

# Structuring data for efficient I/O

**Sébastien Ponce**

sebastien.ponce@cern.ch

**CERN**

Thematic CERN School of Computing 2015

# Overall Course Structure

## Structuring Data for efficient I/O

- Data formats, data compression
- Data addressing

## Many ways to Store Data

- Storage devices and their specificities
- Distributing and parallelizing storage

## Preserving data

- Data consistency
- Data safety

## Key ingredients to achieve efficient I/O

- Synchronous vs asynchronous I/O
- I/O optimizations and caching

# Outline

- 1 Data format
  - Row vs Column
- 2 Compressing data
  - Compression algorithms
  - Efficiency and use cases
- 3 Data addressing
  - Hierarchical namespaces
  - Limitations
  - Flat namespaces
- 4 Stateful interfaces
  - POSIX
  - Limitations
  - Stateless interfaces
- 5 Conclusion

# Data format

- 1 Data format
  - Row vs Column
- 2 Compressing data
- 3 Data addressing
- 4 Stateful interfaces
- 5 Conclusion

# Data structure by example - scenario

## Scenario

- You are measuring temperatures within a piece of detector
- You have 10K captors and you take one measure every minute
- After a month, you got 432M measures
- That is 1.6GB if you take single precision floats (32bits)

# Data structure by example - row storage

## Naive structure

- You arrange your captors in a sequential order according to the detector geometry
- Each minute, you create a new “row” of data, with 10K floats representing temperatures given by the captors, in that order

# Data structure by example - row storage

## Naive structure

- You arrange your captors in a sequential order according to the detector geometry
- Each minute, you create a new “row” of data, with 10K floats representing temperatures given by the captors, in that order

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$

# Data structure by example - row storage

## Naive structure

- You arrange your captors in a sequential order according to the detector geometry
- Each minute, you create a new “row” of data, with 10K floats representing temperatures given by the captors, in that order

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
1	$a_1$	$b_1$	...	$z_1$



# Data structure by example - row storage

## Naive structure

- You arrange your captors in a sequential order according to the detector geometry
- Each minute, you create a new “row” of data, with 10K floats representing temperatures given by the captors, in that order

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
1	$a_1$	$b_1$	...	$z_1$
...	...	...	...	...
n	$a_n$	$b_n$	...	$z_n$

# Data structure by example - row storage

## Naive structure

- You arrange your captors in a sequential order according to the detector geometry
- Each minute, you create a new “row” of data, with 10K floats representing temperatures given by the captors, in that order

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
1	$a_1$	$b_1$	...	$z_1$
...	...	...	...	...
n	$a_n$	$b_n$	...	$z_n$

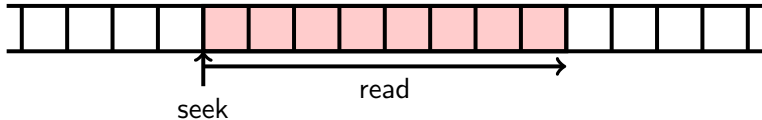
## File content

$a_0$   $b_0$  ...  $z_0$   $a_1$   $b_1$  ...  $z_1$  ...  $a_n$   $b_n$  ...  $z_n$

# Data structure by example - access

Find out overheated devices at a given time

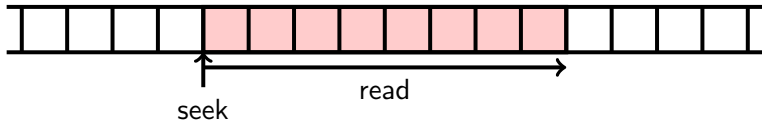
- find the offset of that time in the file
- read 10K numbers
- apply simple filter



# Data structure by example - access

Find out overheated devices at a given time

- find the offset of that time in the file
- read 10K numbers
- apply simple filter



## Cost

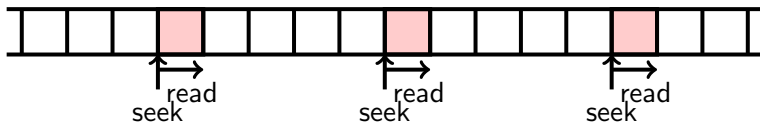
- one seek
- one read of 10K ints

This is efficient !

# Data structure by example - access (2)

Graph the temperature evolution of a given device

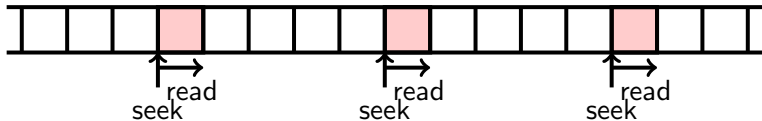
- read 43.2K numbers from the file, every 40K bytes
- graph them



# Data structure by example - access (2)

## Graph the temperature evolution of a given device

- read 43.2K numbers from the file, every 40K bytes
- graph them



## Cost

- 43.2K reads of 4 bytes and 43.2K seeks !
- on top typical block size in a filesystem is 8k
- you will probably read effectively 20% of the file !
- actually reading the whole file will be more efficient

Here the structure of our data is a killer

# Column storage

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
1	$a_1$	$b_1$	...	$z_1$
...	...	...	...	...
n	$a_n$	$b_n$	...	$z_n$

# Column storage

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
1	$a_1$	$b_1$	...	$z_1$
...	...	...	...	...
n	$a_n$	$b_n$	...	$z_n$

## File content

$a_0 a_1 \dots a_n b_0 b_1 \dots b_n \dots z_0 z_1 \dots z_n$



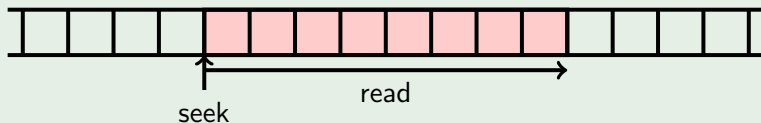
# Column storage

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
1	$a_1$	$b_1$	...	$z_1$
...	...	...	...	...
n	$a_n$	$b_n$	...	$z_n$

## File content

$a_0 a_1 \dots a_n b_0 b_1 \dots b_n \dots z_0 z_1 \dots z_n$

## Back to efficient read



# Row vs column storage

## Definition

**Row storage** respects internal structure of the data and puts the different items one next in a sequence

**Column storage** breaks the internal structure of the data to collate similar pieces

# Row vs column storage

## Definition

**Row storage** respects internal structure of the data and puts the different items one next in a sequence

**Column storage** breaks the internal structure of the data to collate similar pieces

## Why to use column ?

- to optimize I/O in general and avoid scattered reads
- to optimize data compression
- to optimize parallelization of processing

# Row vs column storage

## Definition

**Row storage** respects internal structure of the data and puts the different items one next in a sequence

**Column storage** breaks the internal structure of the data to collate similar pieces

## Why to use column ?

- to optimize I/O in general and avoid scattered reads
- to optimize data compression
- to optimize parallelization of processing

## Drawback of column storage

- a column organized file cannot be updated easily
- column storage is usually created from row storage in a postprocessing phase.

# The parallelization view

## Row/Column

- naming
  - Row storage
  - Column storage
- goal
  - Storage efficiency, data on disk

## AoS/SoA

- naming
  - Array of structs (AoS)
  - Struct of arrays (SoA)
- goal
  - Algorithmic efficiency, data in RAM

## A lot in common

- Same advantages/disadvantages
- Same compromises
- Same ultimate solution ? : column per block / AoSoA

# Column per block storage

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
...	...	...	...	...
$p-1$	$a_{p-1}$	$b_{p-1}$	...	$z_{p-1}$
$p$	$a_p$	$b_p$	...	$z_p$
...	...	...	...	...

## File content

$a_0 a_1 \dots a_{p-1} b_0 b_1 \dots b_{p-1} \dots z_0 z_1 \dots z_{p-1} a_p a_{p+1} \dots a_{2p-1} b_p b_{p+1} \dots b_{2p-1} \dots$

# Column per block storage

Time (mn)	Captor 1	Captor 2	...	Captor c
0	$a_0$	$b_0$	...	$z_0$
...	...	...	...	...
$p-1$	$a_{p-1}$	$b_{p-1}$	...	$z_{p-1}$
$p$	$a_p$	$b_p$	...	$z_p$
...	...	...	...	...

## File content

$a_0 a_1 \dots a_{p-1} b_0 b_1 \dots b_{p-1} \dots z_0 z_1 \dots z_{p-1} a_p a_{p+1} \dots a_{2p-1} b_p b_{p+1} \dots b_{2p-1} \dots$

## Advantages

- limits number of seeks per data to  $\frac{1}{p}$
- allows updates, providing a small cache

# Compressing data

- 1 Data format
- 2 **Compressing data**
  - Compression algorithms
  - Efficiency and use cases
- 3 Data addressing
- 4 Stateful interfaces
- 5 Conclusion



# Data compression

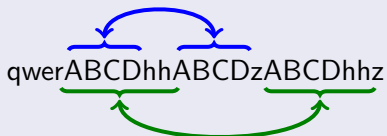
## Main idea

- eliminate redundancy (e.g. LZ77)
- optimize encoding (Huffman coding)
- to squeeze more information into less bytes

# Eliminate redundancy

LZ77

Replace redundancy with pointers


  
 qwerABCDhhABCDzABCDhhz

qwerABCDhh(6,4)z(11,6)z

# Optimize encoding

## Huffman coding idea

- use short codes for symbols repeated often
- build an optimal code set depending on symbols' frequencies

# Optimize encoding

Encoding "faeaebacdeeabadeacdfbdaddadcafdacbaababbaaccaa"

# Optimize encoding

Encoding “faeaebacdeeabadeacdfbdaddadcafdacbaababbaaccaa”

Frequencies

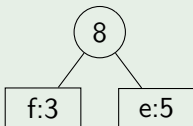
a	17	d	8
b	7	e	5
c	5	f	3

# Optimize encoding

Encoding "faeaebacdeeabadeacdfbdaddadcafdacbaababbaaccaa"

Frequencies

a 17	d 8
b 7	e 5
c 5	f 3



# Optimize encoding

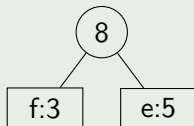
Encoding "faeaebacdeeabadeacdfbdaddadcafdacbaababbaaccaa"

Frequencies

a 17      d 8

b 7      ef 8

c 5



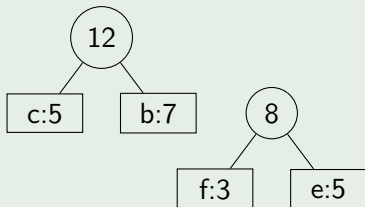
# Optimize encoding

Encoding "faeaebacdeeabadeacdfbdaddadcafdacbaababbaaccaa"

Frequencies

a 17      d 8

bc 12      ef 8





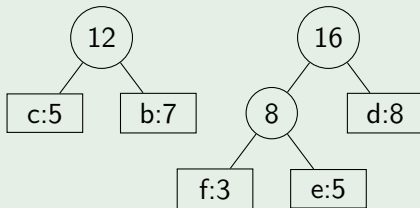
# Optimize encoding

Encoding "faeaebacdeeabadeacdfbdaddadcafdacbaababbaaccaa"

Frequencies

a 17      def 16

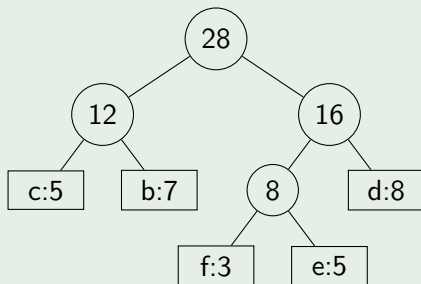
bc 12



# Optimize encoding

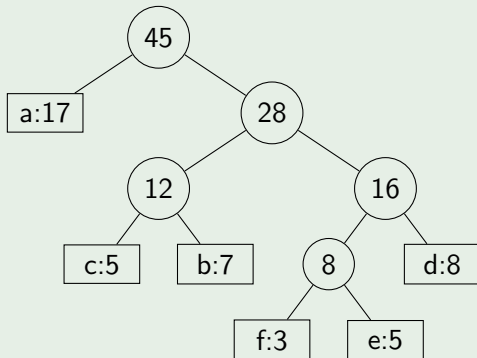
Encoding "faeaebacdeeabadeacdfbdaddadcafdacbaababbaacaa"

Frequencies  
 a 17    bcdef 28



# Optimize encoding

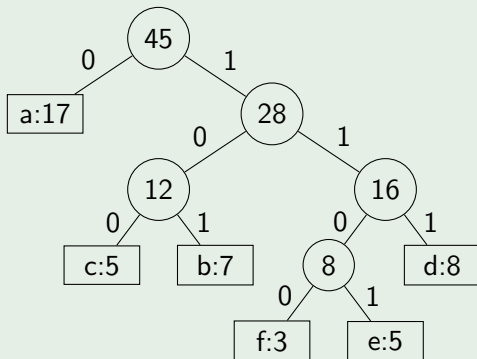
Encoding "faeaebacdeeabadeacdfbdaddadcafdacbaababbaaccaa"



Frequencies  
 all 45

# Optimize encoding

Encoding "faeaeabacdeeabadeacdfbdaddadcafdacbaababbaaccaa"

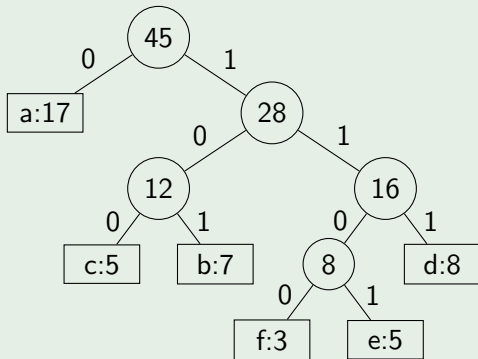


Frequencies

a	17	d	8
b	7	e	5
c	5	f	3

# Optimize encoding

Encoding "faeaeabacdeeabadeacdfbdaddadcafdacbaababbaaccaa"



## Frequencies

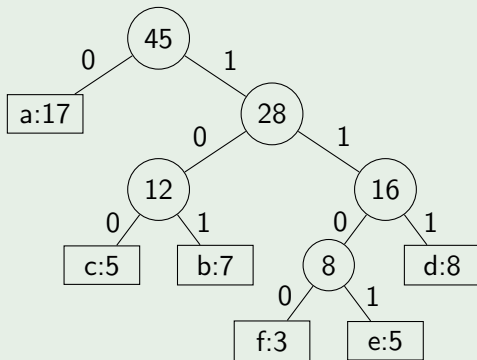
a	17	d	8
b	7	e	5
c	5	f	3

## Codes

a	0	d	111
b	101	e	1101
c	100	f	1100

# Optimize encoding

Encoding "faeabacdeeabadeacdfbdaddadcafdacbaababbaaccaa"



## Frequencies

a	17	d	8
b	7	e	5
c	5	f	3

## Codes

a	0	d	111
b	101	e	1101
c	100	f	1100



*f6b6a7dd57d4ff7bf78fe94ab490*  
28+6 bytes instead of 46

# Classical compression algorithms

## Best compression first

**bzip2** high quality, size within 10% to 15% to best compressors and twice faster to compress, six times faster to decompress

**gzip** classical zipping with *deflate* algorithm, combination of LZ77 and Huffman Coding

**LZO** fast - twice faster than gzip - but less good - 50% bigger result. Composed of small blocks of 256 kB of compressed data

**Snappy** even faster - 250MB/s encoding, 500MB/s decoding on single core i7 - but even less good. Use internally by Google

# A word on lossy compression

## What does it mean, lossy ?

- that you will lose part of your data !
- but that you may not see the difference
- and it may allow to compress MUCH better



# A word on lossy compression

## What does it mean, lossy ?

- that you will lose part of your data !
- but that you may not see the difference
- and it may allow to compress MUCH better

## Classical examples

`jpg` did you notice the images may be slightly blurred ?

`mp3` removes all the frequencies you cannot hear

# A word on lossy compression

## The main ideas

- 1 transformation of the data, usually based on fourier transforms
- 2 sensibility (nb bits) is reduced on less important parts
  - e.g. high frequencies
- 3 thanks to the previous cuts, a lot of similar values (0) appear
- 4 use standard compression

# A word on lossy compression

## The main ideas

- 1 transformation of the data, usually based on fourier transforms
- 2 sensibility (nb bits) is reduced on less important parts
  - e.g. high frequencies
- 3 thanks to the previous cuts, a lot of similar values (0) appear
- 4 use standard compression

## Usage in scientific computing

- mostly when encoding the data, that is within a subdetector
- very seldom at the level of complete data files

# Compression efficiency factors

## Algorithm - a small factor

- always a tradeoff between speed and compression
- best algorithms will gain maximum a factor 2 in size compared to fast ones

## Data - the main factor

- content** some contents are most compressable than others - zeros compress very well, while already compressed data don't
- format** human readable formats (XML) tend to compress very well as they are full of redundant markers
- structure** column storage compresses better than raw storage, as identical data types are close to each other

# Globally, is compression interesting ?

## Yes if

- your data are very cold, that is not frequently accessed
  - and the gain of space is worth the annoyance
  - tapes are compressing systematically
- you are really I/O bound and compression factor is high
- you use a fast compression algorithm (snappy) and gain space for little price

## No if

- you become CPU bound due to (de)compression
  - typical with bzip2
- you do frequent partial reads
- you need to split and cannot use LZO or bzip2

# Data addressing

- 1 Data format
- 2 Compressing data
- 3 Data addressing**
  - Hierarchical namespaces
  - Limitations
  - Flat namespaces
- 4 Stateful interfaces
- 5 Conclusion

# Data addressing

**The most important part of your data is your metadata**

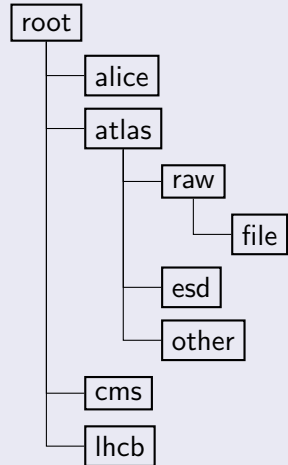
## Addressing options

- good old hierarchical namespace
- databases
- object store approach

# Traditionnal, hierarchical data addressing

## Directory tree

- structured via “directories”
- containing “files” and subdirectories
- filesystem like



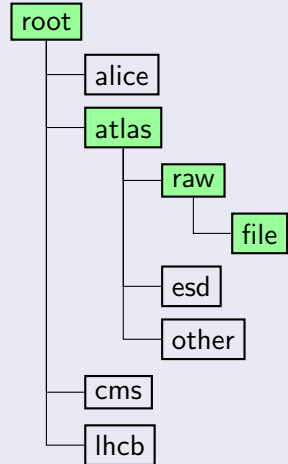


# Traditionnal, hierarchical data addressing

## Directory tree

- structured via “directories”
- containing “files” and subdirectories
- filesystem like

`/atlas/raw/file`



# Hierarchy pros and contras

## Pros

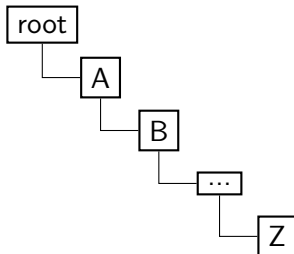
- easiness of use (for humans)
- helps to structure the data sets
  - easy to list/erase/deny access to a whole subtree
  - allows to implement quotas per directory
- atomic operations
  - in particular moves, removals
  - change of permissions
- always consistent

## Contras

- becomes slow for deep hierarchies
- in general scales badly

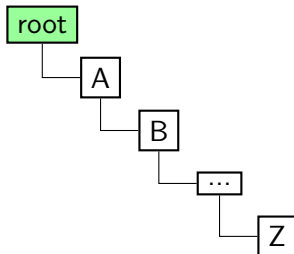
# Influence of depth

- Suppose you access '/A/B/C/.../Z'
- The following will happen :



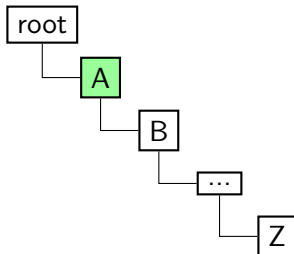
# Influence of depth

- Suppose you access `'/A/B/C/.../Z'`
- The following will happen :
  - find *root*
  - check you have execution right



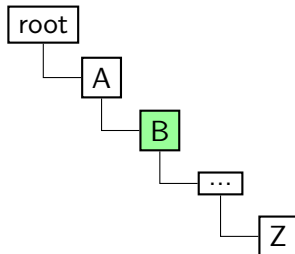
# Influence of depth

- Suppose you access `'/A/B/C/.../Z'`
- The following will happen :
  - find *root*
  - check you have execution right
  - find *A* within *root*
  - check you have execution right



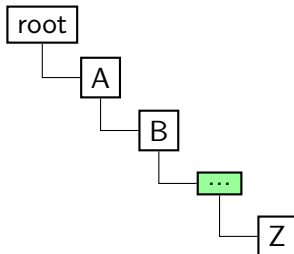
# Influence of depth

- Suppose you access `'/A/B/C/.../Z'`
- The following will happen :
  - find *root*
  - check you have execution right
  - find *A* within *root*
  - check you have execution right
  - find *B* within *A*
  - check you have execution right



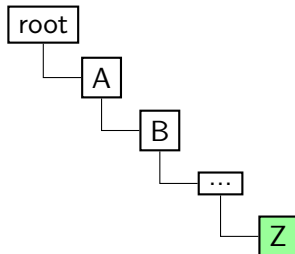
# Influence of depth

- Suppose you access `'/A/B/C/.../Z'`
- The following will happen :
  - find *root*
  - check you have execution right
  - find *A* within *root*
  - check you have execution right
  - find *B* within *A*
  - check you have execution right
  - ... repeat until *Y* ...



# Influence of depth

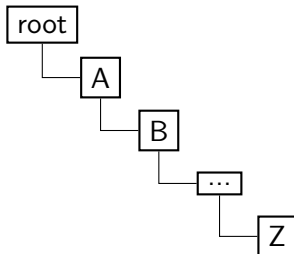
- Suppose you access '/A/B/C/.../Z'
- The following will happen :
  - find *root*
  - check you have execution right
  - find *A* within *root*
  - check you have execution right
  - find *B* within *A*
  - check you have execution right
  - ... repeat until *Y* ...
  - find *Z* within *Y*
  - check you have read/write rights





# Influence of depth

- Suppose you access `'/A/B/C/.../Z'`
- The following will happen :
  - find *root*
  - check you have execution right
  - find *A* within *root*
  - check you have execution right
  - find *B* within *A*
  - check you have execution right
  - ... repeat until *Y* ...
  - find *Z* within *Y*
  - check you have read/write rights



➔  **$2n + 1$  operations for depth  $n$**

depth seen at CERN (EOS, CASTOR) : 20-25

# Atomicity and consistency consequences

*root* is a bottle neck

- every single requests starts at root
- no easy parallelization of this part because of atomicity

# Atomicity and consistency consequences

*root* is a bottle neck

- every single requests starts at root
- no easy parallelization of this part because of atomicity

consistent recursive listing/searching is a nightmare

- you would need to lock all subtree for the whole listing time
- makes full listing impossible

# Some partial solutions - rights

## Precomputation

- precompute effective rights/size/... at every level

root 90 rwxr-xr-x

atlas 90 rwxr-xr-

raw 90 rwxrwxr-x

file 2000 r-xr-xr-x



# Some partial solutions - rights

## Precomputation

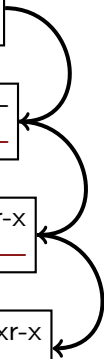
- precompute effective rights/size/... at every level

root 90 rwxr-xr-x  
 2270 rwxr-xr-x

atlas 90 rwxr-xr-  
 2180 rwxr-xr-

raw 90 rwxrwxr-x  
 2090 rwxr-x---

file 2000 r-xr-xr-x  
 2000 r-xr-x---



# Some partial solutions - rights

## Precomputation

- precompute effective rights/size/... at every level

## But...

- updating a top level right means full recomputation !
- the complete expansion for each file may be extremely heavy
- especially if complex ACLs are used at directory levels

root 90 rwxr-xr-x  
2270 rwxr-xr-x

atlas 90 rwxr-xr-  
2180 rwxr-xr-

raw 90 rwxrwxr-x  
2090 rwxr-x---

file 2000 r-xr-xr-x  
2000 r-xr-x---

# Some partial solutions - rights

## Avoiding full recomputation

- one can mark a subtree as “dirty”
- can then be recomputed offline
- in case we hit a dirty tree, we go back to tree scanning

# Some partial solutions - rights

## Avoiding full recomputation

- one can mark a subtree as “dirty”
- can then be recomputed offline
- in case we hit a dirty tree, we go back to tree scanning

## But...

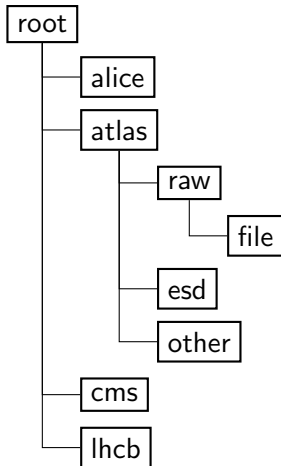
- tree scanning is needed to check for the dirty flag !
- actually, structures like bit map indexes can help speeding it up



# Some partial solutions - root bottleneck

## Path indexing

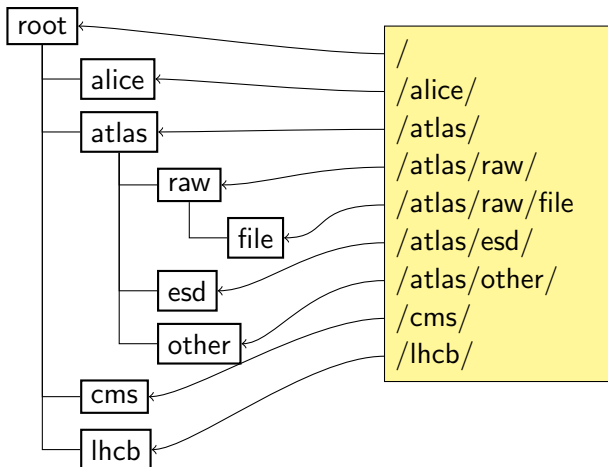
- keep an index of full paths and their locations



# Some partial solutions - root bottleneck

## Path indexing

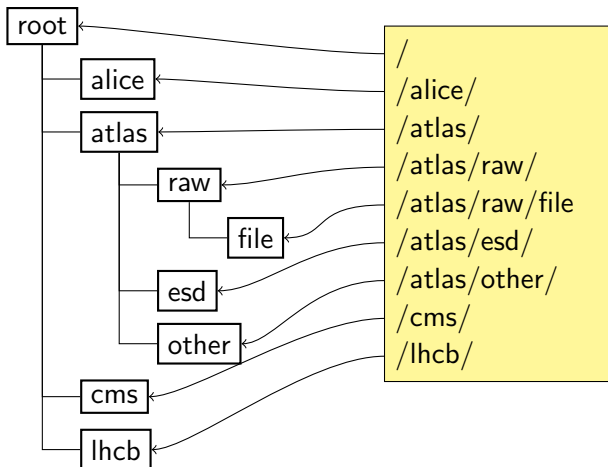
- keep an index of full paths and their locations



# Some partial solutions - root bottleneck

## Path indexing

- keep an index of full paths and their locations



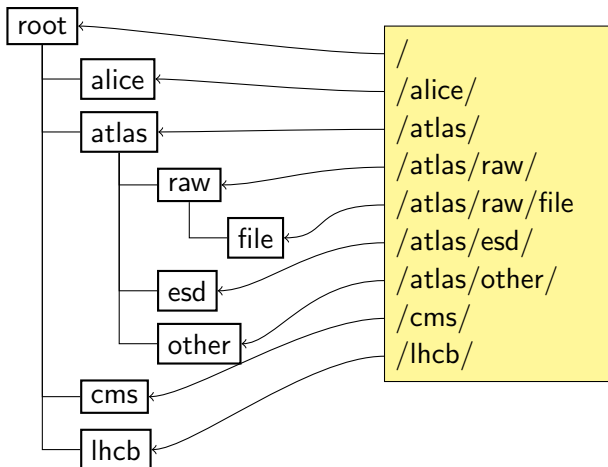
But...

- just think of a rename of atlas into titan...

# Some partial solutions - root bottleneck

## Path indexing

- keep an index of full paths and their locations



### But...

- just think of a rename of atlas into titan...

### Ideas

- use redirections, dirty flags, ...

# Some partial solutions - conclusion

→ very high complexity  
no scalable solution on the market so far

# Radical change : flat namespaces

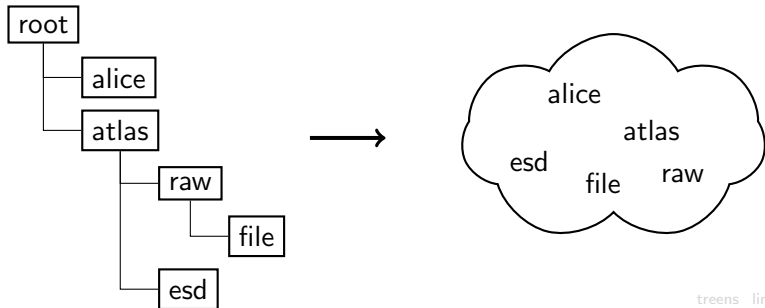
## The idea

- drop the hierarchy
- only have a sea of objects with a unique identifier
- forbid object modifications (only appends)
- forbid renames
  - or implement them as copy + drop

# Radical change : flat namespaces

## The idea

- drop the hierarchy
- only have a sea of objects with a unique identifier
- forbid object modifications (only appends)
- forbid renames
  - or implement them as copy + drop



# Radical change : flat namespaces

## Infinitely scalable namespace

- client can hash the entry path
- and connect accordingly to the right server



# Radical change : flat namespaces

## Infinitely scalable namespace

- client can hash the entry path
- and connect accordingly to the right server

## You lose

- data management aspect, left to user
- listing facilities
  - can be implemented in a Map/Reduce way

# About hashing on the client

## Large systems are dynamic

- i.e. they change over time
  - adding/removing hardware
  - changing network topology
- data may need to be rebalanced (see later)
- the client hashing algorithm needs to deal with that

# About hashing on the client

## Large systems are dynamic

- i.e. they change over time
  - adding/removing hardware
  - changing network topology
- data may need to be rebalanced (see later)
- the client hashing algorithm needs to deal with that

## CRUSH : the ceph's hash

- achieves pseudo-random but deterministic data distribution
- supports replication and erasure coding (see later)
- taking into account storage structure (machines, racks, rows, ...)
- minimizing the data movements in case of cluster modifications

# Stateful interfaces

- 1 Data format
- 2 Compressing data
- 3 Data addressing
- 4 Stateful interfaces**
  - POSIX
  - Limitations
  - Stateless interfaces
- 5 Conclusion

# The traditional POSIX interface for I/O

## Based on file descriptors

```
int open(const char *pathname, int flags);
int socket(int domain, int type, int protocol);
int connect(int sockfd,
            const struct sockaddr *addr,
            socklen_t addrlen);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd,
              const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int close (int fd);
int stat(const char *pathname, struct stat *buf);
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

# Key features and their consequences

## hierarchical namespace

- can be a limitation

## strong coherency

- last writer wins, metadata updated
- not suited for distributed environment, out of order concurrent writes
- lack of explicit atomicity
- metadata maintenance (e.g. last access time) is costly

## file oriented

- no bulk interfaces for metadata
- stateful, requires server to maintain a state per client
  - e.g. opendir / readdir

# Limitations of stateful interfaces

## Server resources

- any stateful client call means allocation of server resources
- these resources will accumulate with the number of clients
- limiting single server scalability

# Limitations of stateful interfaces

## Server resources

- any stateful client call means allocation of server resources
- these resources will accumulate with the number of clients
- limiting single server scalability

## Parallelization issues

- multiple servers in parallel would need to share resources
  - which creates a single point of failure and a possible contention
- alternatively you can dedicate servers to clients
  - but you lose server fail over
  - and load balancing



# Limitations of stateful interfaces

## Server resources

- any stateful client call means allocation of server resources
- these resources will accumulate with the number of clients
- limiting single server scalability

## Parallelization issues

- multiple servers in parallel would need to share resources
  - which creates a single point of failure and a possible contention
- alternatively you can dedicate servers to clients
  - but you lose server fail over
  - and load balancing

**Practically stateful interfaces do not scale**

# Stateless interfaces

## Approach

- each client request must bring enough information to complete it without context
- simply the state is stored on the client

# Stateless interfaces

## Approach

- each client request must bring enough information to complete it without context
- simply the state is stored on the client

## Example of listing

- stateful

```
DIR *opendir(const char *name);  
struct dirent *readdir(DIR *dirp);  
int closedir(DIR *dirp);
```

- stateless

```
struct dirent *readdir(const char *name);  
struct dirent *readdir(struct dirent* current);
```

- providing you have a strict order defined

# Conclusion

- 1 Data format
- 2 Compressing data
- 3 Data addressing
- 4 Stateful interfaces
- 5 Conclusion**

# Conclusion

## Key messages of the day

- Choose well your data format
- Think twice before compressing
- Value your metadata
- Think stateless for scalability