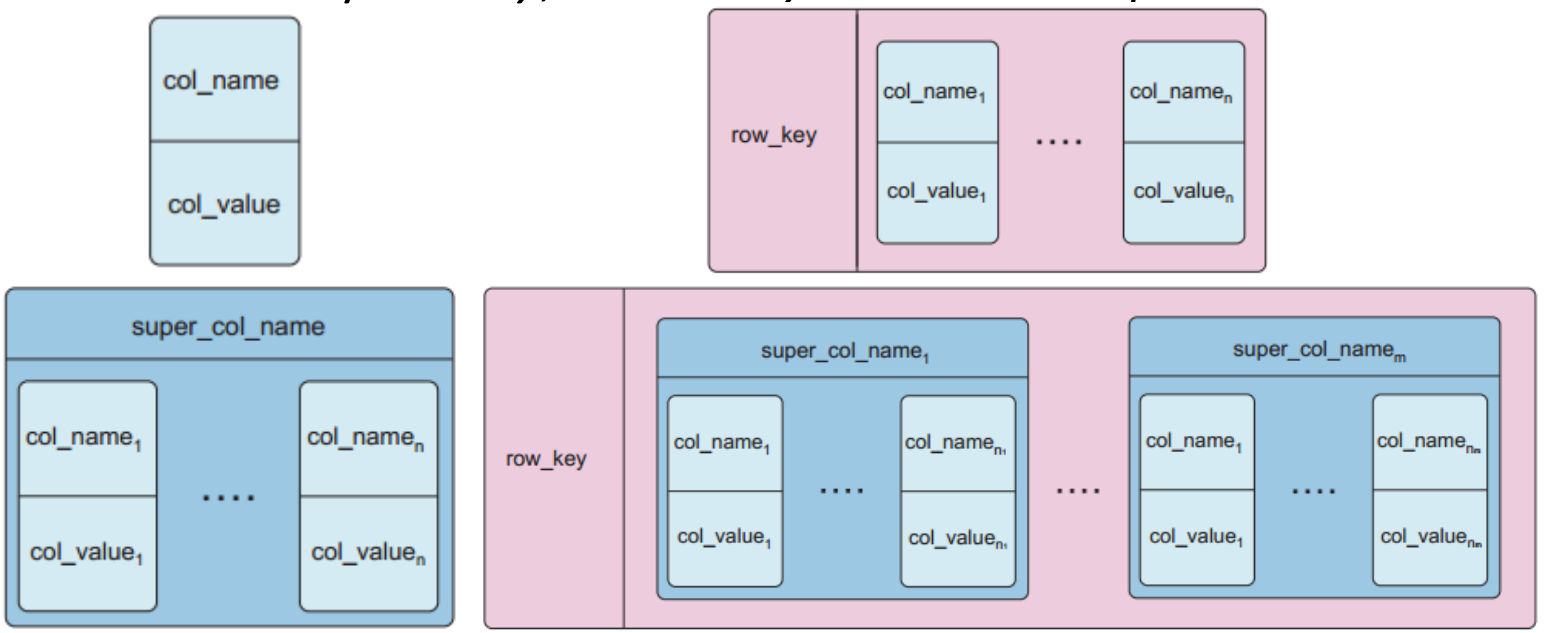
```
{
   Name: "Jaroslav",
   Address: "Malostranske nám. 25, 118 00 Praha 1”,
   Grandchildren: {Claire: "7", Barbara: "6", "Magda: "3",
         Kirsten: "1", Otis: "3", Richard: "1"}
   Phones: [ “123-456-7890”, “234-567-8963” ]
 }
```

Document-based

Name	Producer	Data model	Querying
MongoDB	10gen	object-structured documents stored in collections; each object has a primary key called ObjectId	manipulations with objects in collections (find object or objects via simple selections and logical expressions, delete, update, etc.)
Couchbase	Couchbase	document as a list of named (structured) items (JSON document)	by key and key range, views via Javascript and MapReduce
ElasticSearch	Elastic.co	documents (JSON), stored in indexes	REST APIs, support for both query string and request body queries (using DSL)

Column-based

- Based on Google's BigTable paper
- Like column oriented relational databases (store data in column order) but with a twist
- Tables: Similar to RDBMS, but handle semi-structured data
- Data model:
  - Collection of Column Families
  - Column family = (key, value) where value = set of **related** columns (standard, super)
  - indexed by *row key*, *column key* and *timestamp*



Column-based

- One column family can have variable numbers of columns
- Cells within a column family are sorted "physically"
- Very sparse, most cells have null values
- Comparison: RDBMS vs column-based NoSQL
  - Query on multiple tables
    - RDBMS: must fetch data from several places on disk and glue together
    - Column-based NoSQL: only fetch column families of those columns that are required by a query (all columns in a column family are stored together on the disk, so multiple rows can be retrieved in one read operation data locality)

Column-based

Name	Producer	Data model	Querying
BigTable	Google	set of pairs (key, {value})	selection (by combination of row, column, and time stamp ranges)
HBase	Apache	groups of columns (a BigTable clone)	JRUBY IRB-based shell (similar to SQL)
Hypertable	Hypertable	like BigTable	HQL (Hypertext Query Language)
CASSANDRA	Apache (originally Facebook)	columns, groups of columns corresponding to a key (supercolumns)	simple selections on key, range queries, column or columns ranges
PNUTS	Yahoo	(hashed or ordered) tables, typed arrays, flexible schema	selection and projection from a single table (retrieve an arbitrary single record by primary key, range queries, complex predicates, ordering, top-k)

Graph-based

- Focus on modeling the structure of data (*interconnectivity*)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory (G=(E,V))
- Data model:
  - (Property Graph) nodes and edges
    - Nodes may have properties (including ID)
    - Edges may have labels or roles
  - Key-value pairs on both
- Interfaces and query languages vary
- Example:
  - Neo4j, FlockDB, InfoGrid …

NoSQL Pros and Cons

- Advantages
  - Massive scalability
  - High availability
  - Lower cost (than competitive solutions at that scale)
  - (usually) predictable elasticity
  - Schema flexibility, sparse & semi-structured data
- Disadvantages
  - Doesn't fully support relational features
    - no join, group by, order by operations (except within partitions)
    - no referential integrity constraints across partitions
  - Eventual consistency is not intuitive to program for
    - Makes client applications more complicated
  - Not always easy to integrate with other applications that support SQL
  - Relaxed ACID (see CAP theorem) → fewer guarantees

Introduction to ElasticSearch (Preliminaries)

source

Indexing Overview

- Why do we need indexing?
  - Much of the information is represented as text (Web pages, business documents, health records)
  - Searching can be done through linear scan, to a certain extent (e.g., using Unix's grep)
  - Linear scan has its limitations:
    - Scanning large collections of documents (with billions or trillions of words) becomes very slow for most applications (specially interactive ones)
    - More flexible operations might be impractical using grep (e.g. finding words that appear "near" to other words)
    - Ranked retrieval -> Rank retrieval results base on a given matching criteria.

C. Manning et al., "Introduction to information retrieval", Cambridge University Press, Online edition, 2009

Inverted Index

- Key idea -> An index that maps terms (e.g. words) to the documents it occurs

Inverted list

Terms	Documents
act	1, 4, 63, 77, 143, ...
Australia	2, 4, 89, 91, 231...
constitution	4, 8, 99, 107, 431...
...	...

dictionary
(terms)

postings list
(documents identified
by a docID)

Steps to Build an Inverted List

1. Collect documents that needs to be indexed
2. Turn documents in to a list of tokens (tokenization)
3. Perform preprocessing to produce a normalized list of tokens (e.g. stemming)
4. Create list of terms and the corresponding postings (documents) where they occur
5. Sort terms and postings
6. Record (in dictionary) stats such as document frequency

Steps to Build an Inverted List

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

⟹

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

⟹

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

C. Manning et al., "Introduction to information retrieval", Cambridge University Press, Online edition, 2009

Boolean queries using Inverted Index

- Example task: Locate documents where terms ”Caesar” and “Capitol” occur together.
- Boolean query: `"Caesar" AND "Capitol"`
- Steps:
  1. Locate `"Caesar"` in dictionary
  2. Retrieve postings where it appears
  3. Locate `"Capitol"` in dictionary
  4. Retrieve postings where it appears
  5. Perform the intersection between the two postings lists

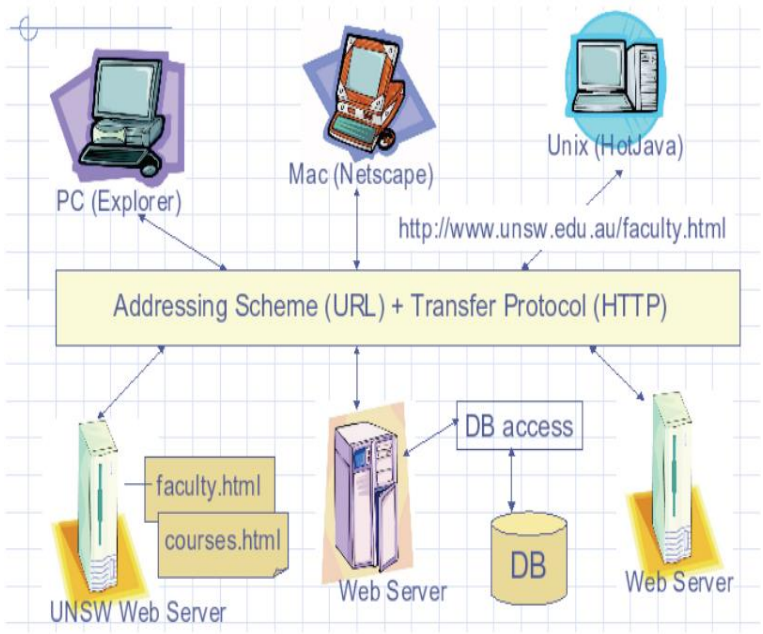
C. Manning et al., “Introduction to information retrieval”, Cambridge University Press, Online edition, 2009

REST API

Web Essentials

- Web = Higher Level Protocol over the Internet
- Three basic components of the Web:
- A Uniform Notation Scheme for addressing resources (Uniform Resource Locator - URL)
- A protocol for transporting messages (HyperText Transport Protocol - HTTP)
- A markup language for formatting hypertext documents (HyperText Markup Language – HTML)

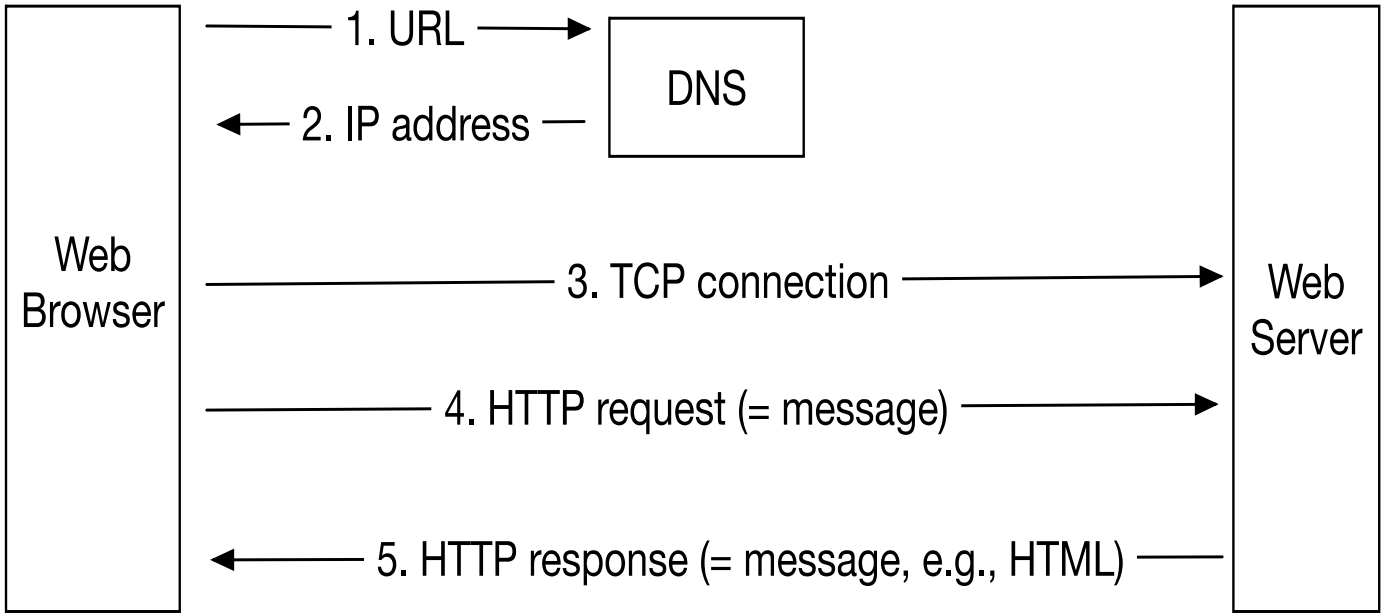
- In the end, these building blocks become the essentials of the Web applications as well.



Web Essentials

The building blocks of the Web

A Typical “Web” Interaction (all components together)



The basic “Web” communication model is HTTP request-response

An URL (HTTP scheme)

`http://www.example.org:56789/a/b/c.txt?t=win&s=chess#para5`

authority part:

- after 'http://' through to the next slash - www.example.org:56789
- It consists of either a domain name or an IP address
- optionally followed by a port number (if omitted, port 80 is implied)

path part:

- after the authority through to question mark (?)
- /a/b/c.txt (/ is part of the path, ? is not)
- much like a file path in file system ...

query string part:

- after the path up to a number sign #
- contains a set of name-value pairs, separated by & (e.g., t=win&s=chess)

fragment identifier part:

- after the number sign #, not including #

Web Essentials - HTTP

HTTP Request (from browser to server):

It is composed of `Request Line + Header + (additional data)`

Syntax for the Request Line:

`Request-Method sp Request-URI sp HTTP-version CRLF`

eg, GET `http://www.smh.com.au/index.html` HTTP/1.1

- There must be a newline (CRLF) between the header and the additional data part.
- Common Request methods: GET, POST, HEAD ...
- Request header: `User-Agent`, `Referer`, `Authorization`.
- Additional data (body): parameters (POST), block of data

You can utilise many parts of these HTTP request data to be more effective

HTTP Request Methods (version 1.1)

Method	Description
GET	It the simplest, most used. It simply retrieves the data identified by the URL. If the URL refers to a script (CGI, servlet, and so on), it returns the data produced by the script.
HEAD	It only returns HTTP headers without the document body.
POST	It is like GET. Typically, POST is used in HTML forms. POST is used to transfer a block of data to the server.
OPTIONS	It is used to query a server about the capabilities it provides. Queries can be general or specific to a particular resource.
PUT	It stores the body at the location specified by the URI. It is similar to the PUT function in FTP.
DELETE	It is used to delete a document from the server. The document to be deleted is indicated in the URI section of the request.
TRACE	It is used to tract the path of a request through firewall and multiple proxy servers. TRACE is useful for debugging complex network problems and is similar to the traceroute tool.

Web Essentials - HTTP

HTTP Response (from server to browser):

- Composed of `Status Line + Header + Body`
- Status line: 200 OK, 404 Not Found, etc.
- Header:
  - Content-Type, Content-Language, Content-Length, Cache-control, etc.
- Body:
  - Body contains the requested data
  - Body is in specific MIME format (eg., text/HTML)
  - MIME (Multipurpose Internet Mail Extension): text (plain, HTML), multimedia data, applications such as PDF, PowerPoint, etc.

JSON/REST is ‘preferred’ choice

GET /stockquote/DIS HTTP/1.1

Host: www.stockquoteserer.com

Accept: application/json

HTTP/1.1 200 OK

Content-Type: application/json

Content-Length: xxx

```
{
  “ticker”: “DIS”,
  “price”: 34.5
}
```

ElasticSearch

Elasticsearch

source

- Open source search engine based on Apache Lucene
- Initial release in 2010
- Provides a distributed, full-text search engine with a REST APIs
  - E.g. GET http://localhost:9200/person/student/8871

Elasticsearch

source

- Document oriented (JSON as serialization format for documents)
- Developed in Java (cross platform)
- Focused on scalability – distributed by design
- Highly efficient search

Elasticsearch Use Cases

- E-commerce
  - Online web stores
  - Fast search for products
  - Autocomplete suggestions
- Storage, analysis and mining of transaction data
  - Trends
  - Statistics
  - Summarizations
- Analytics/Business intelligence
  - Investigation
  - Analysis
  - Visualization
  - Ad-hoc business questions

Elasticsearch Elements

- Cluster
  - An Elasticsearch cluster is a collection of nodes (servers)
  - Identified by a unique name
  - Data is stored in this collection of nodes
  - Provide indexing and search capabilities across all nodes

Elasticsearch Elements

- Node
  - A single server in the cluster
  - Identified by a unique name
  - Stores all or parts of the whole dataset
  - Contributes to the indexing and search capabilities of Elasticsearch

Elasticsearch Elements

- Shard
  - Individual instances of Lucene index
  - Elasticsearch leverages Lucene indexing in a distributed system
- Index
  - Distributed across shards
  - Collection of documents (e.g. person, employee, etc.)
  - Identifiable by a name
  - Replicas (fault tolerance)
  - Analogy to RDMS: Index → Database

Elasticsearch Elements

- Type
  - Category of documents of the same class (e.g. product, employee)
  - Types have a name and mapping
  - Indexes can have one or more types
  - Analogy to RDMS: Type → Table

Elasticsearch Elements

- Mapping
  - Defines the fields contained in a given Type
  - Describes data type for each field (e.g. String, Integer, etc.)
  - Describes how fields must be indexed and stored
  - Dynamic mapping is possible
  - Analogy to RDMS: Mapping → Schema of Table

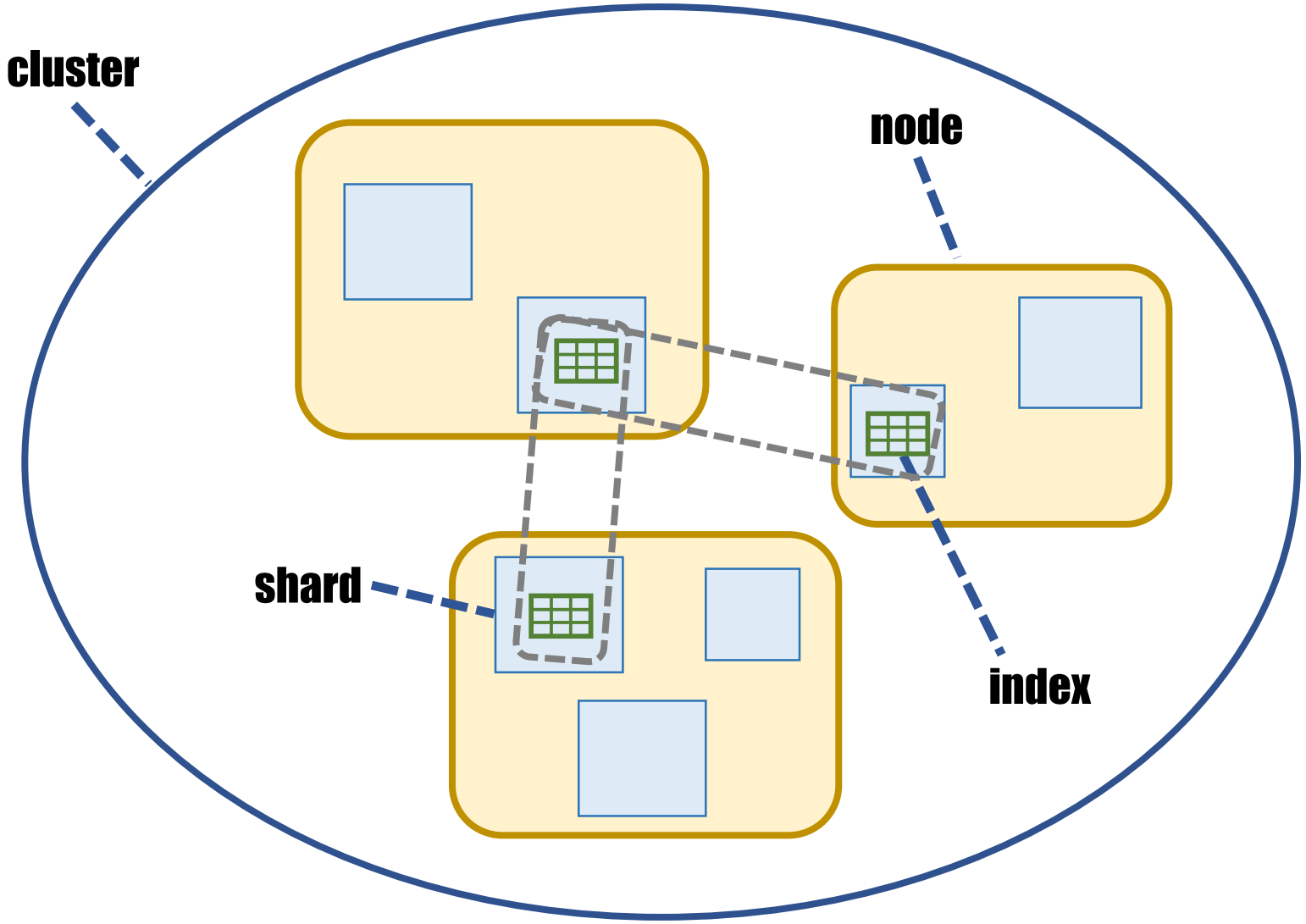
Elasticsearch Elements

- Document
  - Basic unit of information
  - Documents contain fields (key/value pairs)
  - ElasticSearch uses JSON to represent documents
  - Analogy to RDMS: Document → Tuple

Elasticsearch Elements

- Replicas
  - Copy of a shard
  - Provides fault tolerance (shards and node failures)
  - Scalability -> Queries can be executed in parallel
  - Default ElasticSearch configuration:
    - 5 primary shards
    - 1 replica for each index

Elasticsearch Ecosystem



Search APIs

- Querying using **query strings** (HTTP request)
  - Search the twitter index:

```
GET /twitter/_search?q=user:kimchy
```

  - Search all indices

```
GET /_all/tweet/_search?q=tag:wow
```

  - Search within specific types

```
GET /twitter/tweet,user/_search?q=user:joe
```

- Not all search options are available using this mode

https://www.elastic.co/guide/en/elasticsearch/reference/6.1/index.html

Search APIs

- Querying using ElasticSearch DSL

```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title":   "Search"        }},
        { "match": { "content": "Elasticsearch" }}
      ],
      "filter": [
        { "term":  { "status": "published" }},
        { "range": { "publish_date": { "gte": "2015-01-01" }}}
      ]
    }
  }
}
```

https://www.elastic.co/guide/en/elasticsearch/reference/6.1/index.html

Questions?