NoSQL

Chapter contents:

- NoSQL: General principles
- NoSQL data models: JSON (and XML)
- Data exchange and serialization

Table of contents

Latency numbers every programmer should know (J.Dean)

Latency numbers (2012)

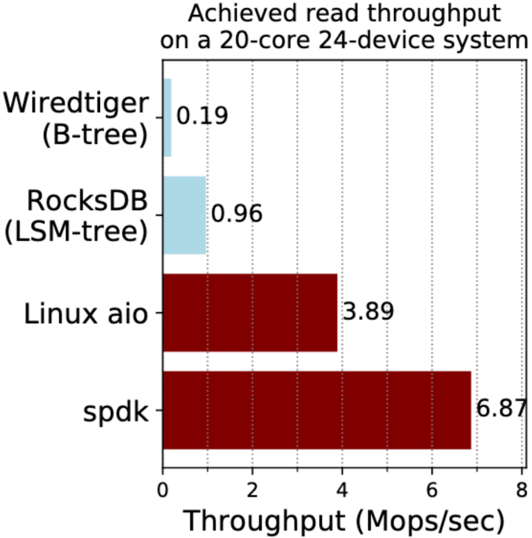
```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns
Mutex lock/unlock ........................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

2019: for a 4kB block:

Latency numbers (2019)

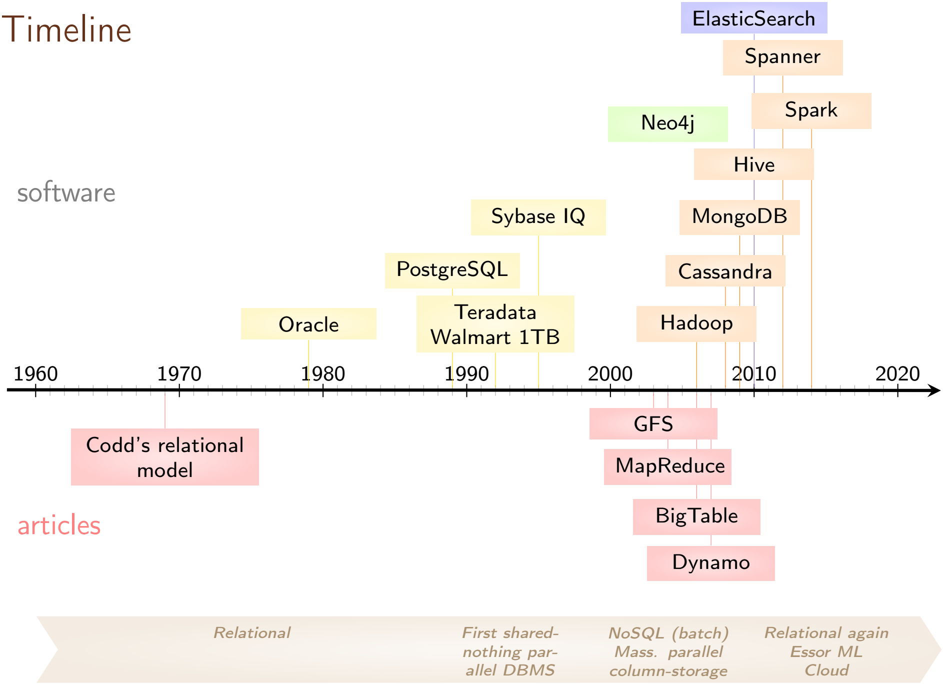
```
Fast NVMe (Optane) ........................... 7 µs
Fast NVMe (Z-SSD) ........................... 12 µs
Round trip TCP packet on 10Gb Ethernet ... 20-50 µs
NVMe Flash SSD .............................. 80 µs
```

Throughput numbers on key-value stores

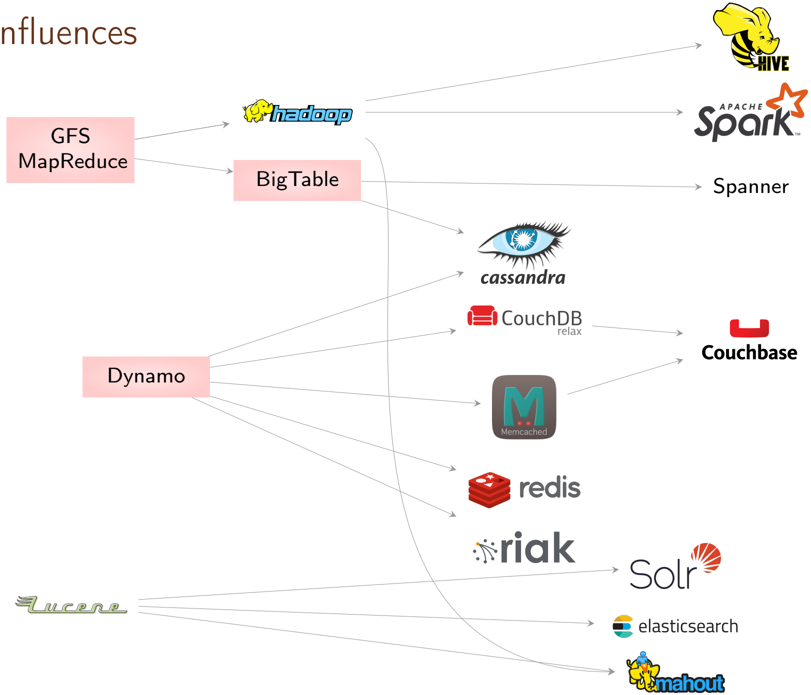


[https://www.usenix.org/system/files/fast19-kourtis.pdf]

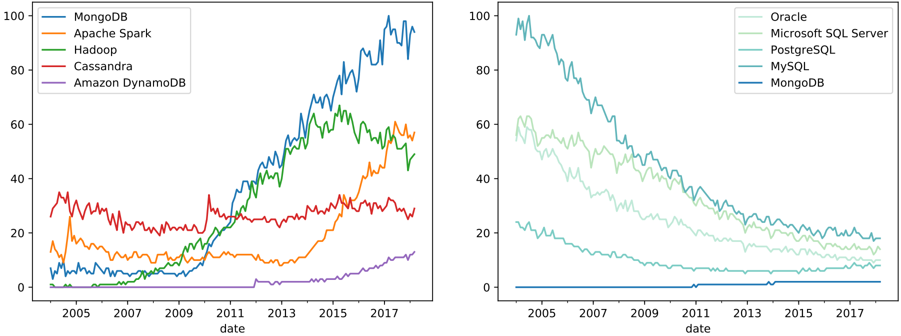
Timeline


ElasticSearch
Spanner
Spark
Neo4j
Hive
software
Sybase IQ
MongoDB
PostgreSQL
Cassandra
Oracle
Teradata Walmart 1TB
Hadoop
1960
1970
1980
1990
2000
2010
2020
GFS
Codd's relational model
MapReduce
BigTable
articles
Dynamo
Relational
First shared-nothing parallel DBMS
NoSQL (batch) Mass. parallel column-storage
Relational again Essor ML Cloud

Influences


GFS
MapReduce
hadoop
BigTable
HIVE
APACHE
Spark
Spanner
cassandra
CouchDB
relax
Couchbase
Dynamo
Memcached
redis
riak
Solr
Lucene
elasticsearch
mahout

Google trends



Google Trends: search comparison

Distributed Databases

Why *distribute*?

- parallelism (=performance)
- scalability
- availability: accessibility and fault tolerance (cloud)
- optimize for different hardware, distribute geographically,...

How to distribute?

- **sharding** $\simeq$ horizontal partitioning
- but also **replication**

Implementation challenges:

- decentralized architecture maintain coherence between copies, task and data partitioning
- **Shared nothing architecture** (nots *shared disk*, not *shared memory pool*). how to chose the partitioning

Types of parallelism

Parallelism in DBMS has a long history:

- **inter-operator:** every CPU computes a query operation (pipeline). Volcano model – a query operation sends the output directly to the next operation.
- **intra-operator:** every CPU computes the entire query on a part of the data
- **inter-query:** several queries executed in parallel

Since then:

- large scale data – distributed computing on a large amount of computers.
- ***Shared nothing architecture*** (neither *shared disk* nor *shared memory pool*).

Replication

Objectives: reliability, read performance.

Techniques:

- RAM+logs on disk: write-ahead logs (WAL)
- generally, asynchronous (eventual consistency)
- sometimes synchronized (but can have slow updates)
- versioning (vector clocks)
- network state (faults,...): gossip
- fault recovery: consensus (Paxos)

Ex: MongoDB: asynchronous, WAL.

In distributed DBMS:

PostgreSQL (WAL), MariaDB, Oracle (materialized views), SQL Server...

Often admin level choices (number of masters, synchronization,...).

Challenges when distributing

- good data partition/replication
- coherence (trade-off between performance and integrity when dealing with reads and writes)
- distributing computation tasks
  (to minimize data exchange)
- fault tolerance
- transaction control
- data privacy

Distributed architectures

Master-slave

- MongoDB: server mongos/mongod
- HDFS: NameNode/DataNodes
- BigTable

Without master servers

- Dynamo
- Cassandra
- (BitTorrent)

Partitioning: *how to distribute data ?*

↪technique in DynamoDB: coherent hashing.

Consistency, Availability, Partition tolerance

Ideally, distributed database systems would need to provide 3 guarantees:

Consistency:

Every read request to the system receives data corresponding to the most recent read.

Availability:

The system must answer any query to the system, even if the answer is wrong or outdated.

Partition tolerance:

The system must answer queries even under arbitrary failures of distributed nodes or of messages between nodes.

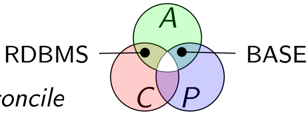
Consistency, Availability, Partition tolerance

CAP Theorem (Consistency, Availability, Partition tolerance):

A system cannot have all 3: when data is partitioned, we cannot guarantee both availability and consistency.

- In a distributed setting, ACID (Atomic, Consistence, Isolation, Durable) guarantees consistency while sacrificing availability.
- NoSQL DBs use rather the BASE consistency model:
  (Basic Availability, Soft state, Eventual consistency):
  - Basic availability: data is always available,
  - Soft-state: different copies are not always consistent
  - Eventual consistency: after a while (if no changes), the system is consistent

  In other words: the system has a mechanism to reconcile (sooner or later) all the versions.



NoSQL

NoSQL: Not SQL or (more often) Not Only SQL
Data model not fixed (and not relational) – e.g., *key/value pairs*, *value*: document or hashmap, . . .

Rough taxonomy (not standardized):

- Key-Value: (Redis, Memcached, Riak)
- Document: (MongoDB, CouchDB)
- Column: *Column-family*: Cassandra,
  Column-oriented: SAP Hana, MonetDB
- Graph: (Neo4j)

Principal characteristic: very vague classification, under constant evolution.

NoSQL (2)

- query language is no longer only SQL:
  method calls in programming languages (object+functional).
- new software stacks:
  - Web development:
    LAMP (Linux, Apache, Mysql, PHP) [1]
    MEAN (MongoDB, ExpressJS, AngularJS, NodeJS)
  - and for Big data? Not settled yet
    e.g, SMACK (Spark, Apache Mesos, Akka, Cassandra, and Apache Kafka)
- NoSQL means also no standard!

[1] +variants: other OS (Windows), server (Nginx), storage (MariaDB), script

NoSQL design principles

NoSQL is focused on data exchange and distribution.
We want *autonomous* data:

- denormalization
- no joins (autonomous documents)

Typical application: machine learning training data, graph data.

Typical NoSQL database: a “bunch” of documents
E.g., scientific papers: each article contains the entire information (authors, etc.)

Comparing NoSQL to relational databases

Roughly:

- ✔ easy to distribute migrate
- ✔ performance-oriented: scalability (>TB), real-time
- ✔ easy to design (usually, no modeling needed)
- ✔ high availability
- ✔ somewhat un-structured data (multimedia, graphs), for which relational DBs are not adapted
- ✔ very useful if many reads, few updates
- ✔ deals directly with programming APIs

- ✘ no standard query language: need to program
- ✘ asymmetric: some data access patterns are favoured
- ✘ no (or fixed) schema
- ✘ no transactions

Table of contents

Relational data model, XML, JSON

Heavily inspired by http://b3d.bdpedia.fr

Data format for structured data: XML and JSON.
Made principally for data exchange.

XML: eXtensible Markup Language
JSON: JavaScript Object Notation

XML

```xml
<personne>
  <nom>Dupond</nom>
  <tels>
    <tel>0612304056</tel>
    <tel>0269159002</tel>
  </tels>
</personne>
```

JSON

```json
{
"nom": "Dupond",
"tels": [0612304056, 0269159002]
}
```

- ✔ very rich ecosystem: XLST, XQuery, SVG, RSS
- ✘ verbose
- ✘ complex

- ✔ simple, compact
- ✘ limited (in terms of types)

JSON

JSON object:set of *key value pairs*

keys are strings

values can be:

- JSON object
- array of values
- string
- number
- boolean
- null

atomic types

Key value pairs exist in different versions:

JSON objects, XML elements, associative arrays (PHP), hash map (Java), dictionaries (Python)...

JSON validation

JSON


```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  },
  "mixed_list": ["aabb", 2018, true, [1,2,3]]
}}
```


Multiple validators available (also for XML, HTML, etc).
E.g.: `http://jsonlint.com`

JSON Schema

JSON

```json
{
  "checked": true,
  "dimensions": {
    "width": 5,
    "height": [1,2]
  }
}
```

JSON Schema

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "checked": {
      "$id": "/properties/checked",
      "type": "boolean",
      "title": "The Checked Schema ",
      "default": false,
      "examples": [ true ]
    },
...
}
```

JSON Schema describes (using JSON) the structure of a JSON object (like XSD for XML).

Examples:

- schema validator: `https://www.jsonschemavalidator.net`
- automatic schema inference: `https://www.jsonschema.net`

Table of contents

(De-)serializing JSON data (Javascript)

Sending data:

```
//JSON.stringify(v) converts value v into JSON string:
var myObj = { "nom":"Dupond", "age":31, "ville":"Paris" };
var myJSON = JSON.stringify(myObj);
window.location = "demo_json.php?x=" + myJSON;


JSON.stringify(new Date(2006, 0, 2, 15, 4, 5))
// ""2006-01-02T15:04:05.000Z""
```

Receiving data:

```
//JSON.parse(s) decodes string s (that represents JSON object)
//    into JavaScript object
var myJSON = '{ "nom":"Dupond", "age":31, "ville":"Paris" }';
var myObj = JSON.parse(myJSON);
document.getElementById("demo").innerHTML = myObj.nom;
// "Dupond"
```

(De-)serializing JSON data (Python)

For simple objects (dict, list...):

```python
import json

# converts a dict into JSON string:
mydict = {'nom': 'Dupond', 'age': 31, 'ville': 'Paris'}
myJson = json.dumps(data)

#  and conversely to recreate object:
json.loads(encoded_hand)
```

```python
import json

# serializes a dict into JSON file:
with open("data_file.json", "w") as f:
    json.dump(data, f)

# and parses JSON file into dict:
with open("data_file.json", "r") as f:
    data = json.load(f)
```

(De-)serializing JSON data (Python)

For a class:

```
import json
""" solution 1: define an encoder for the class"""
data = maclasse(...)
from json import JSONEncoder
class maclasseEncoder(JSONEncoder):
        def default(self, o):
            return o.__dict__ # not really robust

myJson = json.dumps(data, cls=maclasseEncoder)
# or myJSON = maclasseEncoder.encode(data)

""" solution2: make the class serializable by implementing the  toJson method"""
def toJson(self):
        return json.dumps(self, default=lambda o: o.__dict__)
myJson = json.dumps(data.toJson())

""" solution 3: use module 'jsonpickle' """
```

Data serialization framework (RPC)

Protocol Buffers (proto2, proto3): Google

- messages are key-value pairs
- define, in a file `.proto`, the message structure
- the message is then compiled in a programming language of choice
- compact format
- used by gRPC (gRPC+protobuf faster than Rest+JSON)

Thrift: then

- we define a message structure then we compile it into an object
- more languages available than `Protocol Buffers` (?)

Avro

- schema defined in JSON, dynamic, not compiled

Data exchange using REST APIs

Representational state transfer

Restful API contains:

- a resource description URI
- HTTP methods
- MIME types: JSON, XML, but also data structure (pages, sorting, ...) – can take considerable implementation effort

RPC API vs JSON HTTP API

RPC use cases:

- ✔ micro-services (RPC is low latency, high debit, so low network volume)
- ✔ streaming (real-time)

Disadvantages:

- ✘ cannot call services directly via HTTP (=browser)
- ✘ binary format, so not human readable

[https://docs.microsoft.com/fr-fr/aspnet/core/grpc/comparison]

Using a message broker

Publish & subscribe platforms

Intermediary between providers and receivers (subscribers)

- ✔ exchange is simplified (providers and receivers are independent)
- ✔ allows asynchronous communication

Compared to RPC:

- ✘ no direct communication
- ✘ sometimes needs heavy architecture (Kafka vs. gRPC)

message queue: each message read once, or *pub/sub* – only subscribers are read

Example of a message agent

- data streams
- guarantees receiving the message in the same order
- partitioning and replication
- objective: low latency, high debit

Uses a batch messaging principle via TCP

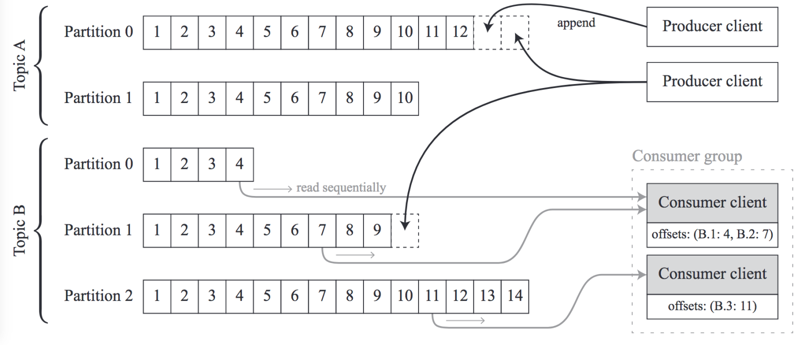


Figure 1: A Kafka *topic* is divided into *partitions*, and each partition is a totally ordered sequence of *messages*.

[https://www.repository.cam.ac.uk/bitstream/handle/1810/286031/kafka-debull15.pdf?sequence=1]

Column stores

4 example projects APACHE for serializing column data:

Arrow : *in-memory*

- allows direct access to column data, without the need to access row-by-row
- vectorization approach, random access,zero-copy
- Feather: adaptation of Arrow for file storagex

Parquet Parquet : *on disk*

- uses compression approaches

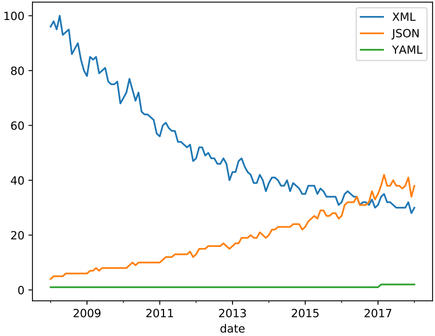
Kudu : *on disk (+ cache)*

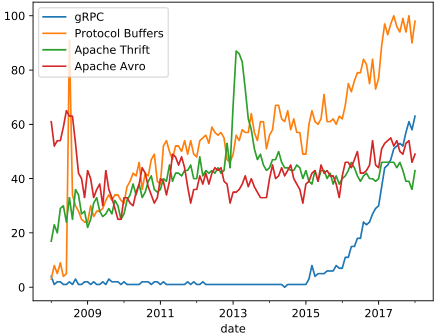
- optimised for updates

ORC

- used to optimise Hive and MapReduce
- compression, indexes, predicate pushdown

Google trends





Apache Avro started in 2009 !

Bibliography

- Semi-structured data
  https://www.w3schools.com/js/js_json_intro.asp

  An overview of JSON data model and the typical operations on JSON:
  JSON: data model, query languages and schema specification, Bourhis et al., PODS 2017
  https://arxiv.org/pdf/1701.02221.pdf

  http://b3d.bdpedia.fr/docstruct.html

  An experimental comparison of serialization mechanisms:
  http://labs.criteo.com/2017/05/serialization/

  Course slides on Protocol Buffers, Thrift, Avro:
  https://ganges.usc.edu/pgroupW/images/a/a9/Serializarion_Framework.pdf

  Arrow vs Parquet: http://wesmckinney.com/blog/arrow-columnar-abadi/

  also here: http://dbmsmusings.blogspot.fr/2018/03/
  an-analysis-of-strengths-and-weaknesses.html