# CS-310 Scalable Software Architectures

# Lecture 15: Choosing a Database

Steve Tarzia

# Last Time: Distributed DB Consistency

- **Replication** of data ensures that a single failure does not lose data.
  - The more nodes you have, the more likely a failure!
- However, replication introduces **consistency** problems.
  - Tradeoff: must choose 2 of **C**onsistency, **A**vailability and **P**artition Tolerance.
- A distributed DB client, at very least, would want to achieve:
  - Montonic reads, monotonic writes, read your writes (*together:* **linearizability**).
- Ensure consistency by **waiting** for responses from multiple replicas.
- Different **quorum** levels (all, majority, one) trade delay of reads/writes and determine whether reads or writes are unavailable during recovery.
  - Cassandra DB lets programmer choose the quorum level for each read/write.
  - Other NoSQL databases are designed to use just one read/write strategy.

## Recall the goals of a Database:

- **Scalability** – work with data larger than computer's RAM.
- **Persistence** – keep data around after your program finishes.
- **Indexing** – efficiently sort & search along various dimensions.
- **Concurrency** – multiple users or applications can read/write.
- **Analysis** – SQL query language is concise yet powerful. Avoid transferring lots of data. Run analysis on the storage machines.
- **Separation of concerns** – decouples app code from storage engine. Also, allows apps to be stateless, allowing parallelism.

Less importantly:

- **Integrity** – restrict data type, disallow duplicate entries.
- **Deduplication** – save space, keep common data consistent.
- **Security** – different users can have access to specific data.

# Data storage options

| | Examples | Use cases |
|---|---|---|
| SQL Relational DB | MySQL, Oracle | Structured data. Transactional data. |
| Column-oriented DB | Snowflake, BigQuery | SQL queries for analytics on huge datasets (OLAP). |
| Search engine | Elastic search | Searchable text documents. |
| Document store | MongoDB | Semi-structured data (JSON docs). |
| Distributed cache | Redis | In-memory cache with expiration. Very fast. |
| NoSQL DB | Cassandra, Dynamo | Huge data to be accessed in parallel. |
| Cloud object store | S3, Azure Blobs | Images, videos, & other static content. |
| Cluster filesystem | Hadoop dist. fs. | Files to be processed in huge parallel computation. |
| Networked filesystem (NAS) | NFS, EFS, EBS | App is designed to write to local file system, but we want that storage to be scalable and shared. |

With so many options, choosing the "right" storage option is difficult!
Dozens of other DBs exist, but these examples are popular and representative examples.

## SQL Relational Databases

- The most common, traditional solution.
- Data is organized into **tables**, with **foreign keys** to cross reference.
  - The format of the data (schema) is predefined. Consistent, not flexible.
- SQL language run common data analyses *inside* the database:

  ```
  SELECT category, avg(price) FROM products GROUP BY category;
  ```
  - Running calculations on the storage machine helps performance.
    Data transfer (I/O) is a bottleneck in most systems.
  - Reduces data transfer between app and data store.
    Above query just returns a short answer over the network.
- Supports **transactions** (sequence of ops to be committed *all or none*).
- Works very well up to a certain size.
  - Writes must happen on **one "master" machine**.
  - **Read-replicas** give read scaling (w/delay). **Sharding** can help write scaling.

# Database transaction example: *spending gift card balance*

```
-- 1. start a new transaction
START TRANSACTION;

-- 2. get the gift card balance
SELECT @cardBalance:=balance FROM giftCards
  WHERE cardId=23902;

-- 3. insert a new order for customer 145
INSERT INTO orders(orderDate, status,
                   customerNumber)
  VALUES('2021-02-22', 'In Process', 145);

-- 4. get the newly-created order id
SELECT @orderId:=LAST_INSERT_ID();

-- 5. Insert order line items
INSERT INTO orderdetails(orderNumber, product,
                         quantity, priceEach)
  VALUES(@orderId,'S18_1749', 3, '136'),
        (@orderId,'S18_2248', 5, '55');

-- 6. deduct from balance
UPDATE giftCards SET balance=(@cardBalance-683)
  WHERE cardId=23902;

-- 7. end the transaction (commit changes)
COMMIT;
```

Why must these steps be completed *atomically* (together)?

- Prevent card balance from being spent twice.
- Prevent clients from seeing the order without line items.

The first is a race condition, the second is an inconsistency.

## Transactions on a distributed (NoSQL) DB?

- Transactions less common on NoSQL DBs because they are slow.
- Often, transactions are not necessary because a single key stores a lot of related data that can be modified at once.
- Transaction can be implemented by **locking** the keys involved:
  1. Lock the keys involved (the lock prevents reads/writes).
     - All replicas must agree to the lock.
     - Multiple competing lock requests may occur in parallel, but one must be chosen, so multiple rounds of communication may be needed to agree.
  2. Execute the transaction on all replicas. Wait for all to confirm.
  3. Unlock the keys involved (let reads/writes proceed).
- Reference: Google's Spanner OSDI 2012 paper

## How to implement a distributed lock?

- A lock requires an atomic conditional write operation, like:

```
PUT("key", "new_val") IF
GET("key") == "old_val";
```

- Many NoSQL databases support something like this (Cassandra, Mongo).

- Or if you don't care too much about scalability:
  - Store your transactional data in a SQL Database.
  - Or use a SQL Database to implement a lock used to control access in NoSQL.

NoSQL transaction to deduct $1 from an account:

```
id=37; // my unique client id
while(GET("lock") != id) {
   // get the lock if possible
   PUT("lock", id)
   IF GET("lock") == 0;
}
// do my 2-step transaction:
x = GET("balance");
PUT("balance", x - 1);

// release the lock
PUT("lock", 0);
```

# Throughput/scaling limitations

| Data store | Examples | Throughput limitations |
|---|---|---|
| SQL Relational DB | MySQL, Oracle | All writes to primary. Read-replication adds delay. |
| Column-oriented DB | Snowflake, BigQuery | |
| Search engine | Elastic search | |
| Document store | MongoDB | All use a scalable data partitioning method, such as hashing. |
| Distributed cache | Redis | |
| NoSQL DB | Cassandra, Dynamo | |
| Cloud object store | S3, Azure Blobs | |
| Cluster filesystem | Hadoop dist. fs. | |
| Networked filesystem | NFS, EFS, EBS | One machine, many disks (RAID). |

# Data abstractions

| Data store | Examples | Data abstraction |
|---|---|---|
| SQL Relational DB | MySQL, Oracle | Tables, rows, columns |
| Column-oriented DB | Snowflake, BigQuery | Tables, rows, columns |
| Search engine | Elastic search | JSON, text |
| Document store | MongoDB | Key → JSON |
| Distributed cache | Redis | Key → value (lists, sets, etc.) |
| NoSQL DB | Cassandra, Dynamo | 2D Key-value (pseudo-cols) |
| Cloud object store | S3, Azure Blobs | K-V / Filename-contents |
| Cluster filesystem | Hadoop dist. fs. | K-V / Filename-contents |
| Networked filesystem | NFS, EFS, EBS | Filename-contents |

Highly structured

Semi-structured

Files with data "blobs"

Files may have some internal structure, but the storage API is not aware of it and makes no use of it.

## Column-oriented Relational Databases

Previously, we saw that:

- Read-Replication and Sharding allow lots of **parallel** reads and writes.
- This is useful for **OLTP** applications (Online Transaction Processing).

**OLAP** (Online Analytics Processing) involves just a few huge queries

- Eg., Over the past three years, in which locations have customers been most responsive to our mailed-to-home coupons?
- Analytics queries involve **scanning** tables, not using indexes.
- Must be parallelized over many nodes.
- The workload is mostly **reads**, with occasional importing of new data.

Column-oriented DBs are optimized for SQL analytics workloads.

# Many choices for semi-structured, scalable stores!

| Data store | Examples | Data abstraction | |
|---|---|---|---|
| SQL Relational DB | MySQL, Oracle | Tables, rows, columns | |
| Column-oriented DB | Snowflake, BigQuery | Tables, rows, columns | |
| Search engine | Elastic search | JSON, text | Semi-structured |
| Document store | MongoDB | Key → JSON | Semi-structured |
| Distributed cache | Redis | Key → value (lists, sets, etc.) | Semi-structured |
| NoSQL DB | Cassandra, Dynamo | 2D Key-value (pseudo-cols) | Semi-structured |
| Cloud object store | S3, Azure Blobs | K-V / Filename-contents | |
| Cluster filesystem | Hadoop dist. fs. | K-V / Filename-contents | |
| Networked filesystem | NFS, EFS, EBS | Filename-contents | |

- Best choice depends on the structure of data being stored.

## Distributed data store comparison

Copied from: https://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis

## **MongoDB** stores JSON objects

- Below, "_id" is the sharding key used for data partitioning:

```
{
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Alan", last: "Turing" },
    birth: new Date('Jun 23, 1912'),
    death: new Date('Jun 07, 1954'),
    contribs: [ "Turing machine", "Turing test", "Turingery" ],
    views : NumberLong(1250000)
}
```

## MongoDB (3.2) – *a "document store"*

- **Main point:** JSON document store.

- **Best used:** If you like JSON. If documents change frequently, and you want to keep a history of changes.

- **For example:** For most things that you would do with MySQL or PostgreSQL, but having predefined columns really holds you back.

# MongoDB (3.2) – *a "document store"*

- Written in: C++
- **Main point: JSON document store**
- License: AGPL (Drivers: Apache)
- Protocol: Custom, binary (BSON)
- Master/slave replication (auto failover with replica sets)
- Sharding built-in
- Queries are javascript expressions
- Run arbitrary javascript functions server-side
- Geospatial queries
- Multiple storage engines with different performance characteristics
- Performance over features
- Document validation
- Journaling (efficiently keeping previous versions of documents)
- Powerful aggregation framework
- Text search integrated
- GridFS to store big data + metadata (not actually a FS)
- Has geospatial indexing
- Data center aware
- **Best used:** If you need dynamic queries. If you prefer to define indexes, not map/reduce functions. If you need good performance on a big DB. If documents change frequently, and you want to keep a history of changes.
- **For example:** For most things that you would do with MySQL or PostgreSQL, but having predefined columns really holds you back.

## ElasticSearch also stores JSON documents

- But it's designed to **index every word** in the document and to handle advanced queries:

```
{
    "date": "2020-04-15",
    "txt": "$1,000 in donations buy lunch for ambulance, victim assistance workers ... WATERTOWN,
N.Y. (WWNY) - Two anonymous donations at a downtown Watertown restaurant are buying first responders
lunch. Vito's Gourmet received the donations totaling $1,000 from people who wanted to give back to
the community. On Wednesday, owner Todd Tarzia delivered gift certificates to Guilfoyle Ambulance
and the Victims Assistance Center for each of its employees. \u201cOne of the donors in particular
specifically designated first responders as who they wanted the lunch to go to and we thought, well,
geez, first responders don\u2019t all sit around a lunch table, especially in the world with the
virus right now so what can we do to get lunch to them on their schedule. So we decided to make up
gift certificates for amounts that\u2019s enough for everyone to have lunch,\u201d said Tarzia. \"To
get a gift like this is just so thoughtful and our people are going to be very thankful for this,\"
said Bruce Wright, CEO, Guilfoyle Ambulance. …",
    "title": "$1,000 in donations buy lunch for ambulance, victim assistance workers",
    "lang": "en",
    "url": "https://www.wwnytv.com/2020/04/15/donations-buy-lunch-ambulance-victim-assistance-
workers/"
}
```

- How could you use a simple Distributed Hash Table (eg., Redis or Cassandra) to implement an index of all the words in this document? User wants to search for "Watertown AND gift".
- An inverted index. For each word in document, store this document id under the word key.

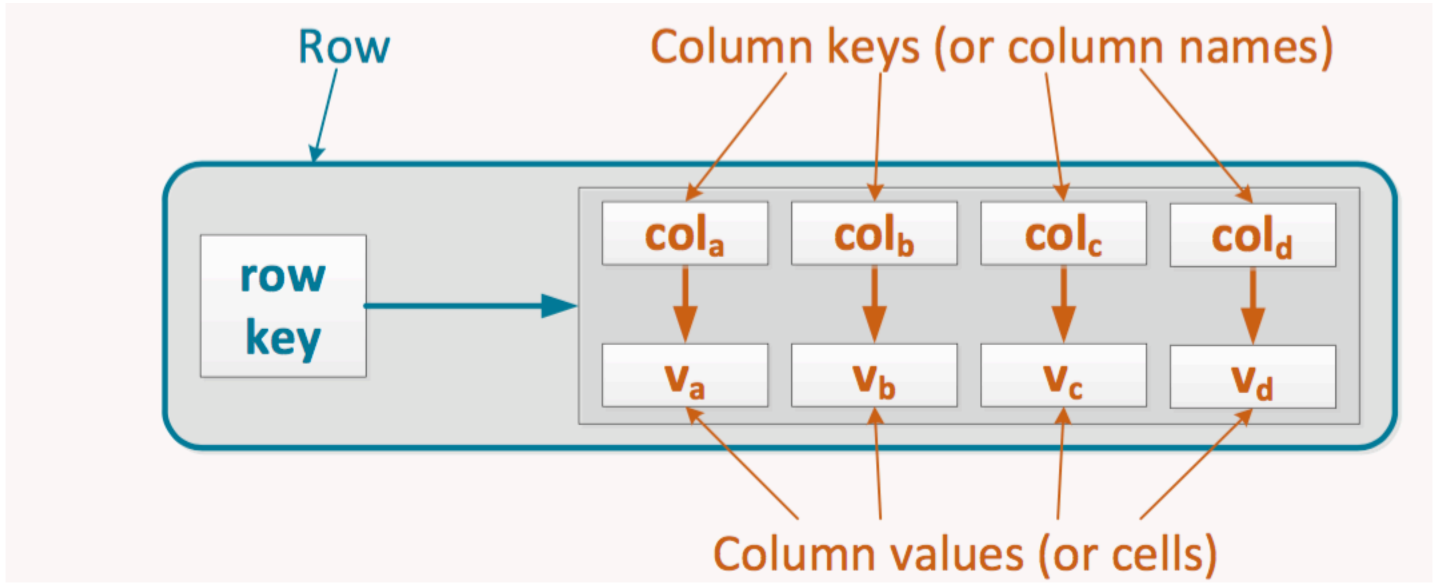
## ElasticSearch – *a "search engine"*

- **Main point:** Advanced Search
- **Best used:** When you have objects with (flexible) fields (or plain text), and you need search by all words in the document, or you need to construct complex search queries (AND, OR, NOT, …)
- **For example:** Full-text document search, a leaderboard system that depends on many variables.

# ElasticSearch (0.20.1) – *a "search engine"*

- Written in: Java
- **Main point: Advanced Search**
- License: Apache
- Protocol: JSON over HTTP (Plugins: Thrift, memcached)
- Stores JSON documents
- Has versioning
- Parent and children documents
- Documents can time out
- Very versatile and sophisticated querying, scriptable
- Write consistency: one, quorum or all
- Sorting by score (!)
- Geo distance sorting
- Fuzzy searches (approximate date, etc) (!)
- Asynchronous replication
- Atomic, scripted updates (good for counters, etc)
- Can maintain automatic "stats groups" (good for debugging)
- **Best used:** When you have objects with (flexible) fields, and you need "advanced search" functionality.
- **For example:** A dating service that handles age difference, geographic location, tastes and dislikes, etc. Or a leaderboard system that depends on many variables.

# **Cassandra** rows (NoSQL, 2d key-value store)



Each row's value is a map of "**columns**" to value. Column names are indexed within the row.

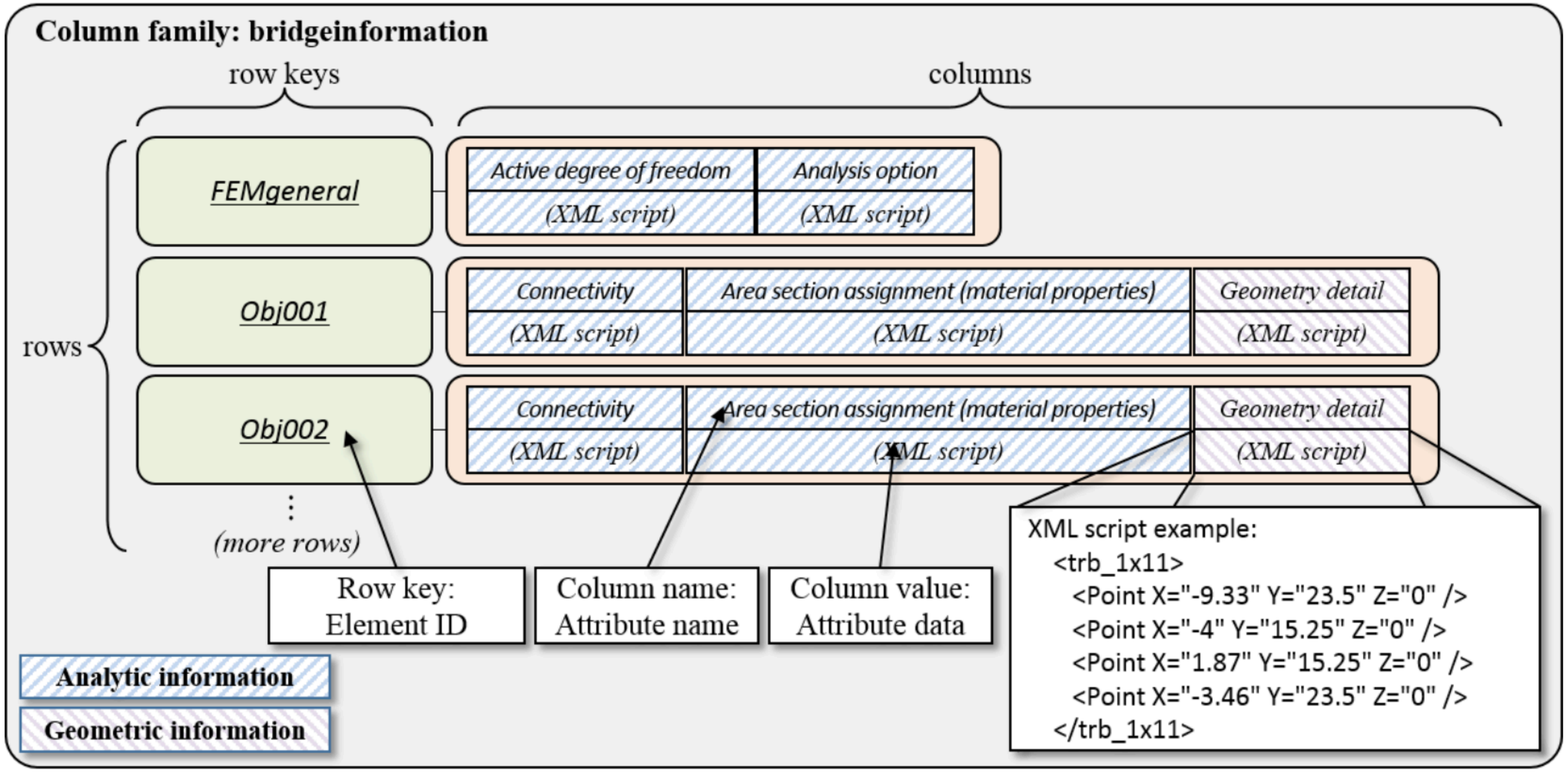
Row key is the hashing key that determines on which nodes the row is stored.

| | | | |
|---|---|---|---|
| **7b976c48...** | name: Bill Watterson | state: DC | birth_date: 1953 |
| **7c8f33e2...** | name: Howard Tayler | state: UT | birth_date: 1968 |
| **7d2a3630...** | name: Randall Monroe | state: PA | |
| **7da30d76...** | name: Dave Kellett | state: CA | |

*Columns are defined separately for each row!*

Details: https://tech.ebayinc.com/engineering/cassandra-data-modeling-best-practices-part-1/

## Cassandra bridge information example


Column family: bridgeinformation
row keys
columns
rows
FEMgeneral
Active degree of freedom (XML script)
Analysis option (XML script)
Obj001
Connectivity (XML script)
Area section assignment (material properties) (XML script)
Geometry detail (XML script)
Obj002
Connectivity (XML script)
Area section assignment (material properties) (XML script)
Geometry detail (XML script)
⋮
(more rows)
Row key: Element ID
Column name: Attribute name
Column value: Attribute data
Analytic information
Geometric information

XML script example:

```
<trb_1x11>
  <Point X="-9.33" Y="23.5" Z="0" />
  <Point X="-4" Y="15.25" Z="0" />
  <Point X="1.87" Y="15.25" Z="0" />
  <Point X="-3.46" Y="23.5" Z="0" />
</trb_1x11>
```

Fig. 6 Data schema of bridge information model on Apache Cassandra

https://www.researchgate.net/publication/301630614_A_NoSQL_data_management_infrastructure_for_bridge_monitoring

## Cassandra – *a "NoSQL database"*

- **Main point:** Store huge datasets.
- **Best used:** When you need to store data so huge that it doesn't fit on server, but still want a friendly familiar interface to it.
- **For example:** Web analytics, to count hits by hour, by browser, by IP, etc. Transaction logging. Data collection from huge sensor arrays.

## Cassandra (2.0) – *a "NoSQL database"*

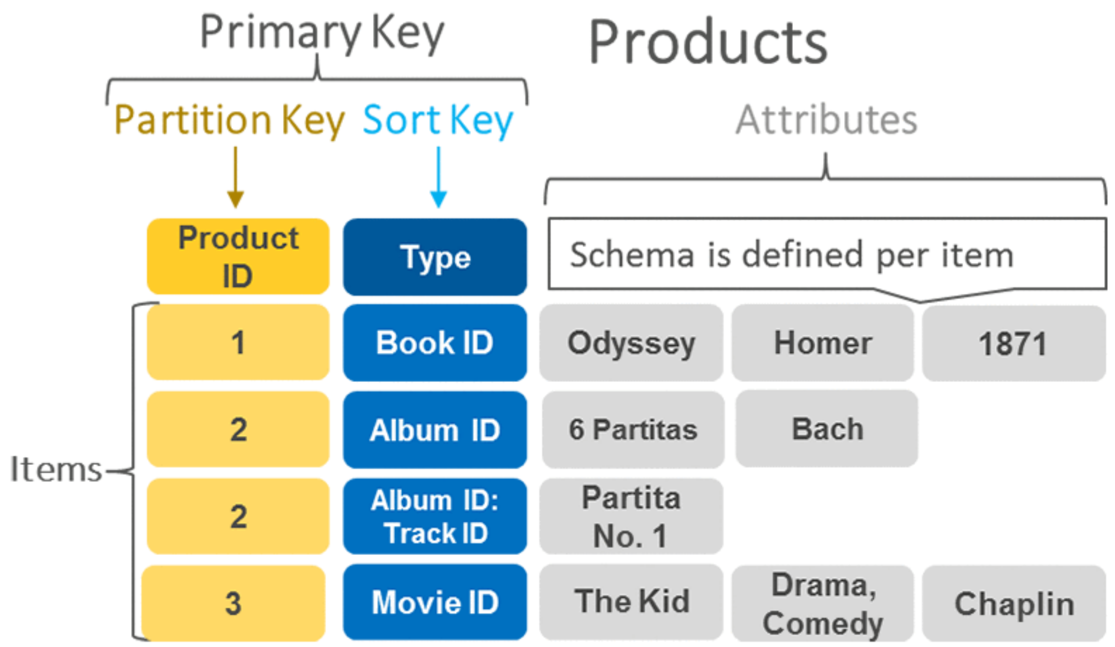
- Written in: Java
- **Main point: Store huge datasets in "almost" SQL**
- License: Apache
- Protocol: CQL3 & Thrift
- CQL3 is very similar to SQL, but with some limitations that come from the scalability (most notably: no JOINs, no aggregate functions.)
- Querying by key, or key range (secondary indices are also available)
- Tunable trade-offs for distribution and replication (N, R, W)
- Data can have expiration (set on INSERT).
- Writes can be much faster than reads (when reads are disk-bound)
- Map/reduce possible with Apache Hadoop
- All nodes are similar, as opposed to Hadoop/HBase
- Very good and reliable cross-datacenter replication
- Distributed counter datatype.
- You can write triggers in Java.

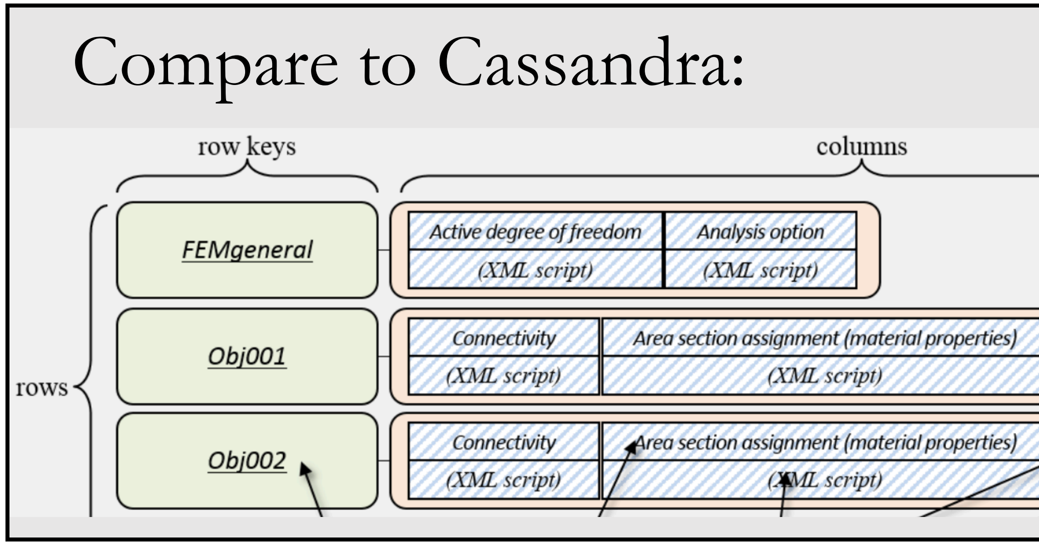
- **Best used:** When you need to store data so huge that it doesn't fit on one server, but still want a friendly familiar interface to it.

- **For example:** Web analytics, to count hits by hour, by browser, by IP, etc. Transaction logging. Data collection from huge sensor arrays.

# DynamoDB is a 2D key-value store, like Cassandra

- **Partition Key** (like Cassandra's row key) is hashed to find partition.
- **Sort Key** (optional) allows efficient range queries within the partition key.
  - Together, the Partition and Sort keys form the **Primary Key**.
- **Attributes** are key-value pairs stored under the Primary Key.

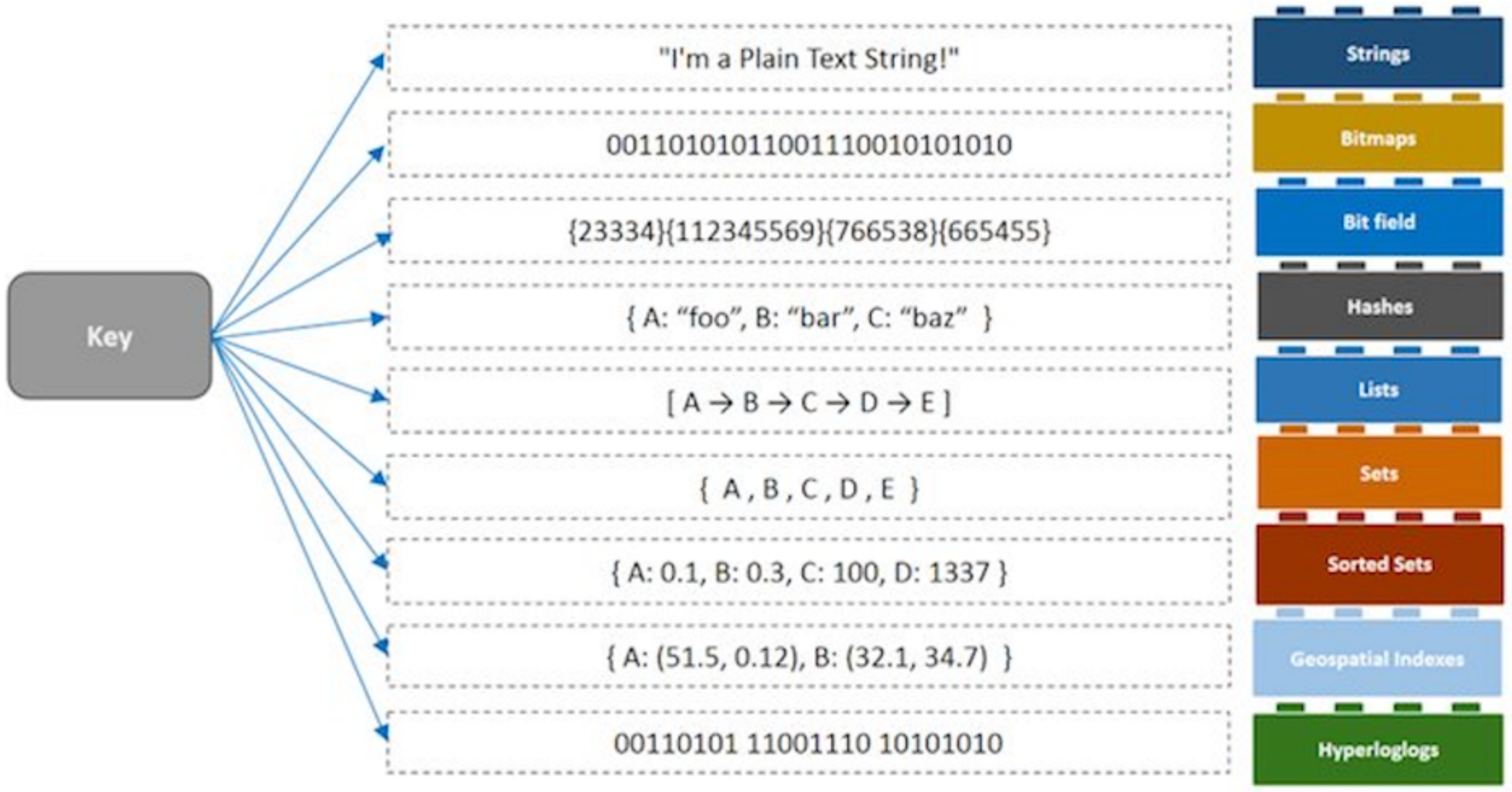



Compare to Cassandra:
row keys
columns
rows
FEMgeneral
Active degree of freedom (XML script)
Analysis option (XML script)
Obj001
Connectivity (XML script)
Area section assignment (material properties) (XML script)
Obj002
Connectivity (XML script)
Area section assignment (material properties) (XML script)

Reference: https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/

## **Redis** DB/cache values

- All data is stored in RAM (not just disk), for high performance.
- Redis understands many types of data values:
    - allows operations like "add to a set" that modify (or get) part of a value.



## Distributed Caches

- For example: Redis, Memcached, ElastiCache, Riak

- Originally developed in order to reduce load on relational databases.
  - Cache responses to frequent DB requests or other materialized application data.
  - Always support timed **expiration** of data.
- Use the same basic key-value abstraction as NoSQL distributed DBs.
- Store data across many nodes.
- Have the same data consistency issues as NoSQL databases.
- Often optimized to do everything in-memory,
  - but most also store data persistently to disk.

*So…* distributed caches and NoSQL databases are very similar.

## Comparison

### NoSQL Database

- Items are **permanent**/persistent.
- All items are stored on **disk** (some are cached in RAM).
- **Scale** is the primary goal.

### Distributed Cache

- Items **expire**.
- Items are stored in RAM (though maybe persisted to disk).
- **Speed** is the primary goal.
- RAM capacity is limited.
  - Once capacity is reached, start evicting oldest/least-used items.

## Comparison

**CDN / Reverse Proxy Cache**

- Cache common HTTP responses.
- Transparent to the application.
- Just configure the cache's origin

**Distributed Cache**

- Cache common's used data that contributes to responses.
- For example:
  - A leaderboard in header of ever HTML page.
  - Session information for the user.

## Redis (V3.2) – *a "cache"*

- **Main point:** Blazing fast storage.
- **Best used:** For rapidly changing data with a foreseeable database size (should fit mostly in memory). Also, for caching data than can be rebuilt from another data store.
- **For example:** To store real-time stock prices. Real-time analytics. Leaderboards. Real-time communication. And wherever you used memcached before.

## Redis (V3.2) – *a "cache"*

- Written in: C
- **Main point: Blazing fast**
- License: BSD
- Protocol: Telnet-like, binary safe
- Disk-backed in-memory database,
- Master-slave replication, automatic failover
- Simple values or data structures by keys
- but complex operations like ZREVRANGEBYSCORE.
- INCR & co (good for rate limiting or statistics)
- Bit and bitfield operations (eg. to implement bloom filters)
- Has sets (also union/diff/inter)
- Has lists (also a queue; blocking pop)
- Has hashes (objects of multiple fields)
- Sorted sets (high score table, good for range queries)
- Lua scripting capabilities
- Has transactions
- Values can be set to expire (as in a cache)
- Pub/Sub lets you implement messaging
- GEO API to query by radius (!)

- **Best used:** For rapidly changing data with a foreseeable database size (should fit mostly in memory).

- **For example:** To store real-time stock prices. Real-time analytics. Leaderboards. Real-time communication. And wherever you used memcached before.

## Filesystem choices

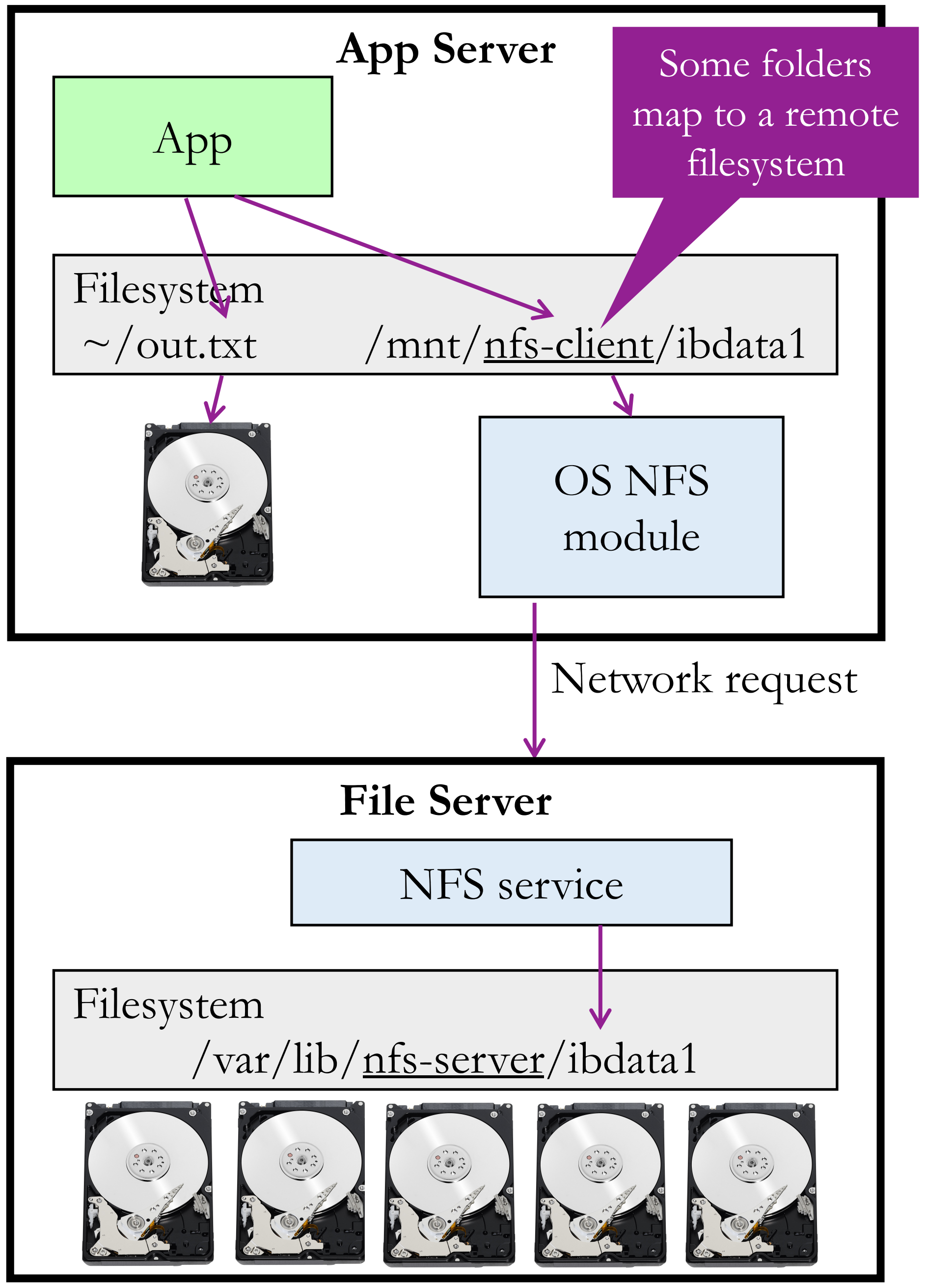
| Data store | Examples | Data abstraction |
|---|---|---|
| SQL Relational DB | MySQL, Oracle | Tables, rows, columns |
| Column-oriented DB | Snowflake, BigQuery | Tables, rows, columns |
| Search engine | Elastic search | JSON, text |
| Document store | MongoDB | Key → JSON |
| Distributed cache | Redis | Key → value (lists, sets, etc.) |
| NoSQL DB | Cassandra, Dynamo | 2D Key-value (pseudo-cols) |
| Cloud object store | S3, Azure Blobs | K-V / Filename-contents |
| Cluster filesystem | Hadoop dist. fs. | K-V / Filename-contents |
| Networked filesystem | NFS, EFS, EBS | Filename-contents |

$\left.\vphantom{\begin{matrix}a\\b\\c\end{matrix}}\right\}$ Files with arbitrary data

# Networked file system

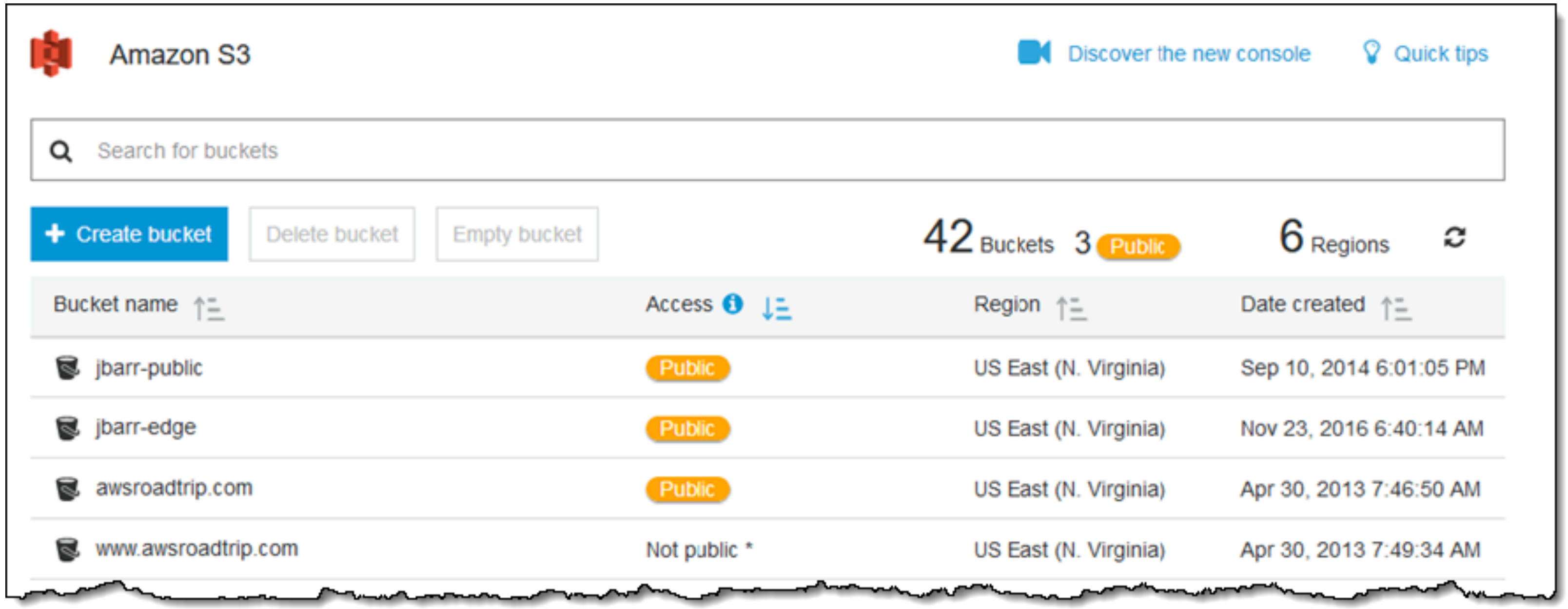
- Eg., NFS (unix), SMB (Windows).
- Managed by the OS.
- Provides a regular filesystem interface to applications by *mounting* the remote drive.

- Not too useful in modern applications, but may be necessary if your app is built to work directly with a local file system.
- Modern apps should instead interact with cloud-based storage services.



## Cloud object store (S3)

- A flexible general-purpose file store for cloud apps.
- Managed by cloud provider. Capacity available is "unlimited."
- Provides a network API for accessing files (maybe REST).
- In other words, app accesses files like a remote database.
- Often provides a public HTTP GET interface to access files:
  - Can be easily connected to CDN or urls used directly

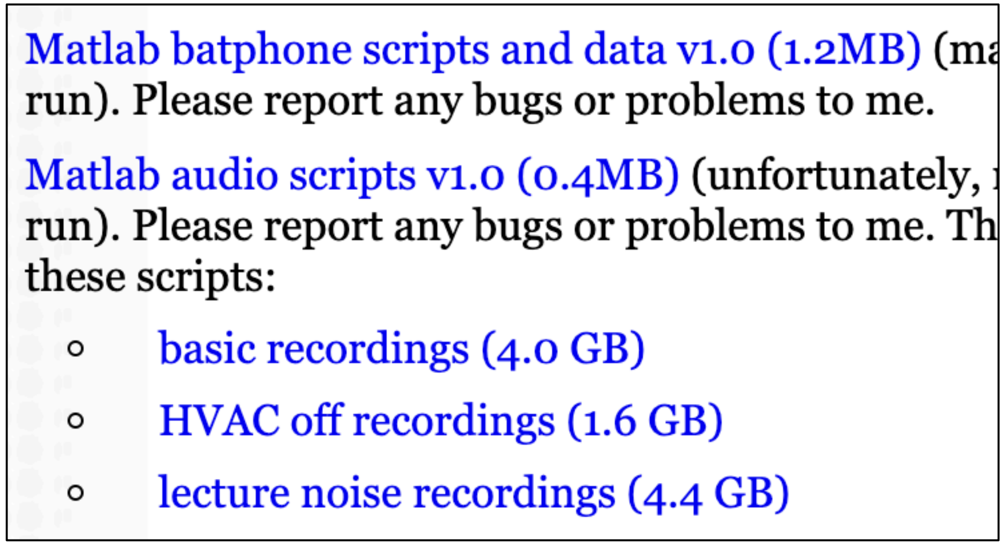


| Bucket name | Access | Region | Date created |
|---|---|---|---|
| jbarr-public | Public | US East (N. Virginia) | Sep 10, 2014 6:01:05 PM |
| jbarr-edge | Public | US East (N. Virginia) | Nov 23, 2016 6:40:14 AM |
| awsroadtrip.com | Public | US East (N. Virginia) | Apr 30, 2013 7:46:50 AM |
| www.awsroadtrip.com | Not public * | US East (N. Virginia) | Apr 30, 2013 7:49:34 AM |

# S3 example for hosting media files on web

- [https://stevetarzia.com/localization](https://stevetarzia.com/localization)

Browser view:



Matlab batphone scripts and data v1.0 (1.2MB) (ma
run). Please report any bugs or problems to me.

Matlab audio scripts v1.0 (0.4MB) (unfortunately,
run). Please report any bugs or problems to me. Th
these scripts:

- basic recordings (4.0 GB)
- HVAC off recordings (1.6 GB)
- lecture noise recordings (4.4 GB)

HTML:

```
<li><p><a href="mobisys11_batphone_v1.0.tar.gz">Matlab
batphone scripts and data v1.0 (1.2MB)</a> (may require some
toolkits to run).  Please report any bugs or problems to
me.</li>

<li><p><a href="mobisys11_scripts_v1.0.tar.gz">Matlab audio
scripts v1.0 (0.4MB)</a> (unfortunately, requires several
toolkits to run).  Please report any bugs or problems to me.
The following data is needed for these scripts:

<ul><li><p><a href="https://s3-us-west-
2.amazonaws.com/starzia/www/mobisys11_recordings_passive.tar.
gz">basic recordings (4.0 GB)</a>

<li><p><a href="https://s3-us-west-
2.amazonaws.com/starzia/www/mobisys11_recordings_HVAC_off.tar
.gz">HVAC off recordings (1.6 GB)</a>

<li><p><a href="https://s3-us-west-
2.amazonaws.com/starzia/www/mobisys11_recordings_lectures.tar
.gz">lecture noise recordings (4.4 GB)</a></ul>
```

## Hadoop File System (HDFS)

- When you need to use Hadoop/Spark to do distributed processing.
- Data is too big to move it for analysis.
- Allows data to reside on the same machines where computation happens, thus making processing efficient.

- Hadoop distributed filesystem and its distributed processing tools were designed to work together.

## Recap – Choosing a data store

| Data store | Examples | Data abstraction | |
|---|---|---|---|
| SQL Relational DB | MySQL, Oracle | Tables, rows, columns | Highly structured |
| Column-oriented DB | Snowflake, BigQuery | Tables, rows, columns | |
| Search engine | Elastic search | JSON, text | Semi-structured |
| Document store | MongoDB | Key → JSON | |
| Distributed cache | Redis | Key → value (lists, sets, etc.) | |
| NoSQL DB | Cassandra, Dynamo | 2D Key-value (pseudo-cols) | |
| Cloud object store | S3, Azure Blobs | K-V / Filename-contents | Files with data "blobs" |
| Cluster filesystem | Hadoop dist. fs. | K-V / Filename-contents | |
| Networked filesystem | NFS, EFS, EBS | Filename-contents | |

Your choice depends mainly on the **structure** of data and pattern of **access**.