Git for Data Lakes

How lakeFS Scales Data Versioning to Billions of Objects

Amit Kesarwani
Director of Solution Engineering

lakeFS


38
RETENTION.SCALA

Amit Kesarwani: Solutions Architect

github.com/kesarwam

@AmitKesarwani

github.com/treeverse/lakeFS

@lakeFS

axolotl

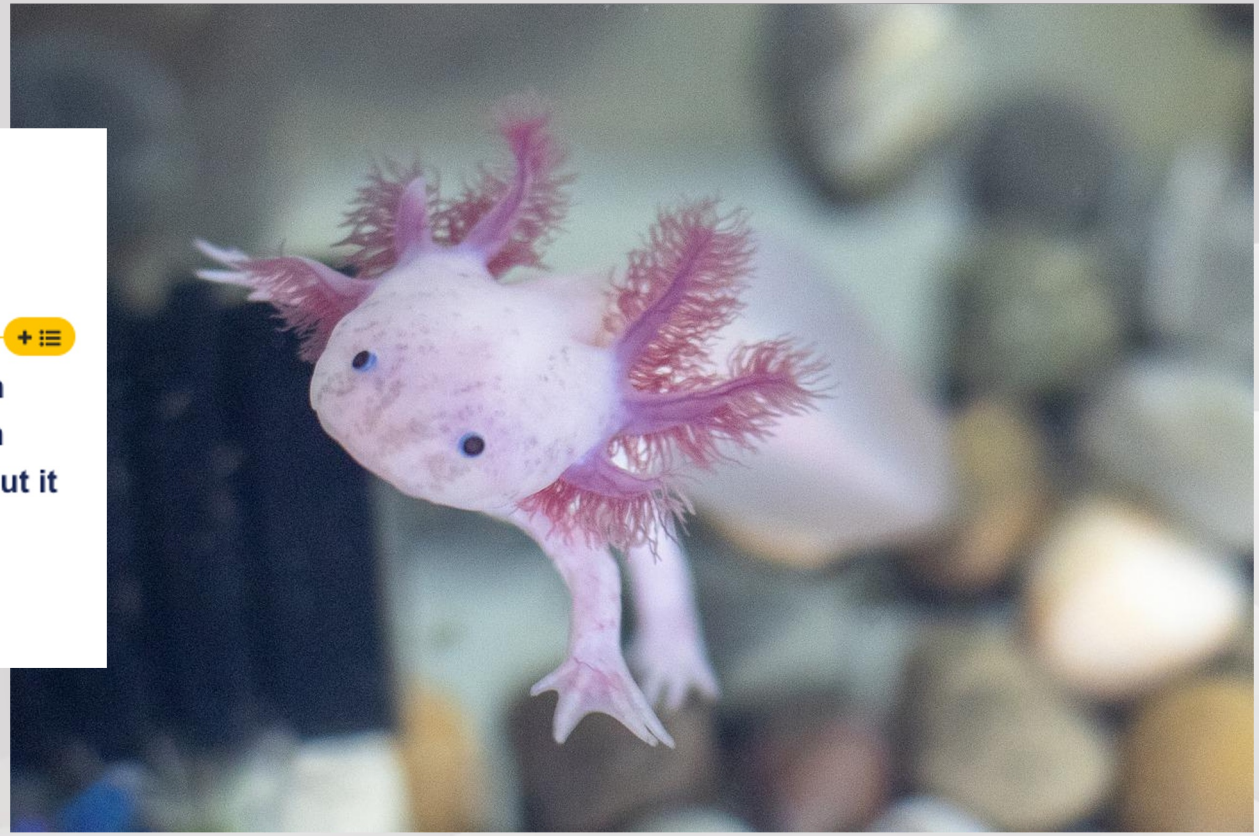
noun [C]

UK /ˈæk.sə.lɒt.ᵊl/ US /ˈæk.sə.lɑː.ɾᵊl/

a small animal that lives in water and looks like a fish with four legs. An axolotl is a type of amphibian (= an animal that usually lives both on land and in water) but it only lives in water.

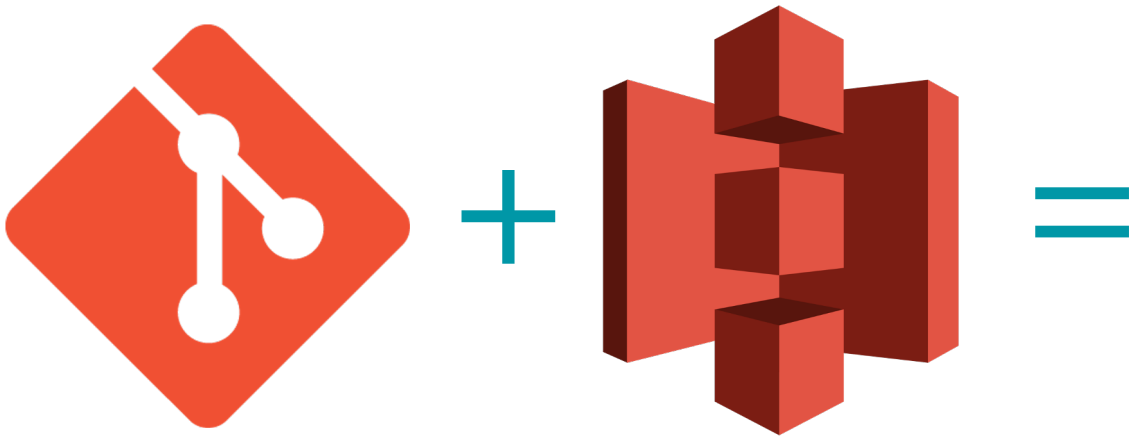
GlobalP/iStock / Getty Images Plus/GettyImages

https://dictionary.cambridge.org/dictionary/english/axolotl



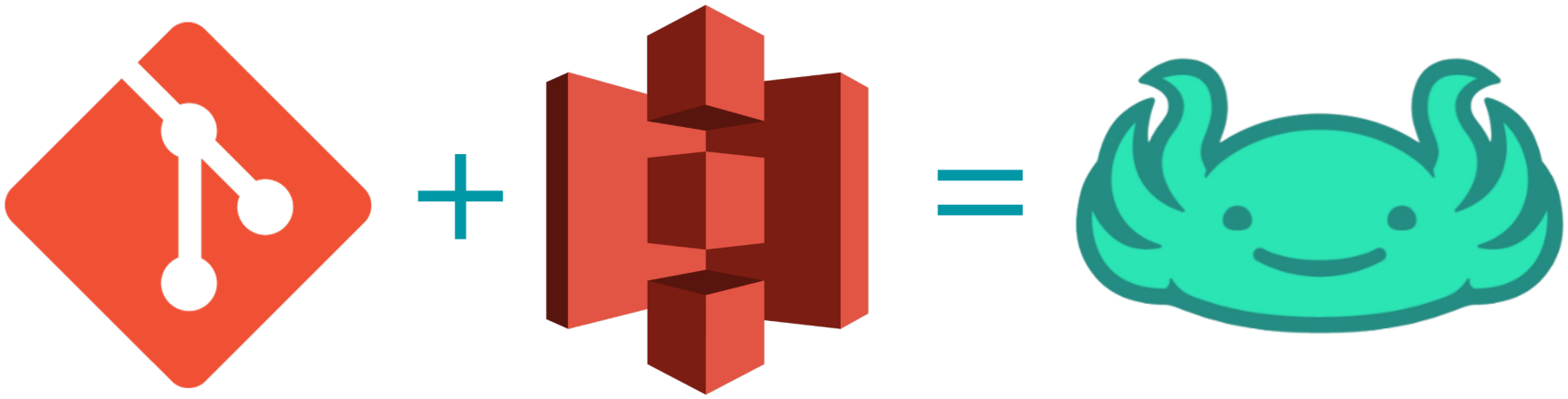
What if we had Git?

But, you know, that scales to S3 sizes?



What if we had Git?

But, you know, that scales to S3 sizes?



Wishlist

Sometimes referred to as “requirements”

- Cheap, CoW Branching
- Fast
- Efficient diffing/merging
- ~~Intuitive~~ Familiar branching committing and merging semantics

OK, so how do you scale the Git model to billions of objects?

Attempt #1

Let’s use Git!

Metadata (pointers to objects)

Data (objects)

Attempt #1

Let’s use Git!

Metadata (pointers to objects)

Data (objects)

```
root tree - e8d7621521

100644 blob f45a79533c   README.md
040000 tree b5df52155c   events
040000 tree 9fadcccbbc   marketing-data
040000 tree 01ffd0da88   sales-data
040000 tree 5cc330844a   raw-events
```

```
tree - b5df52155c

040000 tree f45a79537f   country=US
040000 tree 7bdf5217a2   country=IT
040000 tree f9adcec9ee   country=IL
040000 tree 18ffd0d216   country=UK
040000 tree d8c33089c1   country=NL
```

```
tree - 01ffd0da88

040000 tree 40218ea9e4 year=1995
040000 tree 76d4e36205 year=1996
040000 tree 54d483d269 year=1997
040000 tree f0c3bfe2a3 year=1998
040000 tree 70e3e07f8a year=1999
040000 tree 80430d4c57 year=2000
040000 tree 1b2d99146a year=2001
040000 tree d5203bd4b1 year=2002
040000 tree f4203a7b5e year=2003
040000 tree 059496b48f year=2004
040000 tree 43893289a2 year=2005
040000 tree 60a6a68b57 year=2006
040000 tree 8a243ccd59 year=2007
040000 tree 0d9f60fa7a year=2008
040000 tree cb692970d7 year=2009
040000 tree bbf9bfef4b year=2010
040000 tree 0325a8040b year=2011
040000 tree d84c7b4646 year=2012
```

Attempt #1

Let’s use Git!

Metadata (pointers to objects)

Data (objects)

- ✓ Cheap, CoW branching
- ✗ Fast
- ✗ Efficient diffing/merging
- ✓ ~~Intuitive~~ Familiar branching committing and merging semantics

Attempt #2

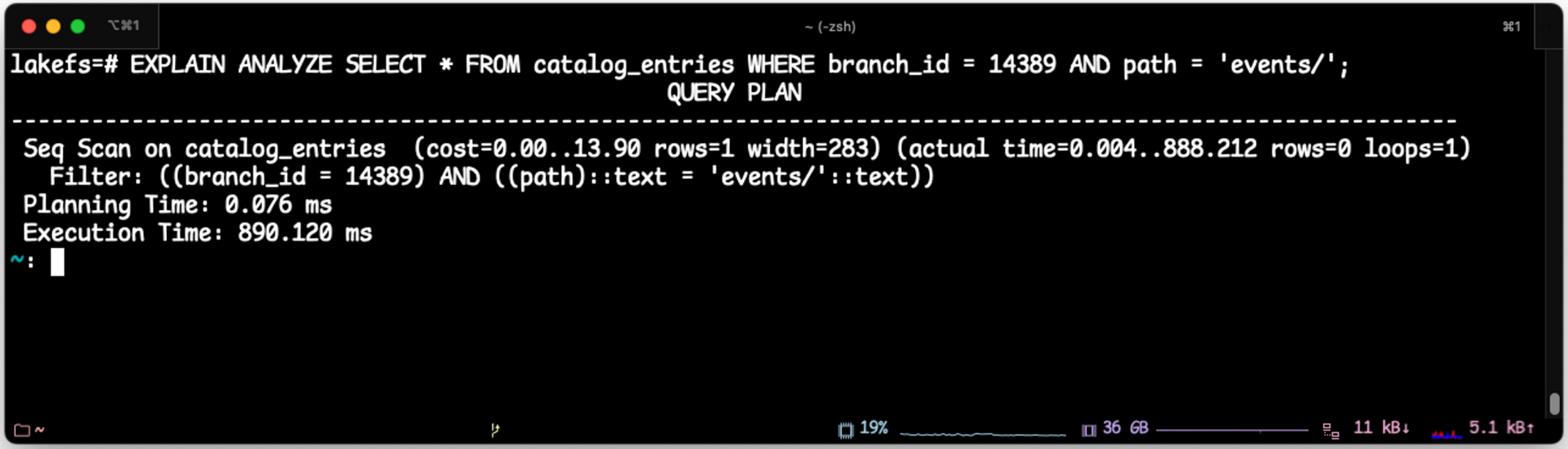
Let’s use a database!

Metadata (pointers to objects)

Data (objects)

Attempt #2

Let’s use a database!



Attempt #2

Let’s use a database!

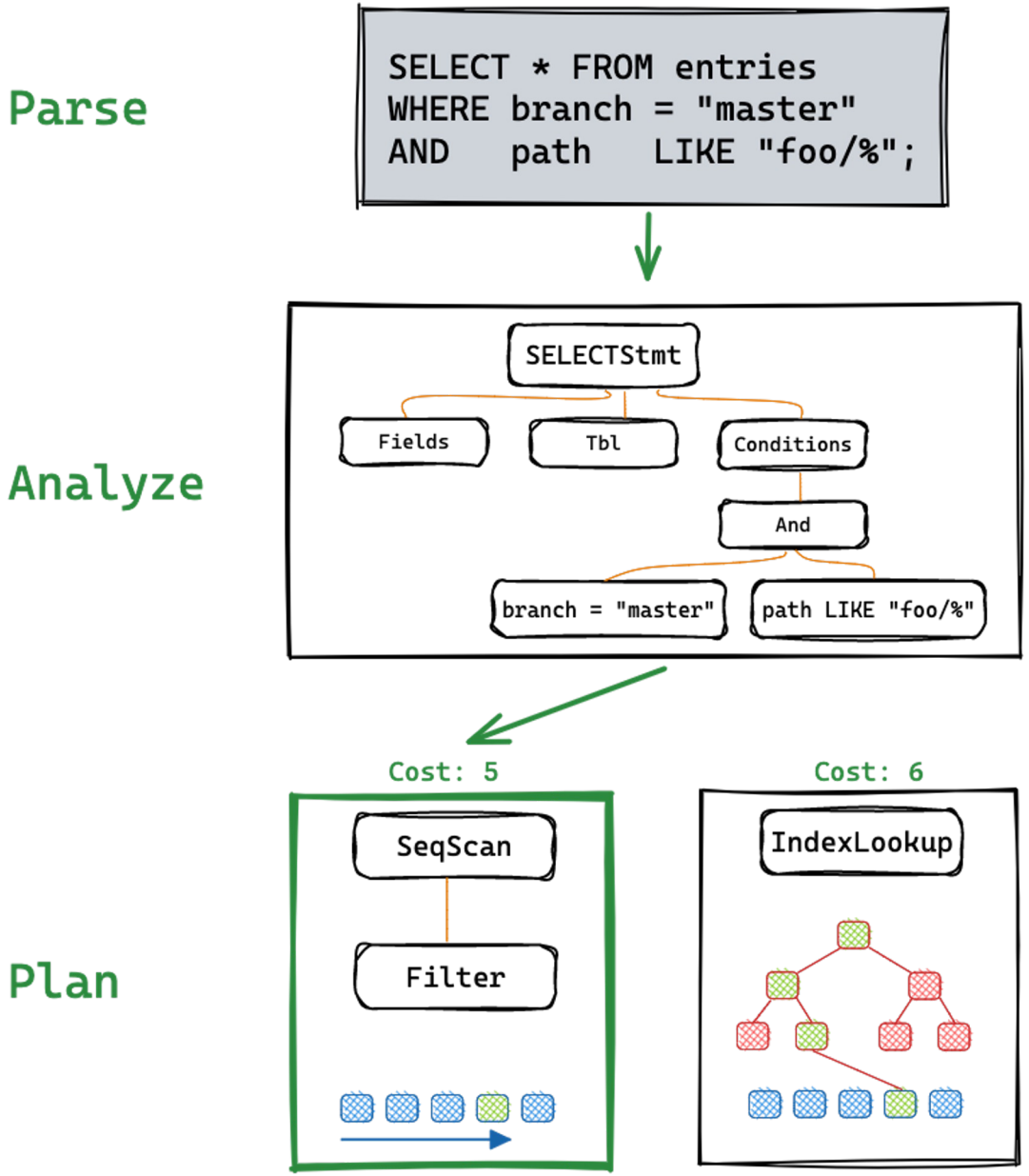
```
lakefs=# EXPLAIN ANALYZE SELECT * FROM catalog_entries WHERE branch_id = 14389 AND path = 'events/';
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Seq Scan on catalog_entries  (cost=0.00..13.90 rows=1 width=283) (actual time=0.004..888.212 rows=0 loops=1)
   Filter: ((branch_id = 14389) AND ((path)::text = 'events/'::text))
 Planning Time: 0.076 ms
 Execution Time: 890.120 ms
~:
```

Attempt #2

Let’s use a database!

Cost Based Optimization

- Mostly based on table statistics
- Won’t always use the index
- Bad when growing/shrinking 1000x



Attempt #2

Let’s use a database!

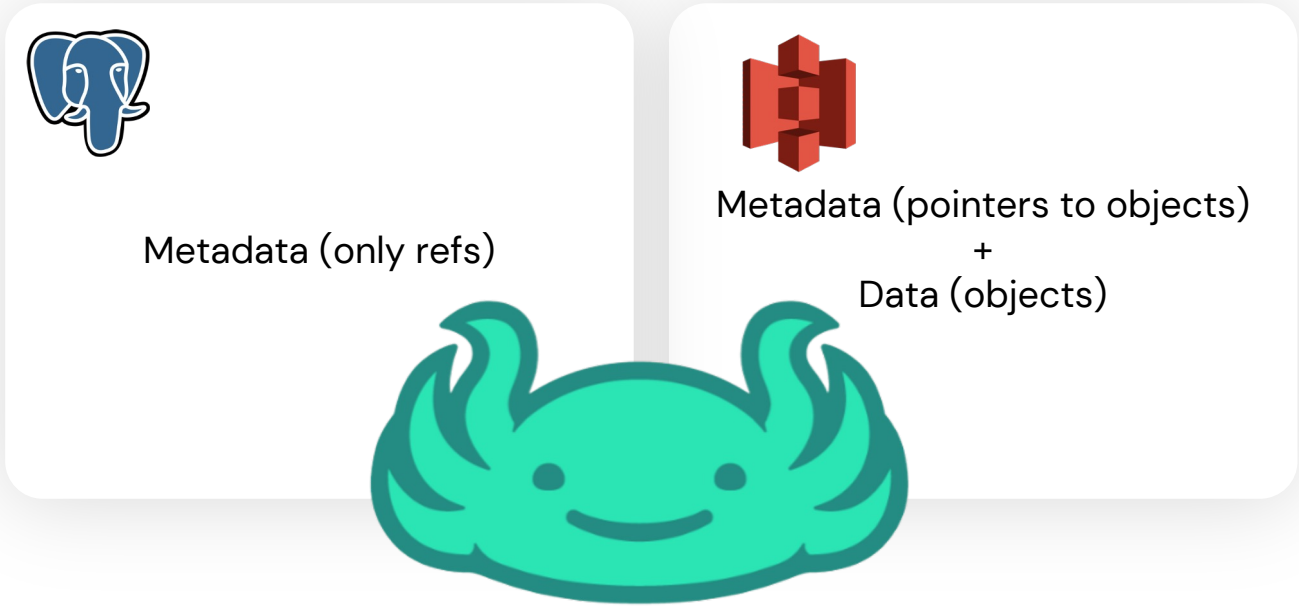
Metadata (pointers to objects)

Data (objects)

- ✓ Cheap, CoW branching
- ✗ Fast
- ✗ Efficient diffing/merging
- ✓ ~~Intuitive~~ Familiar branching committing and merging semantics
- ✗ Predictable behavior
- ✗ Easy to extend and maintain

Attempt #3

Let’s not use a database!*



* Almost.

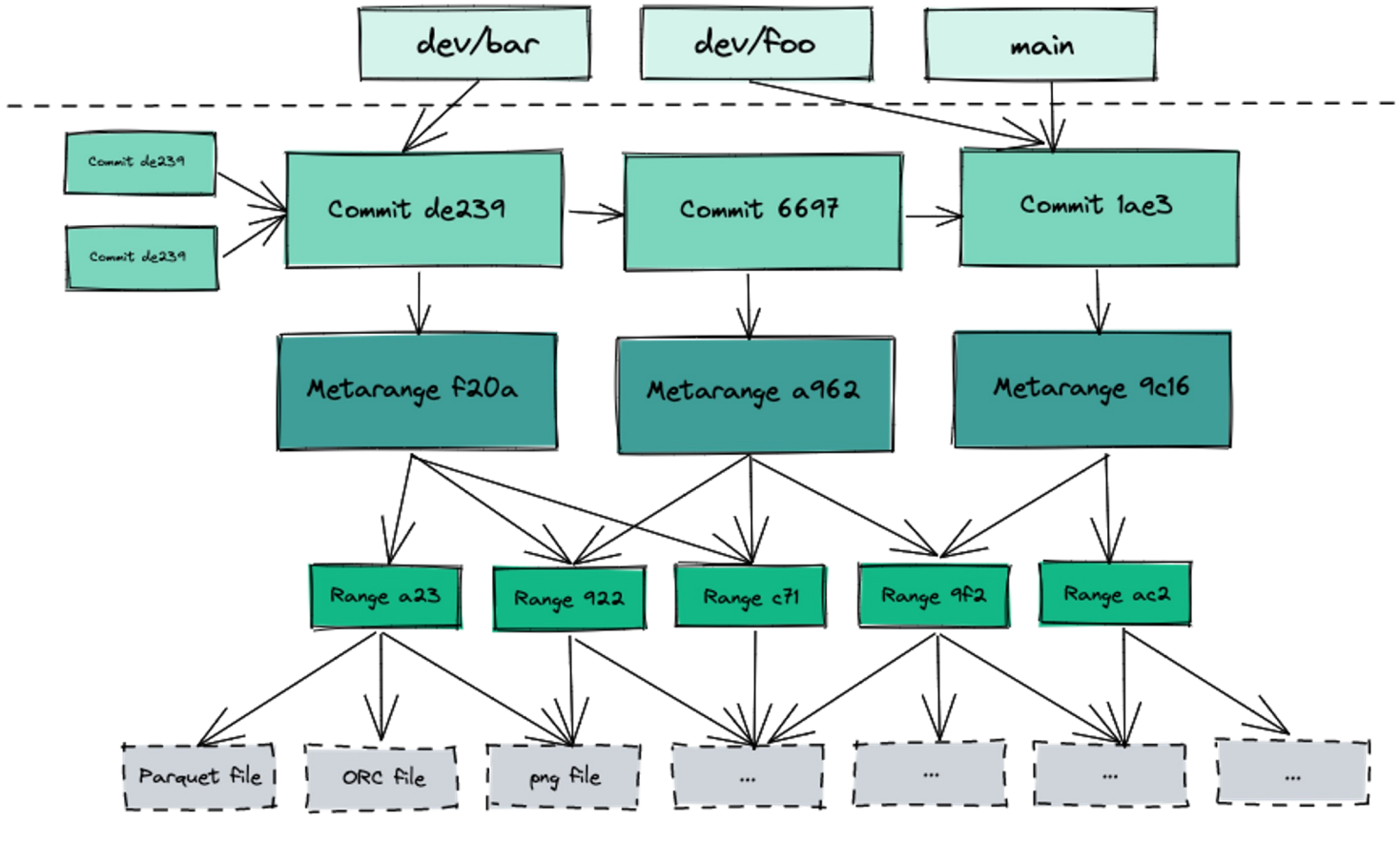
Attempt #3

Let’s not use a database!

Metadata (refs only)

Metadata (pointers to objects)

Data (Objects)



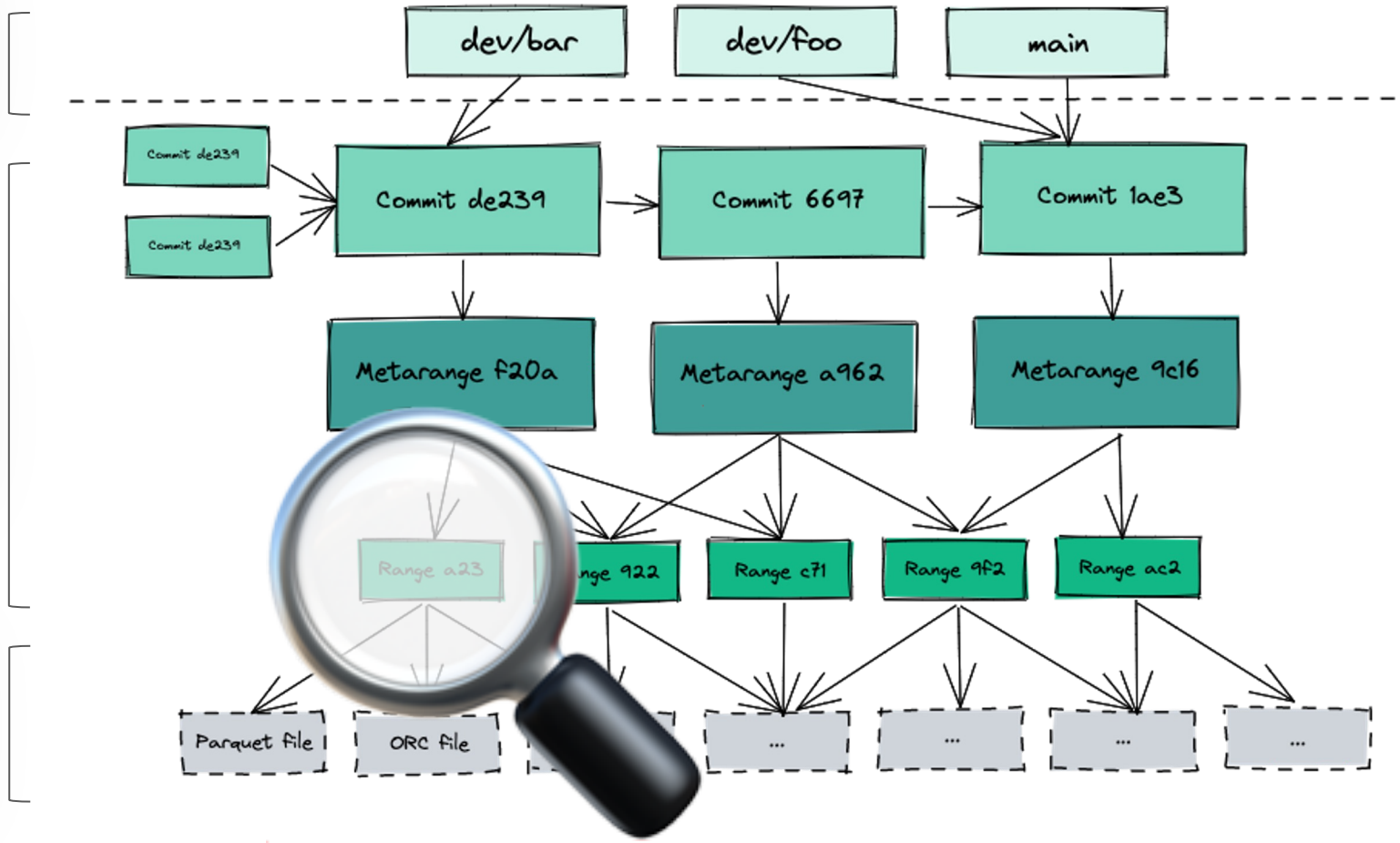
Attempt #3

Let’s not use a database!

Metadata (refs only)

Metadata (pointers to objects)

Data (Objects)

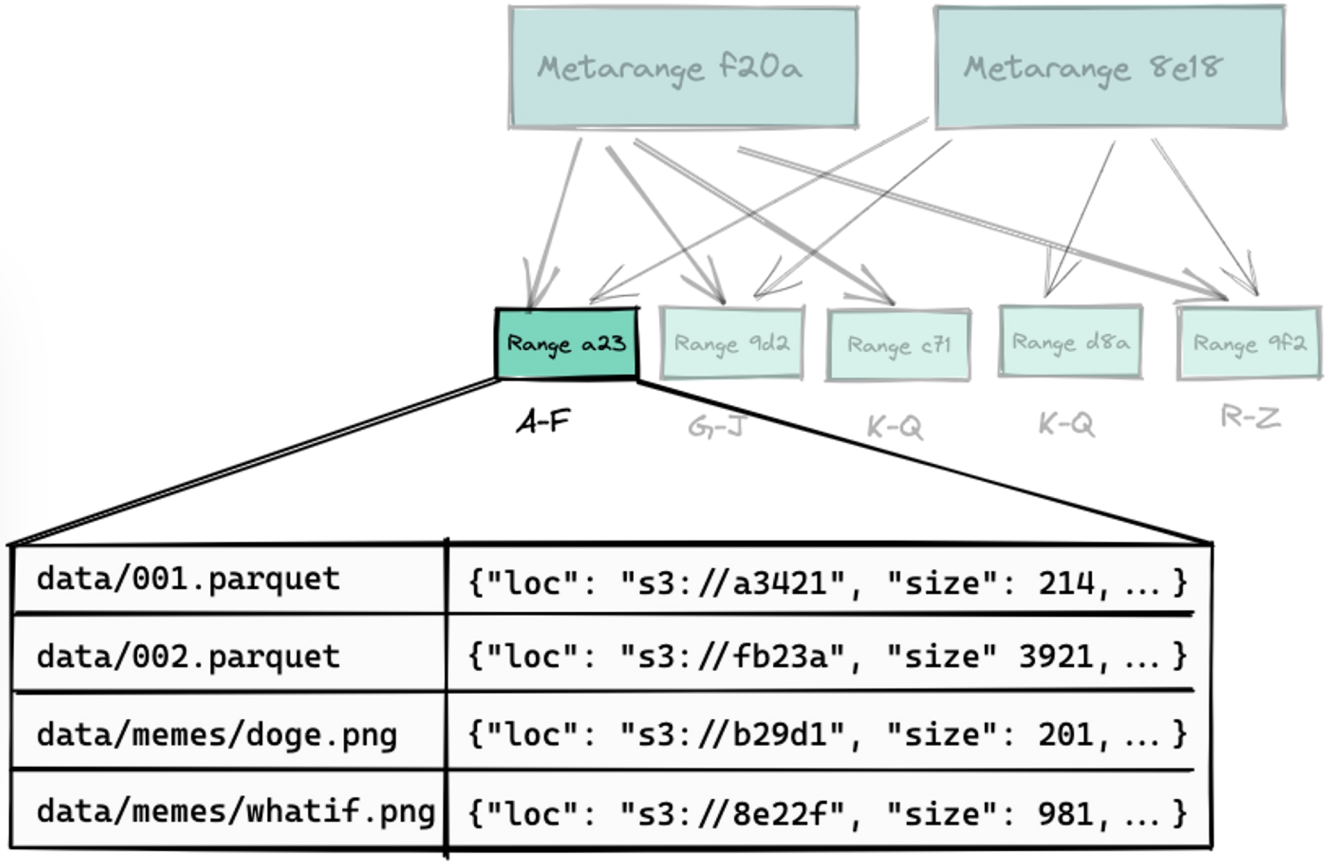


Attempt #3

Let’s not use a database!

Ranges

- Key/value pairs
- Lexicographically sorted paths
- Balancing throughput and latency:
  1-8 MB in size
- Immutable, hash addressed



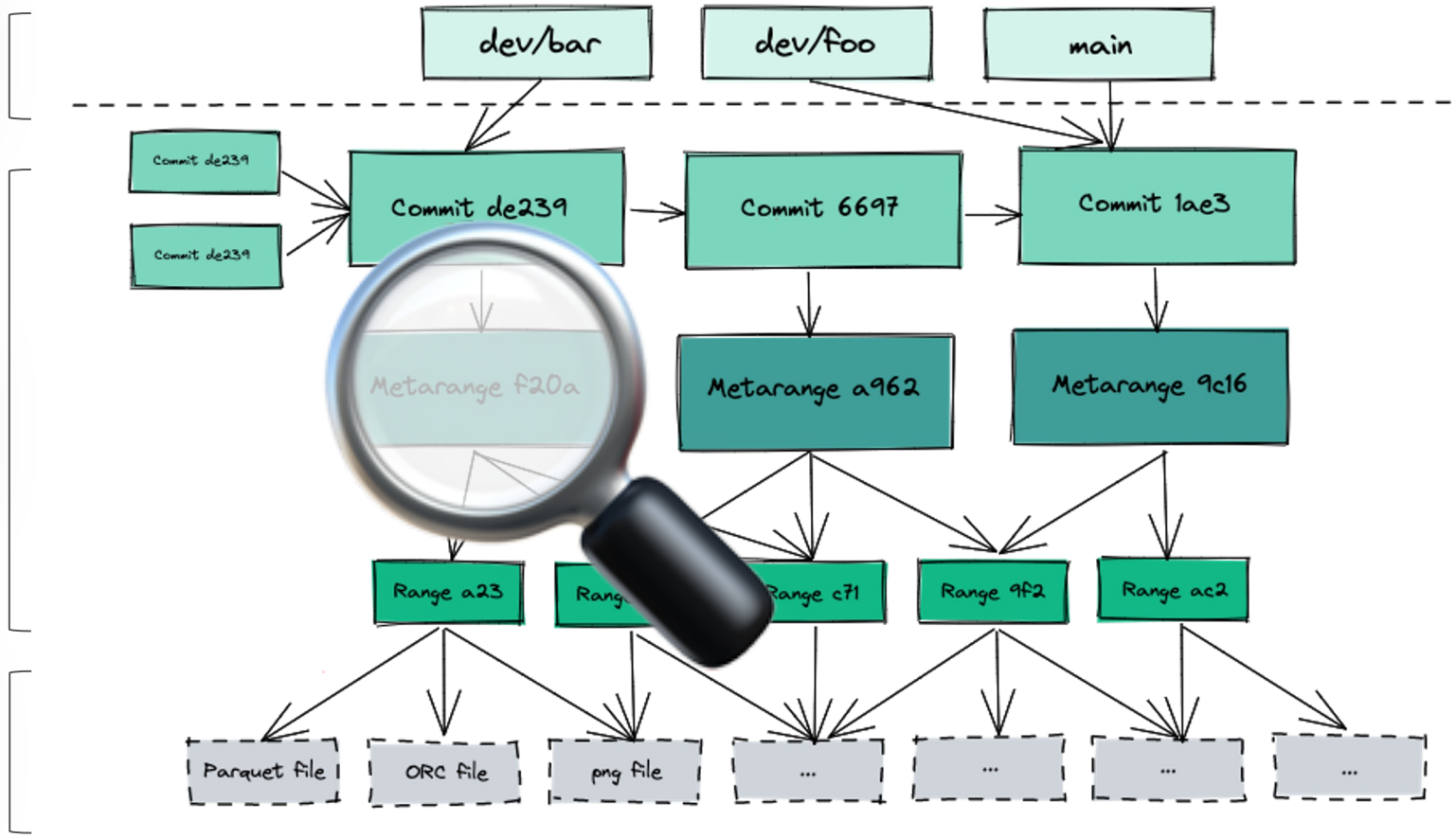
Attempt #3

Let’s not use a database!

Metadata (refs only)

Metadata (pointers to objects)

Data (Objects)

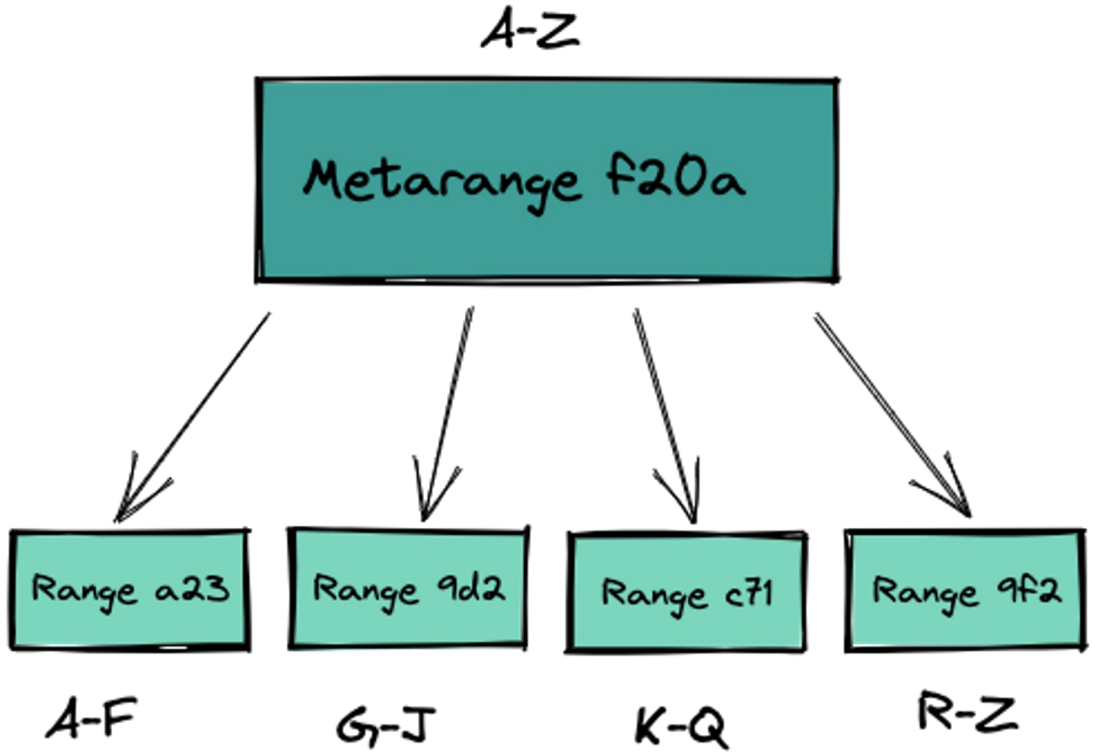


Attempt #3

Let’s not use a database!

Metaranges

- Are ranges!
- That point to ranges!
- These ranges do not overlap

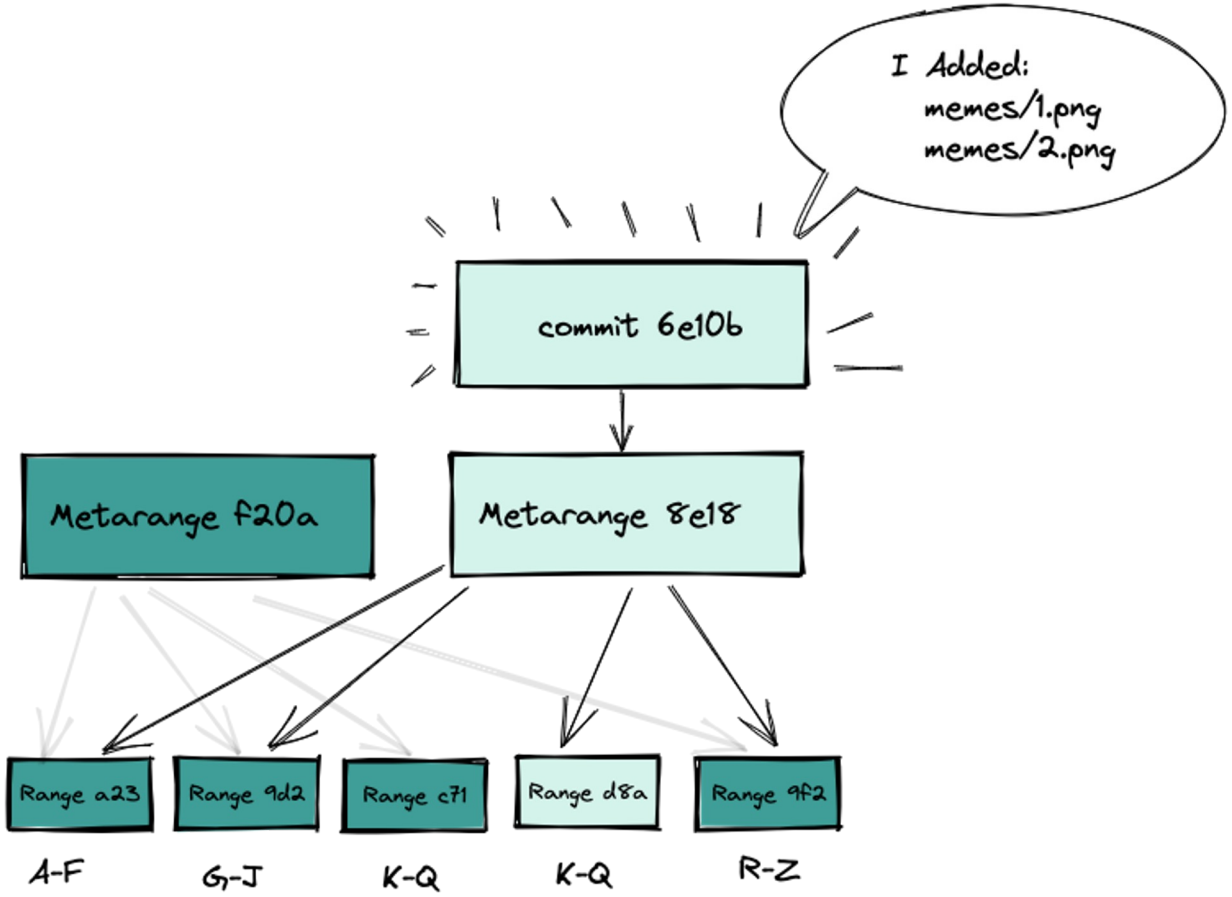


Attempt #3

Let’s not use a database!

Commits

- Are pointers to metaranges
- Space = O(diff)

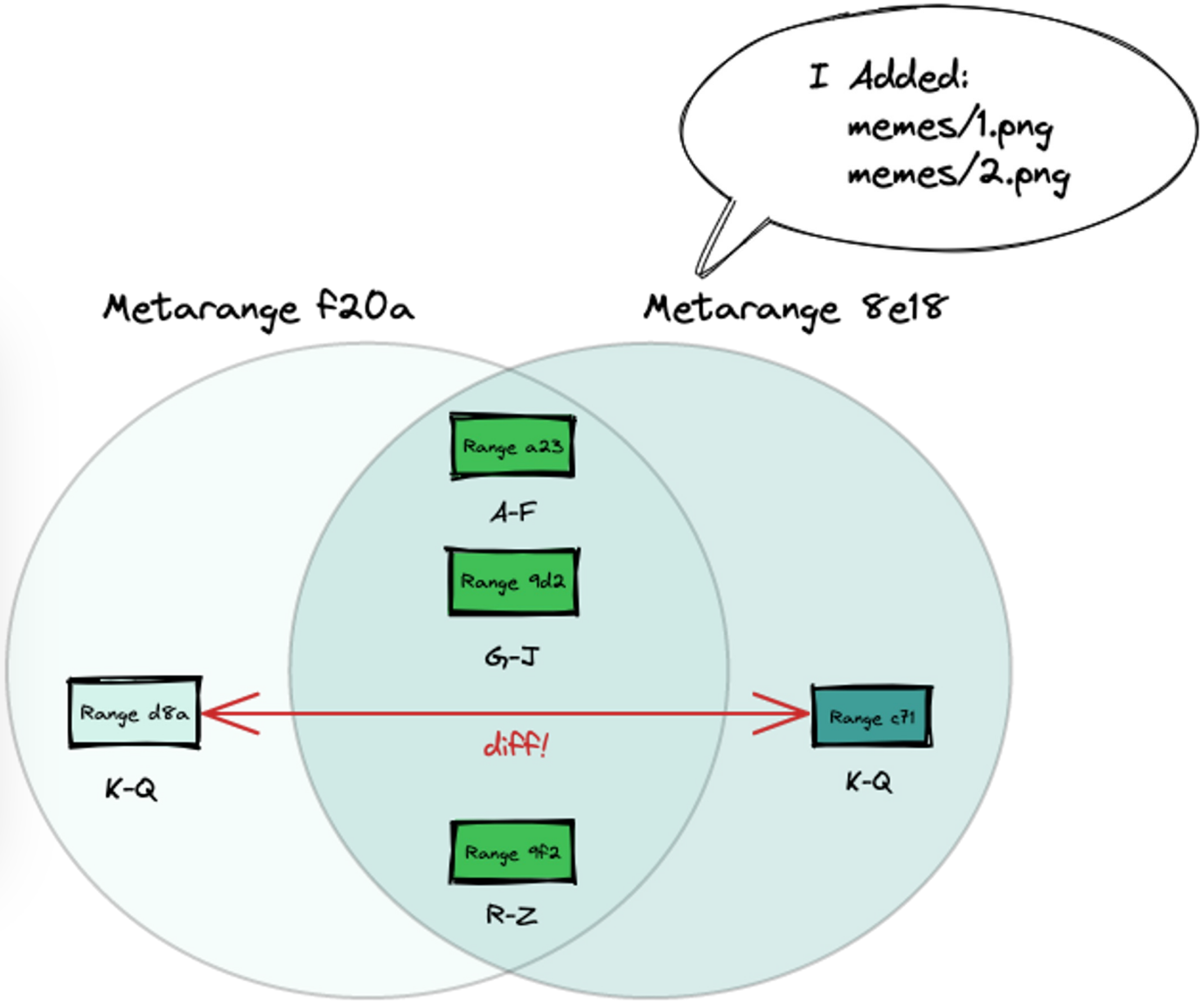


Attempt #3

Let’s not use a database!

Diffing and Merging

- Are efficient!
- Time = O(diff)



Attempt #3

Let’s not use a database!

Metadata (only refs)

Metadata (pointers to objects)
+
Data (objects)

- Cheap, CoW branching
- Fast…? 🤷
- Efficient diffing/merging
- ~~Intuitive~~ Familiar branching committing and merging semantics

Attempt #3

Let’s not use a database!

Metadata (only refs)

Metadata (pointers to objects)
+
Data (objects)

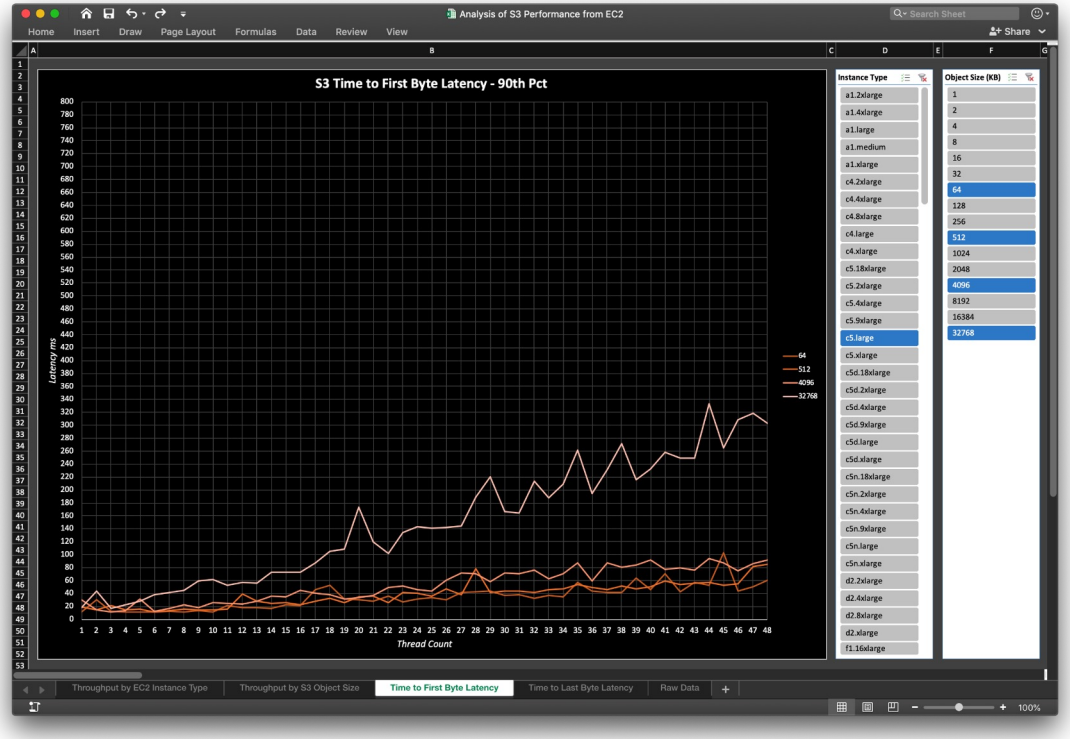
- ✓ Cheap, CoW branching
- ✗ Fast 😢
- ✓ Efficient diffing/merging
- ✓ ~~Intuitive~~ Familiar branching committing and merging semantics

Attempt #3

Let’s not use a database!

Object store != Key Value Store

- TTFB is *high* (tens of ms)
- Gets worse at higher percentiles



https://github.com/dvassallo/s3-benchmark

Attempt #3.5

Caching!

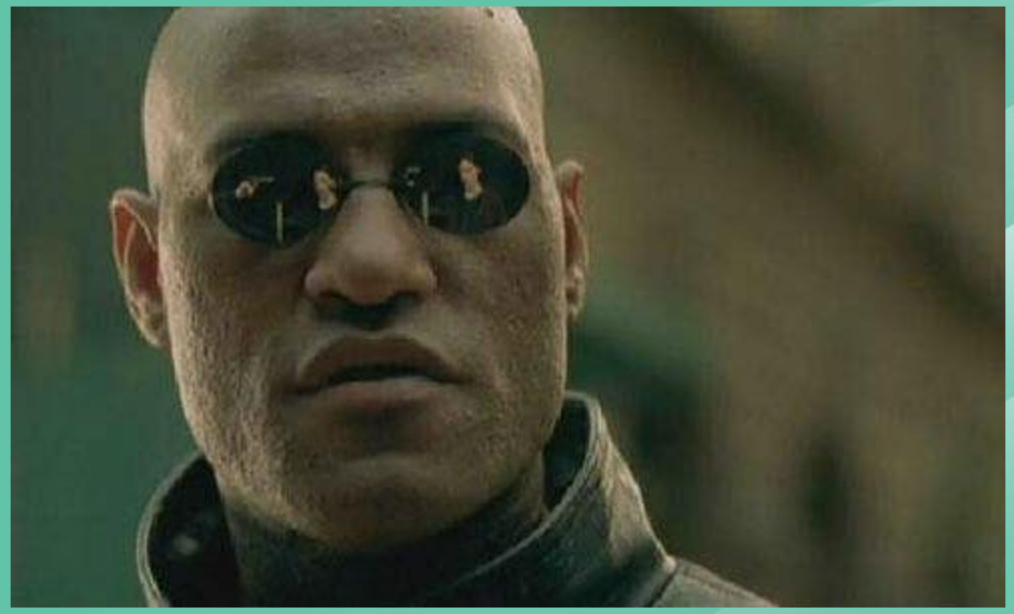
Metadata (only refs)

Metadata (pointers to objects)
+
Data (objects)

- ✓ Cheap, CoW branching
- ✓ Fast!
- ✓ Efficient diffing/merging
- ✓ ~~Intuitive~~ Familiar branching committing and merging semantics

*“There are only two hard things in Computer Science: **cache invalidation** and naming things.”*

— Phil Karlton



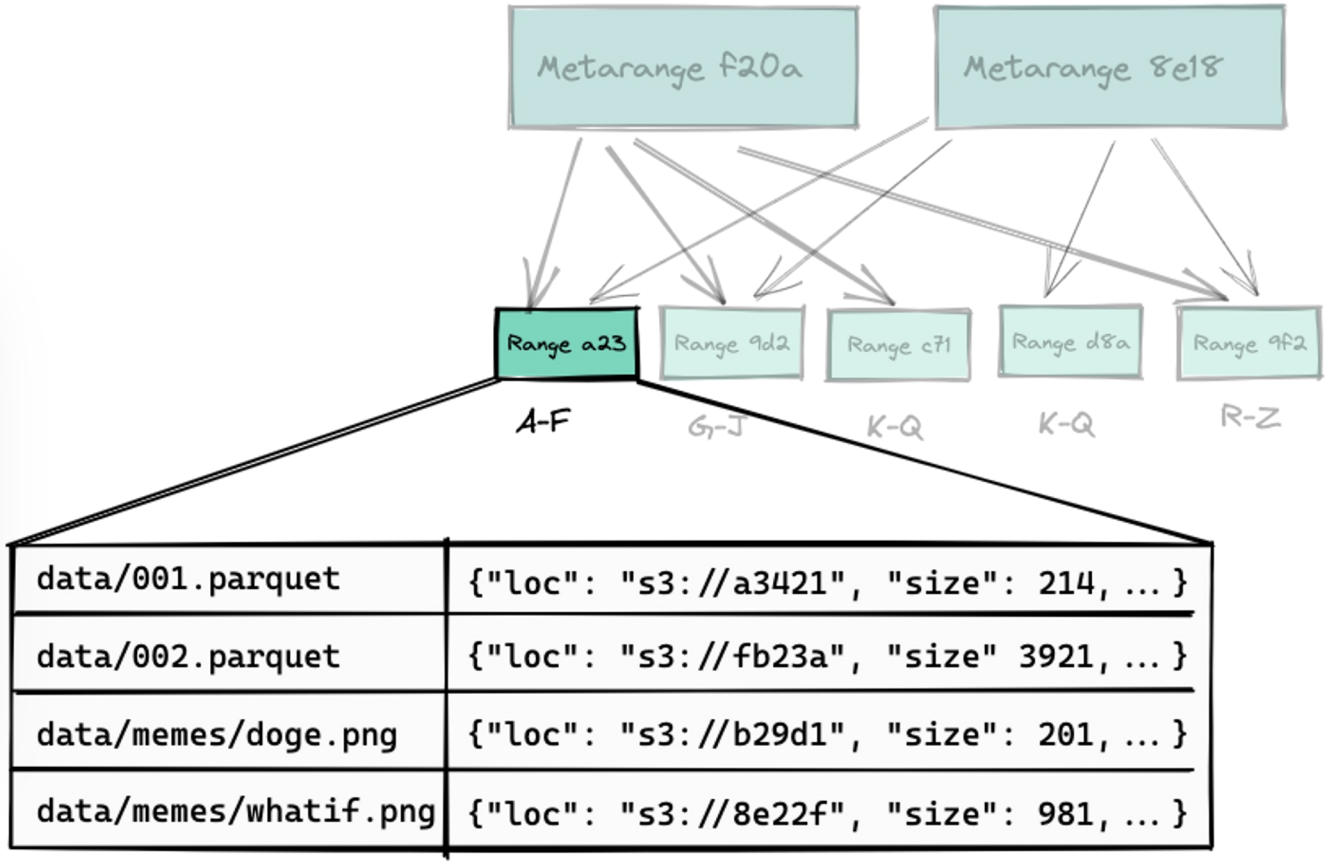
Attempt #3.5

Caching

Ranges

- Key/value pairs
- Lexicographically sorted paths
- Balancing throughput and latency:
  1-8 MB in size
- **Immutable, hash addressed**

So no invalidation necessary



Attempt #3.5

Caching!

Metadata (only refs)

Metadata (pointers to objects)
+
Data (objects)

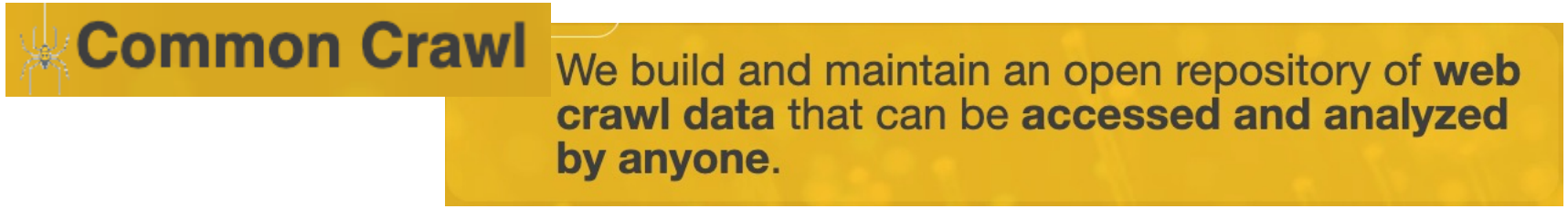
- Cheap, CoW branching
- **Fast!**
- Efficient diffing/merging
- ~~Intuitive~~ Familiar branching committing and merging semantics

Demo Time

Demo: Use Case

Cleaning the internet with lakeFS and Spark

https://commoncrawl.org/



https://www.kaggle.com/datasets/taruntiwarihp/phishing-site-urls

Delete phishing sites from Common Crawl's data

What can we learn from this?

Some key points

- Define constraints early
- You cannot predict your next bottleneck
- **Choosing a correct data model is 80% of the work**


38
RETENTION.SCALA


38
RETENTION.SCALA

Learn More

https://lakeFS.io/

github.com/treeverse/lakeFS

https://lakefs.io/community

Thanks!

lakeFS

Director of Solution Engineering

amit.kesarwani@treeverse.io