Ghislain Fourny

Big Data

3. Object storage
Where are we?

Last lecture:
Reminder on relational databases
Where are we?

Relational databases fit on a single machine
Where are we?

Petabytes do not fit on a single machine
The lecture journey

Monolithic Relational Database

Modular "Big Data" Technology Stack
Not reinventing the wheel

99% of what we learned with 46 years of SQL and relational can be reused
Important take-away points

**Relational algebra:**
- Selection
- Projection
- Grouping
- Sorting
- Joining
Important take-away points

Language
- SQL
- Declarative languages
- Functional languages
- Optimizations
- Query plans
Important take-away points

What a table is made of

✅ Table
✅ Columns
✅ Primary key
✅ Row
Important take-away points

Denormalization

- 1NF vs. nesting
- 2NF/3NF vs. pre-join
Important take-away points

Transactions

- ✔ Atomicity
- ✔ Consistency
- ✗ Isolation
- ✗ Durability
- NEW Atomic Consistency
- NEW Availability
- NEW Partition tolerance
The stack

User interfaces
Querying
Data stores
Indexing
Processing
Validation
Data models
Syntax
Encoding
Storage
The stack:
Storage

Local filesystem
NFS
GFS
HDFS
S3
Azure Blob Storage
The stack:
Encoding

ASCII
ISO-8859-1
UTF-8
BSON
The stack:
Syntax

Text
CSV
XML
JSON
RDF/XML
Turtle
XBRL
The stack:
Data models

Tables: Relational model
Trees: XML Infoset, XDM
Graphs: RDF
Cubes: OLAP
The stack:
Validation

XML Schema
JSON Schema
Relational schemas
XBRL taxonomies
The stack:
Processing

Two-phase processing:
MapReduce

DAG-driven processing:
Tez, Spark

Elastic computing:
EC2
The stack: Indexing

Key-value stores
Hash indices
B-Trees
Geographical indices
Spatial indices
The stack:
Data stores

RDBMS
(Oracle/IBM/Microsoft)
MongoDB
CouchBase
ElasticSearch
Hive
HBase
MarkLogic
Cassandra
...

Data stores
The stack: Querying

SQL
XQuery
MDX
SPARQL
REST APIs
The stack: User interfaces (UI)

Excel
Access
Tableau
Qlikview
BI tools
Storage: from a single machine to a cluster
Storage needs to be stored somewhere.
Let's start from the 70s...
File storage

Lorem Ipsum
Dolor sit amet
Consectetur
Adipiscing
Elit. In
Imperdiet
Ipsum ante

Files organized in a hierarchy
What is a file made of?

Content + Metadata = File
$ ls -l
total 48
drwxr-xr-x  5 gfourny staff  170 Jul 29  08:11 2009
 drwxr-xr-x 16 gfourny staff  544 Aug 19 14:02 Exercises
 drwxr-xr-x 11 gfourny staff  374 Aug 19 14:02 Learning Objectives
 drwxr-xr-x 18 gfourny staff  612 Aug 19 14:52 Lectures
-rw-r--r--  1 gfourny staff 1788 Aug 19 14:04 README.md

Fixed "schema"
File Content: Block storage

Files content stored in blocks
Local storage

- Local Machine
- LAN (NAS)
- WAN

LAN = local-area network
NAS = network-attached storage
WAN = wide-area network
Scaling Issues

1,000 files ✔
1,000,000 files ✔
1,000,000,000 files ✗
Better performance: Explicit Block Storage

Application

(Control over locality of blocks)
So how do we make this scale?

1. We throw away the hierarchy!
So how do we make this scale?

2. We make metadata flexible
So how do we make this scale?

3. We make the data model trivial
So how do we make this scale?

4. We use commodity hardware
... and we get Object Storage

"Black-box" objects

Flat and global key-value model

Flexible metadata

Commodity hardware
Scale
One machine's not good enough. How do we scale?
Approach 1: scaling up
Approach 1: scaling up
Approach 2: scaling out
Approach 2: scaling **out**
Approach 2: scaling **out**
Approach 2: scaling out
Hardware price comparison

Scale up

Scale out
Approach 3: be smart
Approach 3: be smart

“You can have a second computer once you’ve shown you know how to use the first one.”

Paul Barham
In this lecture

Approach 2

Scale out
Data centers
Numbers - computing

1,000-100,000 machines in a data center

1-100 cores per server
Numbers - storage

1-10 TB
local storage
per server

10-1000 GB
of RAM per server
Numbers - network

1 GB/s network bandwidth for a server
Racks

Height in "rack units" (e.g., 42 RU)
Racks

Modular:
- servers
- storage
- routers
- ...
Rack servers

Lenovo ThinkServer RD430 Rack Server

1-4 RU
Amazon S3
S3 Model
S3 Model
S3 Model

Bucket ID
S3 Model

Bucket ID
S3 Model

Bucket ID + Object ID
Scalability

Max. 5 TB
Scalability

100/account

(more upon request)
Durability

99.9999999999%

Loss of 1 in $10^{11}$ objects
Availability

99.99%

Down 1h / year
API

REST

Java

Python

PHP
REST API

GET
PUT
DELETE
POST
HEAD
OPTIONS
TRACE
CONNECT

Method + Resource (URI)
PUT

(Idempotent)
GET

(Side-effect free)
POST

Most generic: side effects
S3 Resources: Buckets

http://bucket.s3.amazonaws.com

http://bucket.s3-region.amazonaws.com
S3 Resources: Objects

http://bucket.s3.amazonaws.com/object-name

http://bucket.s3-region.amazonaws.com/object-name
S3 REST API

PUT Bucket
DELETE Bucket
GET Bucket

PUT Object
DELETE Object
GET Object
Example

GET /my-image.jpg HTTP/1.1
Host: bucket.s3.amazonaws.com
Date: Wed, 27 Sep 2016 09:30:00 GMT
Authorization: authorization string
Folders: is S3 a file system?

Logical
(Browsing)

Physical
(Object keys)

/food/fruits/orange
/food/fruits/strawberry
/food/vegetables/tomato
/food/vegetables/turnip
/food/vegetables/lettuce
Static website hosting

http://jsoniq.org.s3-website-us-east-1.amazonaws.com/
Replication

Fault tolerance
Faults

Local (node failure) versus Regional (natural catastrophe)
Regions
Regions

1. Optimize latency
2. Resilient to natural catastrophes
## Storage Class

<table>
<thead>
<tr>
<th>Storage Class</th>
<th>Availability/Cost Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard</strong></td>
<td>High availability</td>
</tr>
<tr>
<td><strong>Standard – Infrequent Access</strong></td>
<td>Less availability&lt;br&gt;Cheaper storage&lt;br&gt;Cost for retrieving</td>
</tr>
<tr>
<td><strong>Amazon Glacier</strong></td>
<td>Low-cost&lt;br&gt;Hours to GET</td>
</tr>
</tbody>
</table>
# Overall comparison Azure vs. S3

<table>
<thead>
<tr>
<th></th>
<th>S3</th>
<th>Azure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object ID</strong></td>
<td>Bucket + Object</td>
<td>Account + Partition + Object</td>
</tr>
<tr>
<td><strong>Object API</strong></td>
<td>Blackbox</td>
<td>Blocks or pages</td>
</tr>
<tr>
<td><strong>Limit</strong></td>
<td>5 TB</td>
<td>195 GB (blocks) 1 TB (pages)</td>
</tr>
</tbody>
</table>
Azure Architecture: Storage Stamp

Account name

Partition name

Object name

Virtual IP address

Front-Ends

Partition Layer

Stream Layer
Azure Architecture: One storage stamp

10-20 racks * 18 storage nodes/rack (30PB)
Azure Architecture: Keep some buffer kept below 70/80% storage capacity
Storage Replication

Intra-stamp replication (synchronous)
Storage Replication

Inter-stamp replication (asynchronous)
Location Services

Account name mapped to one Virtual IP (primary stamp)

DNS ➔ Location Services ➔ Virtual IP (primary)

Virtual IP ➔ Front-Ends ➔ Partition Layer ➔ Stream Layer

Front-Ends ➔ Partition Layer ➔ Stream Layer

Virtual IP ➔ Front-Ends ➔ Partition Layer ➔ Stream Layer
Location Services

North America
Europe
Asia
Location Services

Account name
Primary stamp’s VIP
Partition + Object

DNS

Front-Ends
Partition Layer
Stream Layer

Front-Ends
Partition Layer
Stream Layer
Key-value storage (sneak peek)
Can we consider object storage a database?
Issue: latency

~100-300ms

S3

VS.

1-9 ms

Typical Database
Key-value stores

1. Similar data model to object storage
Key-value stores

2. Smaller objects

(DynamoDB)
Key-value stores

3. No Metadata
Key-value stores: data model
**Key-value stores: data shape**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Key" /></td>
<td><img src="image2" alt="Value" /></td>
</tr>
<tr>
<td><img src="image3" alt="Key" /></td>
<td><img src="image4" alt="Value" /></td>
</tr>
<tr>
<td><img src="image5" alt="Key" /></td>
<td><img src="image6" alt="Value" /></td>
</tr>
<tr>
<td><img src="image7" alt="Key" /></td>
<td><img src="image8" alt="Value" /></td>
</tr>
<tr>
<td><img src="image9" alt="Key" /></td>
<td><img src="image10" alt="Value" /></td>
</tr>
</tbody>
</table>

Lots of rows
Which is the most efficient data structure for querying this?

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key 1</td>
<td></td>
</tr>
<tr>
<td>Key 2</td>
<td></td>
</tr>
<tr>
<td>Key 3</td>
<td></td>
</tr>
<tr>
<td>Key 4</td>
<td></td>
</tr>
<tr>
<td>Key 5</td>
<td></td>
</tr>
<tr>
<td>Key 6</td>
<td></td>
</tr>
</tbody>
</table>
Distributed Hash Tables: Chord

hashed

n-bit ID
(DynamoDB: 128 bits)
Each Node picks a 128-bit hash
Nodes are organized in a ring

\[ \text{mod } 2^n \]
ID stored at next node "up"

111... 000...

mod $2^n$
Take away messages: how to scale out?

- Simplify the model!
- Buy cheap hardware!
- Remove schemas!