

# PuffinDB Meetup

Rovinj, Croatia, March 29-31, 2023



# 🎯 Objectives

What do we want to accomplish during this meetup?

1. Meet the team
2. Review project
3. Discuss architecture
4. Identify challenges
5. Outline game plan

# Agenda

How will we use our limited time together?

## Wednesday 3/29

- |               |                                   |                |
|---------------|-----------------------------------|----------------|
| 11:00 – 12:00 | Arrival & check-In                |                |
| 12:00 – 13:00 | Kick-off session in meeting room  | Meet the team  |
| 13:00 – 14:00 | Lunch at Laurel & Berry           |                |
| 14:00 – 17:00 | Working session with coffee break | Review project |
| 19:00 – 21:00 | Dinner at Kantonon Tavern         |                |

## Thursday 3/30

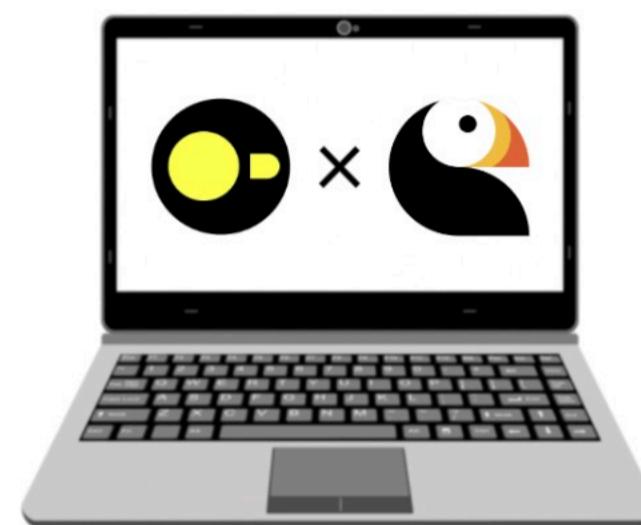
- |               |                                   |   |
|---------------|-----------------------------------|---|
| 08:30 – 11:30 | Working session with coffee break | Discuss architecture                      |
| 12:00 – 13:30 | Lunch at La Riva                  |   |
| 14:00 – 17:00 | Working session with coffee break | Identify challenges and discuss game plan |
| 19:00 – 21:00 | Dinner at Gianino                 |   |

## Friday 3/31

- |               |                                |
|---------------|--------------------------------|
| 08:00 – 11:00 | Bus transfer to Zagreb Airport |
|---------------|--------------------------------|

# 🚀 Onboarding

What is PuffinDB and how can we make it frictionless?



1. Add PuffinDB extension to any client embedding DuckDB

Read & Write Iceberg Tables for Real-Time Collaboration

⋮

2. Install PuffinDB AWS Template

Run DuckDB on EC2 and Lambdas

⋮



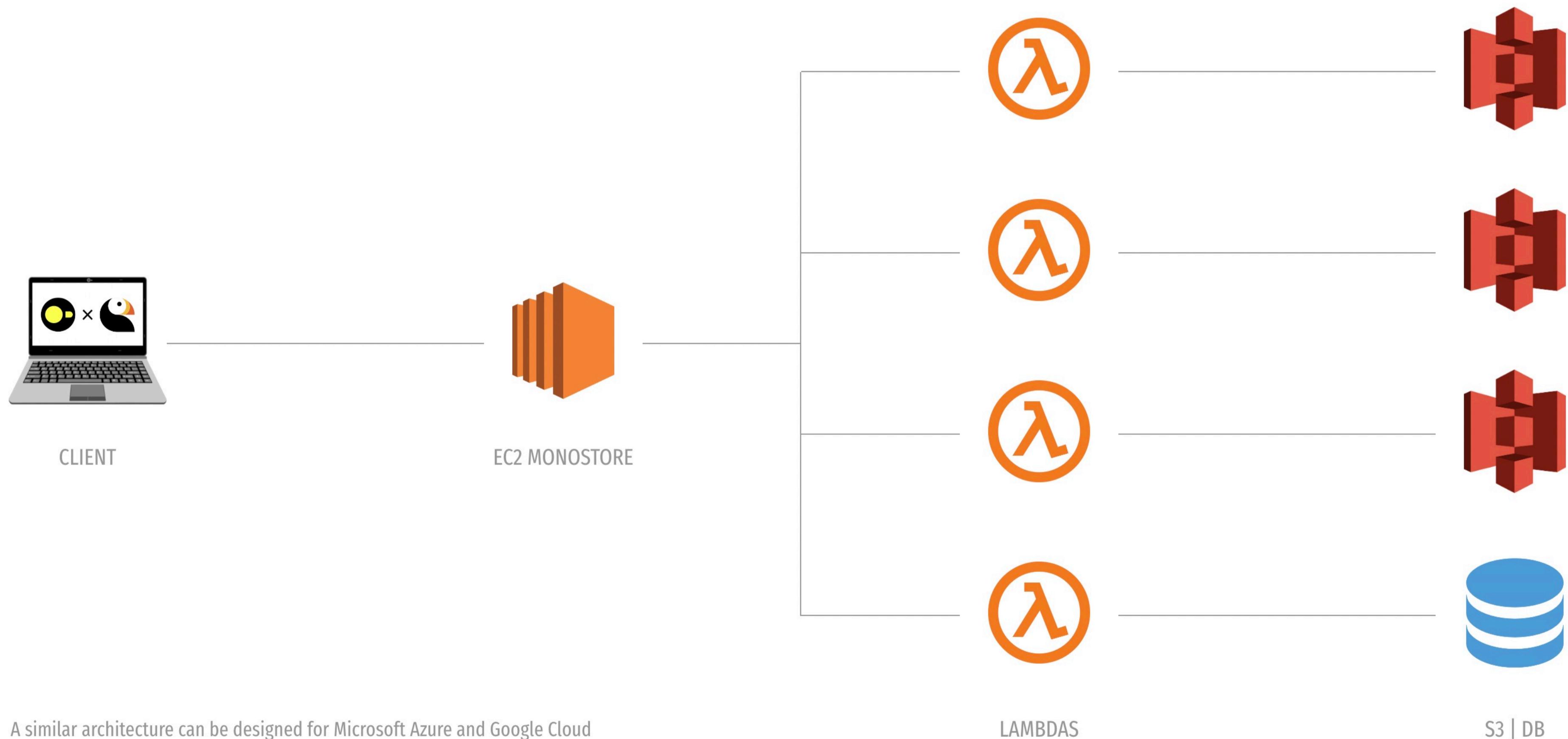
3. Distribute queries across client, EC2 monostore, and Lambdas

Integration · Transformation · Transactions · Analysis



# Architecture

How are SQL queries distributed across the stack?



# DuckDB Extension

What does the DuckDB Extension offer?

Implements query distribution client

Implements remote execution client

Integrates with Monostore

Integrates with Lakehouse

Logs queries on Data Lake

Adds supports for [CURL](#)

Adds support for [Lance](#) file format

# Monostore

What is the configuration of a large Monostore?

p4de.24xlarge

96 vCPUs

1,152 GiB of RAM

8 × 1 TB NVMe SSD

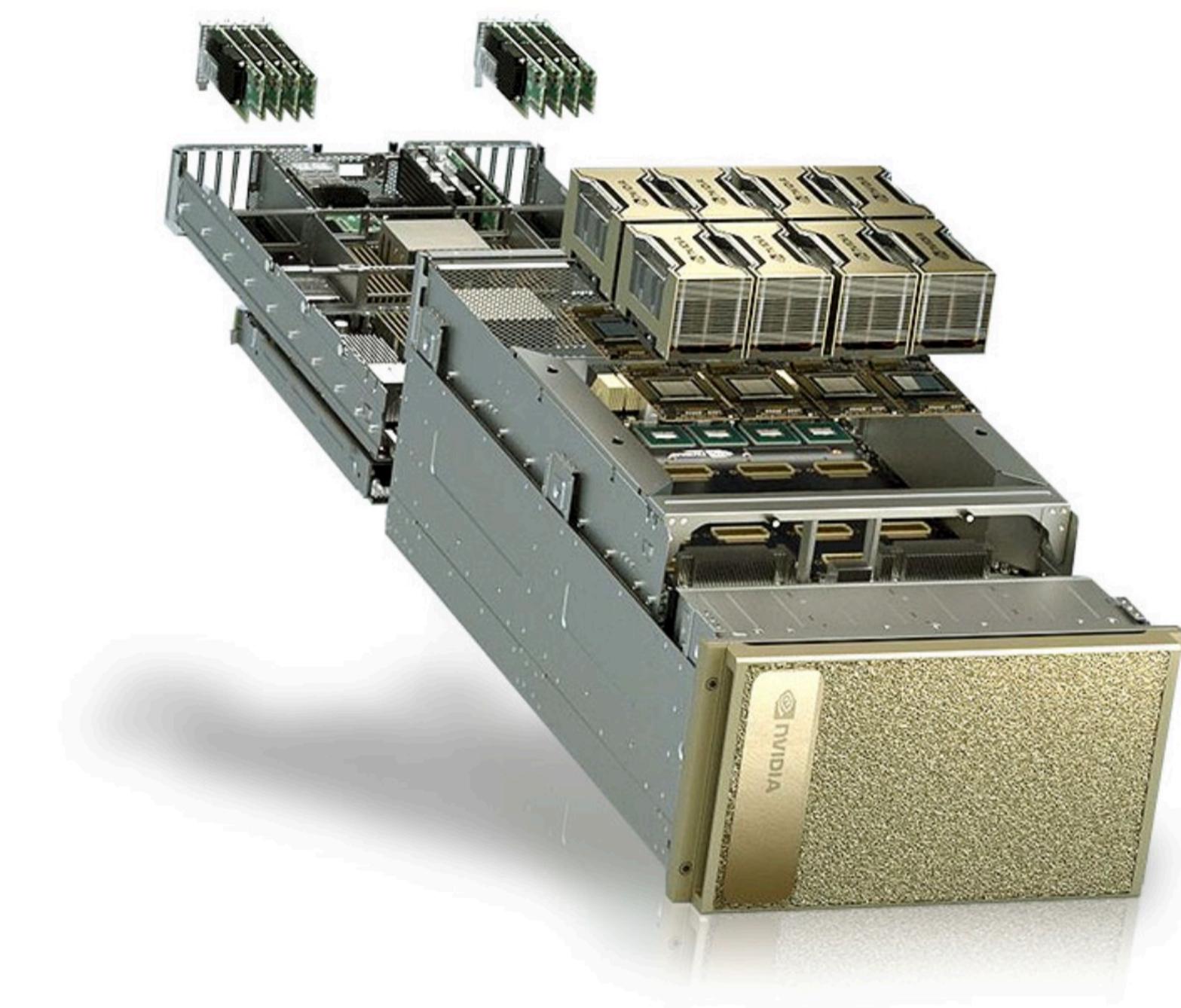
8 × NVIDIA A100 GPUs

640 GB of HBM2e GPU RAM

400 Gbps ENA and EFA bandwidth

2 seconds to fill 80% of RAM with data

\$40.96/hour (on-demand)



# λ Serverless Function

What is the serverless function made of?

Bun JavaScript runtime

DuckDB with Node.js API

PuffinDB Extension

Wasmtime WASM runtime

Distributed Query Planner

Distributed Query Engine

# Cloud Data

What makes Cloud Data so special?

**Cloud Data** is heavy and its center of gravity is the lakehouse

**Cloud Data** is collaborative (multiple users edit data & metadata)

**Cloud Data** is real-time (updated with submillisecond latency)

**Cloud Data** is small or large (from megabytes to petabytes)

**Cloud Data** is agnostic (it can be manipulated by any application)

# Persistence

How are persistence and statefulness implemented?

Table Summaries		Object Store (S3)	
Table Metadata		Lakehouse Catalog (powered by Dynamo DB)	
Sharings		Key-Value Store (DynamoDB)	
Edits		+	Block Store (EBS) then Object Store (S3)
Persistent Logs		+	Block Store (EBS) then Object Store (S3)
Transient Logs		Transient Metadata Store (Redis)	
Cache Metadata		Transient Metadata Store (Redis)	

# Rationale

## Why do we need a new distributed SQL database?

Cloud-native architecture

Designed for virtual private cloud deployment

Designed for small to large datasets

Designed for real-time analytics

Designed for interactive analytics

Designed for transformation and analytics

Designed for analytics and transactions

Optimized for machine-generated queries

Designed for next-generation query engines

Designed for next-generation file formats

Designed for lakehouses

Designed for data mesh integration

Designed for all users

Designed for large groups of users

Designed for extensibility

Designed for embedability

# Statistics

How are table and partition statistics captured?

## statistics.parquet

One column per summary statistic (min, max, mean, etc.)

One row per column of the related table

Ordered as columns are ordered in the related table

Optimized for storing both simple and complex summary statistics

## frequencies.parquet

Columns for column, value, and frequency

One row per pair of column-value

Ordered by column and decreasing frequency

Optimized for columns with large numbers of distinct values

## histograms.parquet

One column per column of the related table

One row per bin (1,000 to 10,000 bins)

Ordered by increasing bin minimum value

Optimized for storing high-resolution histograms

# Lakehouse

## Why should we build for the Lakehouse?

Schema evolution supports add, drop, update, or rename, and has no side-effects.

Hidden partitioning prevents user mistakes that cause silently incorrect results or slow queries.

Partition layout evolution can update the layout of a table as data volume or query patterns change.

Time travel enables reproducible queries that use exactly the same table snapshot.

Table version rollback allows users to quickly correct problems by resetting tables to a good state.

Advanced filtering prunes data files with partition and column-level statistics, using table metadata.

Serializable isolation makes table changes atomic and ensures that readers never see partial changes.

Multiple concurrent writers use optimistic concurrency and transaction retry.

# Query Planner

How are we thinking about the Query Planner?

Logical query optimization using DuckDB's optimizer

Implementation of [multi-relational algebra](#) (to be confirmed)

Domain Specific Language (DSL) for [rule-based query optimization](#)

Rule scripting powered by [TypeScript](#) for dynamic rule injection and client-side + cloud-side execution

Initial set of optimizer rules bootstrapped by porting [Trino's rules](#) from Java to DSL

Automatic generation of optimizer rules using [WeTune](#) (to be confirmed)

Dynamic injection of optimizer rules through standard [SQL API](#)

Rule interpreter implemented in Rust

Memoization for [cost-based optimization](#)

Parallelization of query planning across multiple [serverless functions](#)

Parallelization of [metadata lookups](#) through concurrent invocations of the query engine

Dynamic cascaded replanning at the edges



What does HTAP mean for PuffinDB?

HTAP systems are either OLTP-focused or OLAP-focused

PuffinDB is resolutely OLAP-focused (for now)

Low-volume/high-latency transactions are handled by Athena

High-volume/low-latency transactions are handled by EBS

Things will improve once the Object Store handles updates in place

# ⚙️ Plan Execution

How are query plans executed?

1. Query translated from non-SQL dialect (e.g. PRQL, Malloy) to SQL
2. Abstract syntax tree, relational tree, and logical query plan produced by DuckDB
3. Non-distributed logical query plan optimized by DuckDB
4. Non-distributed logical query plan further optimized by WeTune (to be confirmed)
5. Set of Object Store partitions looked-up from Lakehouse (using Iceberg Java API)
6. Set of cached partitions looked-up from Transient Metadata Store (powered by Redis)
7. Distributed logical query plan generated by Query Planner
8. Distributed physical query plan mapped to serverless functions and Monostore
9. Distributed physical query plan executed by Distributed Query Engine

# Languages

Which languages are used and for what?

C++ for most parts of the DuckDB Extension

TypeScript for most of the middleware code running on [functions](#) and [Monostore](#)

SMT-LIB for the formal models of the [Distributed Query Planner](#)

Rust for the routines of the [Distributed Query Planner](#)

Java for the [connectors](#) to databases (to be confirmed)

Python for the [connectors](#) to applications

HCL for the [Terraform](#) templates

# Python Runtime

How is Python supported and why do we need it?

Powered by [CPython](#) (to be confirmed)

Deployed on [Monostore](#) only (initially)

Offering runtime for [Airbyte connectors](#)

Offering runtime for [SQLGlot transpiler](#)

Offering Python interface to [GPU](#)

# File Formats

Which data file formats will PuffinDB support?

All file formats natively supported by DuckDB (CSV, JSON, Parquet, ...)

Native DuckDB file format (to be stabilized by year's end)

Lance file format (for 100× to 1000× faster random access)

Upcoming file format to be released by Meta

Future replacement for Parquet

# ⚡ Connector Framework

How do we connect to datasources?

Arrow natively supported by DuckDB

CURL support implemented in DuckDB Extension

Airbyte Protocol supported by Monostore

Airbyte Connectors deployed on Monostore

AWS Glue supported by functions and Monostore

# Remote Execution

Which syntax should we use for remote execution?

## Option 1: THROUGH

Proposed by PuffinDB

```
SELECT * THROUGH '127.0.0.1' FROM remoteTable;
```

## Option 2: remote

Implemented by ClickHouse

```
SELECT * FROM remote('127.0.0.1', view(SELECT * FROM remoteTable));
```

# NAT Hole Punching

How can we improve network performance?

Based on [TCPunch library from SPCL](#)

Allows fast Lambda-to-Lambda communication

Increases bandwidth and reduces latency

Requires Lambda registration with [Monostore](#)

Supports up to 1,024 concurrent registrations



How can the project take advantage of AI?

Can we use GPT-4 to convert text to SQL?

Can we use GPT-4 to convert text to Malloy?

Can we use GPT-4 to translate Trino's rules?

How can we use AI or to optimize the Query Planner?

What can we learn from the OtterTune optimizer?

# ⌚ Future Proofing

Serverless functions becoming stateful officially

Serverless functions provisioned by the 10,000's

Serverless containers starting in 15s instead of 60s

Serverless runtimes with 4 times more CPU & RAM

Serverless runtimes with 4 times BW/RAM ratio

Serverless runtimes with GPU acceleration

Serverless runtimes with superchip acceleration

Serverless Redis cluster

What should we get within two or three years?

SSD tier for Object Store

Memory tier for Object Store

Parquet export when filtering Object Store objects

DuckDB running directly within the Object Store

Updates in place for Object Store objects

Full SQL acceleration on GPU

SQL superchips

Similar performance levels across AWS and Azure

# 📍 Roadmap

How can we release early and often?

## Q1 Preliminary DuckDB Extension and Lambda Function

With limited subset of planned features

## Q2 PuffinDB 0.1 with basic Distributed Query Planner

Filter and compute pushdown

## Q3 PuffinDB 0.5 with Connector Framework (powered by Airbyte)

Including support for PRQL and Malloy

## Q4 PuffinDB 1.0 with advanced Distributed Query Planner

Including support for full TPC-H benchmark

## 2024 PuffinDB 2.0

Including support for full TPC-DS benchmark

# #[ Reactiva Caching

How are tables cached throughout the stack?

1. Serverless functions (using early interruption and probabilistic replication)
2. Monostore's Solid State Drive (compressed or uncompressed)
3. Monostore's CPU RAM (compressed or uncompressed)
4. Monostore's GPU RAM (if GPU available, uncompressed)
5. Client's Solid State Drive (compressed or uncompressed)
6. Client's CPU RAM (compressed or uncompressed)
7. Client's GPU RAM (if GPU available, uncompressed)

# Thank you!

puffindb.io

