UNIT VI

NoSQL database

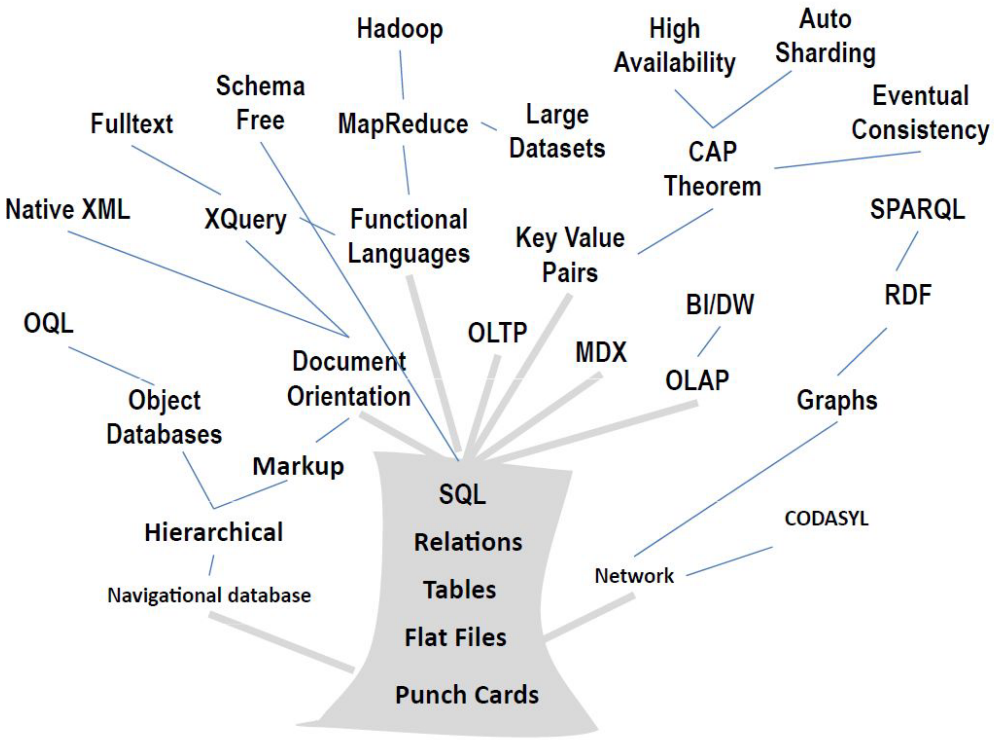
What is NOSQL?

- Stands for **N**ot **O**nly **SQL**
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties (will talk about the CAP theorem)
- NOSQL has become prominent with the advent of web scale data and systems created by Google, Facebook, Amazon, Twitter and others to manage data for which SQL was not the best fit.

- For data storage, an RDBMS cannot be the be-all/end-all
- Just as there are different programming languages, need to have other data storage tools in the toolbox
- A NoSQL solution is more acceptable to a client now than even a year ago
  - Think about proposing a Ruby/Rails or Groovy/Grails solution now versus a couple of years ago

What is NOSQL?

- Definitions for NOSQL vary greatly from newer systems using document stores, key value stores, XML databases, graph databases, column stores, object stores, etc. (like MongoDB, Cassandra, Couchbase, Hadoop, etc.) to older Hierarchical systems that had many similar characteristics (like Cache and GT.M)
- The NOSQL concept tree illustrates the variety of concepts related to NOSQL.



NOSQL Concept Tree

Source: CIO's Guide to NOSQL, Dan McCreary, June 2012

How does NoSQL vary from RDBMS?

- Looser schema definition
- Applications written to deal with specific documents/ data
  - Applications aware of the schema definition as opposed to the data
- Designed to handle distributed, large databases
- Trade offs:
  - No strong support for ad hoc queries but designed for speed and growth of database
    - Query language through the API
  - Relaxation of the ACID properties

Benefits of NoSQL

Elastic Scaling

- RDBMS scale up – bigger load , bigger server
- NO SQL scale out – distribute data across multiple hosts seamlessly

DBA Specialists

- RDMS require highly trained expert to monitor DB
- NoSQL require less management, automatic repair and simpler data models

Big Data

- Huge increase in data RDMS: capacity and constraints of data volumes at its limits
- NoSQL designed for big data

Benefits of NoSQL

Flexible data models

- Change management to schema for RDMS have to be carefully managed
- NoSQL databases more relaxed in structure of data
  - Database schema changes do not have to be managed as one complicated change unit
  - Application already written to address an amorphous schema

Economics

- RDMS rely on expensive proprietary servers to manage data
- No SQL: clusters of cheap commodity servers to manage the data and transaction volumes
- Cost per gigabyte or transaction/second for NoSQL can be lower than the cost for a RDBMS

Drawbacks of NoSQL

- Support
  - RDBMS vendors provide a high level of support to clients
    - Stellar reputation
  - NoSQL – are open source projects with startups supporting them
    - Reputation not yet established
- Maturity
  - RDMS mature product: means stable and dependable
    - Also means old no longer cutting edge nor interesting
  - NoSQL are still implementing their basic feature set

Drawbacks of NoSQL

- **Administration**
  - RDMS administrator well defined role
  - No SQL's goal: no administrator necessary however NO SQL still requires effort to maintain
- **Lack of Expertise**
  - Whole workforce of trained and seasoned RDMS developers
  - Still recruiting developers to the NoSQL camp
- **Analytics and Business Intelligence**
  - **RDMS designed to address this niche**
  - NoSQL designed to meet the needs of an Web 2.0 application - not designed for ad hoc query of the data
    - Tools are being developed to address this need

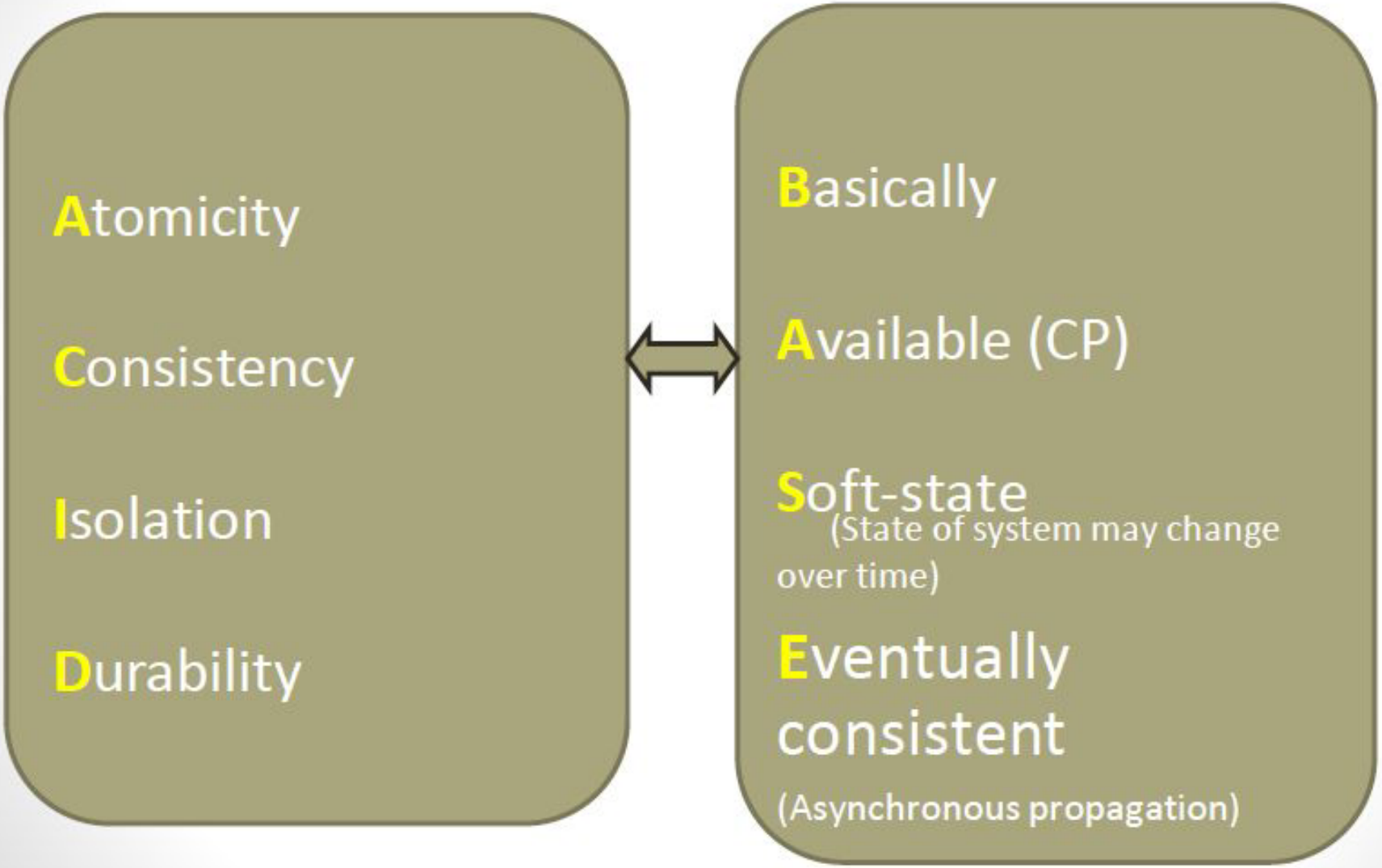
Structured Vs Unstructured Data

- **Structured data is data that has been predefined and formatted to a set structure before being placed in data storage**, which is often referred to as schema-on-write. The best example of structured data is the relational database: the data has been formatted into precisely defined fields, such as credit card numbers or address, in order to be easily queried with SQL.

- Examples of **structured data**
- Structured data is an old, familiar friend. It's the basis for inventory control systems and ATMs. It can be human- or machine-generated.
- Common examples of machine-generated structured data are weblog statistics and point of sale data, such as barcodes and quantity. Plus, anyone who deals with data knows about spreadsheets: a classic example of human-generated structured data.

- **Unstructured data is data stored in its native format and not processed until it is used**, which is known as schema-on-read. It comes in a myriad of file formats, including email, social media posts, presentations, chats, IoT sensor data, and satellite imagery.

RDB ACID to NoSQL BASE



Pritchett, D.: BASE: An Acid Alternative (queue.acm.org/detail.cfm?id=1394128)

Atomicity and Consistency

Atomicity

Transactions are atomic – they don't have parts (conceptually)

can't be executed partially; it should not be detectable that they interleave with another transaction

Consistency

Transactions take the database from one consistent state into another.

In the middle of a transaction the database might not be consistent

Isolation and Durability

Isolation

The effects of a transaction are not visible to other transactions until it has completed

From outside the transaction has either happened or not

To me this actually sounds like a consequence of atomicity…

Durability

Once a transaction has completed, its changes are made permanent

Even if the system crashes, the effects of a transaction must remain in place

Business Intelligence refers to the information required to enhance **business**decision-making activities.

Data Analytics refers to modifying the raw **data** into a meaningful format.

The prime purpose of **business intelligence** is to provide support in decision making and help the organizations to grow their **business**.

Basically Available: This constraint states that the system does guarantee the availability of the data as regards CAP Theorem; there will be a response to any request. But, that response could still be 'failure' to obtain the requested data or the data may be in an inconsistent or changing state, much like waiting for a check to clear in your bank account.

Soft state: The state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft.'

Eventual consistency: The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

CAP THEOREM

- The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **consistency** (among replicated copies), **availability** (of the system for read and write operations) and **partition tolerance** (in the face of the nodes in the system being partitioned by a network fault).

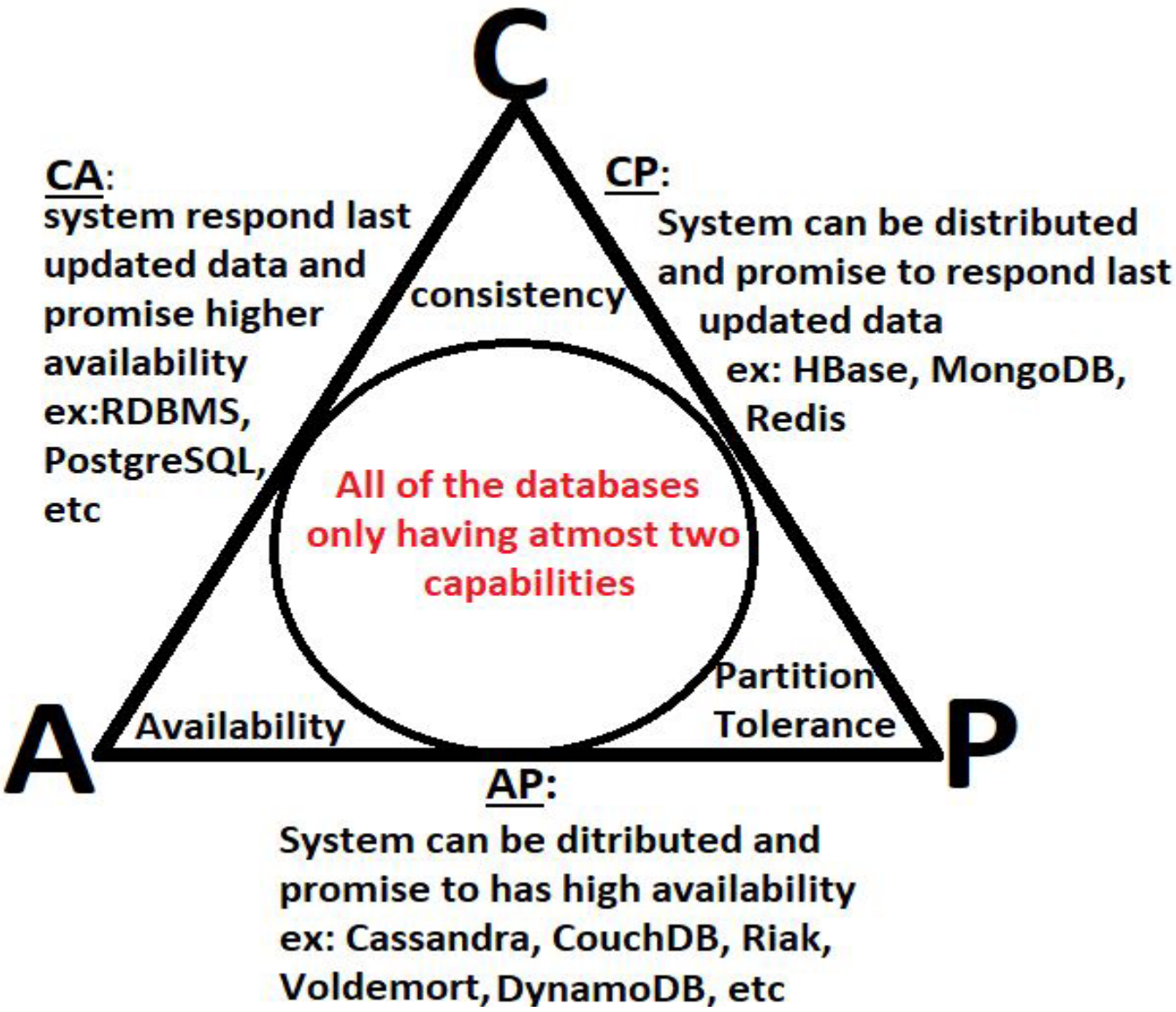
- The CAP theorem states that it is not possible to guarantee all three of the desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication.
- The theorem states that networked shared-data systems can only strongly support two of the following three properties:

- **Consistency** – Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions. A guarantee that every node in a distributed cluster returns the same, most recent and a successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP refers to sequential consistency, a very strong form of consistency.

- **Availability** – Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. Every non-failing node returns a response for all the read and write requests in a reasonable amount of time. The key word here is “every”. In simple terms, every node (on either side of a network partition) must be able to respond in a reasonable amount of time.

- **Partition Tolerance –** Partition tolerance means that the system can continue operating even if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. That means, the system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

- The CAP theorem states that distributed databases can have at most two of the three properties: consistency, availability, and partition tolerance. As a result, database systems prioritize only two properties at a time.


C
CA:
system respond last updated data and promise higher availability
ex:RDBMS, PostgreSQL, etc
CP:
System can be distributed and promise to respond last updated data
ex: HBase, MongoDB, Redis
consistency
All of the databases only having atmost two capabilities
A
Availability
Partition Tolerance
P
AP:
System can be ditributed and promise to has high availability
ex: Cassandra, CouchDB, Riak, Voldemort, DynamoDB, etc