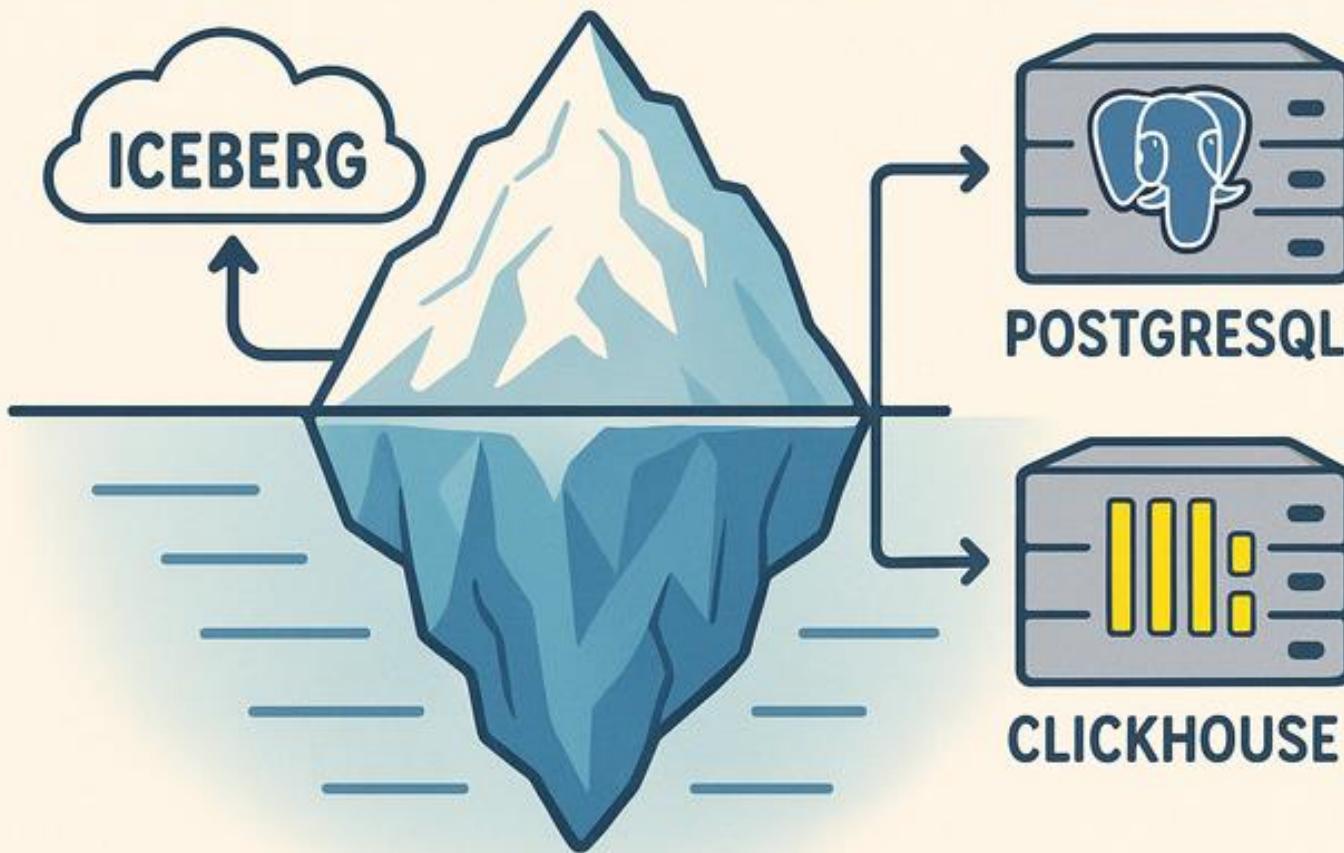


## RUNNING ICEBERG ON EXISTING DBS



# ABOUT US



Shivji Kumar Jha  
Staff Engineer  
Data Platforms, Nutanix

Saurabh Ojha  
Software Engineer  
Open-source hacker  
Data Platforms, Nutanix



- Areas of Interest
  - Distributed Storage Infrastructure
  - Database & Streaming Internals
  - Application Architectures
- Passion for OSS DBs and Communities
  - Contributed to MySQL and Apache Pulsar
  - Exploring Postgres, Clickhouse & iceberg\*
- 25+ talks at conferences & meetups
  - Slides: [github.com/shiv4289/shiv-tech-talks/](https://github.com/shiv4289/shiv-tech-talks/)
  - Recordings: <https://tinyurl.com/shiv-talks>
- Text here: [linkedin.com/in/shiyijha/](https://linkedin.com/in/shiyijha/)

- Areas of Interest
  - Databases Internals
  - Streaming Internals
  - Linux kernel Features
- Open-Source Enthusiast
- Mostly active here:
  - [linkedin.com/in/ojhasaurabh2099/](https://linkedin.com/in/ojhasaurabh2099/)

# Safe Harbor Statement

The views, opinions, and conclusions presented in these slides are solely my own and do not reflect the official stance, policies, or perspectives of my employer or any affiliated organization. Any statements made are based on my personal experiences and research and should not be interpreted as official guidance or endorsement.

# Show of hands

- Heard of iceberg?
  - *Another open table format?*
- Used Iceberg?
  - *Learning? Dev? Prod?*
- What query engine are you most familiar with?
  - ?
- Clickhouse? Postgres?
- Open-source contributions?

# Agenda

- Iceberg's Appeal & Adoption
- The ever-growing adoption
  - *Even your old boring open-source Databases*
- Iceberg and the OLAP Databases ft. Clickhouse
- Iceberg and the OLTP Databases ft. Postgres
- Demos (if time permits... hopefully ☺ )
- Takeaway:
  - *Great time to be a builder*
  - *i.e., if you love hacking around open source*

# ICEBERG'S APPEAL AND ADOPTION

# Setting the Context

- Open table formats like Iceberg are becoming foundational in modern data lakes
- Traditionally used with engines like Spark, Trino, Dremio etc
- Even stream-native platforms like Kafka (TableFlow), Pulsar (Ursa), and RisingWave are adopting Iceberg
- AWS S3 Tables saw a lot eyeballs
- Adoption signals a broader movement towards modular, decoupled lakehouse architectures
- Even your existing, old and boring databases are embracing it

# Iceberg's Appeal

- Works well with object stores (e.g., S3, GCS)
- Open format, less lock-in
- Choice of query engine (More options, less locking of your data... the dreaded migrations!)
  - Got a *data migration story?*
- Decouples storage and compute
- Schema evolution, time travel, partition pruning, ACID
- Increasing support across the ecosystem
- A lot of mindshare at the moment!

# Deconstructing a database

## ■ Storage

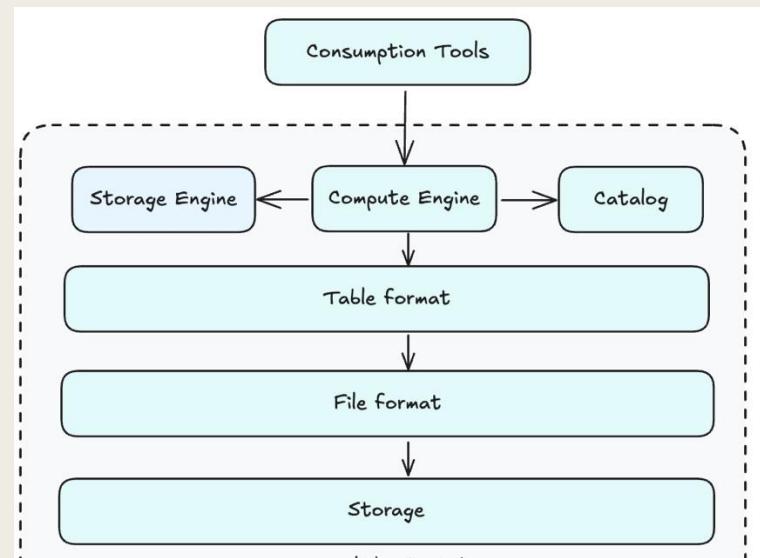
- Store large datasets
- Local filesystem, attached storage, distributed file system, Object storage
- Row vs column-oriented store

## ■ File format

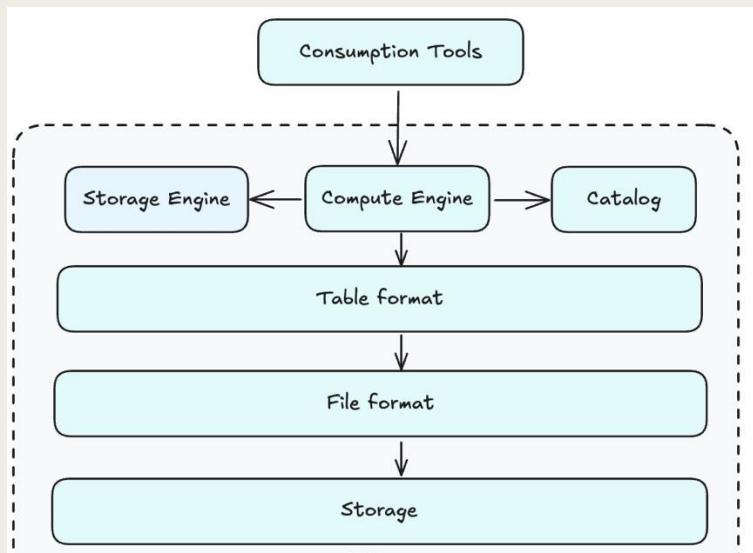
- Impacts compression, data structure, performance.
- Structured(CSV), semi-structured(JSON), unstructured(text file)
- Row-oriented(CSV, Apache Avro), Column Oriented(Parquet, ORC)

## ■ Table format

- Metadata: specifies how datafiles should be laid out in storage
- Abstract the complexity of physical data structure
- Abstract with DMLs (insert, update, delete) schema evolution etc
- Safe DML access with atomicity, consistency etc



# Deconstructing a database



- Table format

- *Metadata: specifies how datafiles should be laid out in storage*
  - *Abstract the complexity of physical data structure*
  - *Abstract with DMLs (insert, update, delete) schema evolution etc*
  - *Safe DML access with atomicity, consistency etc*

- Storage Engine

- *Layout data in the form specified by table format*
  - *Keep files and data structures up-to-date with new data*
  - *Physical optimization of data*
  - *Index maintenance*
  - *Expiry / delete old data*

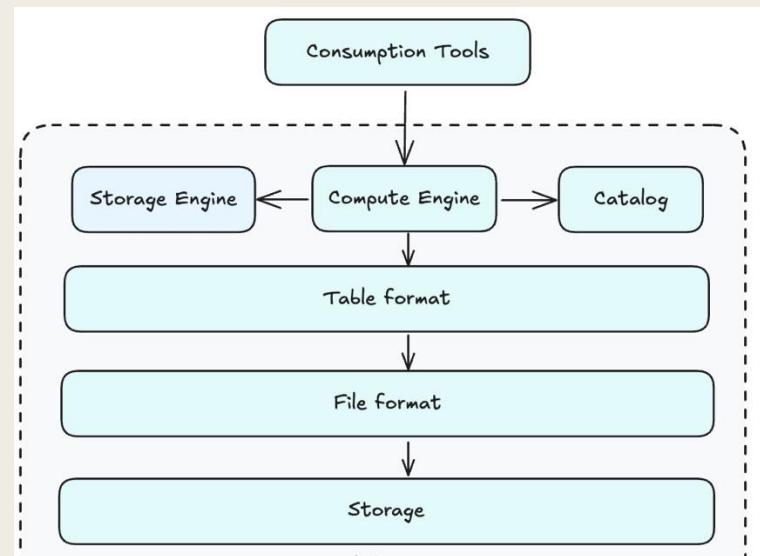
# Deconstructing a database

## ■ Storage Engine

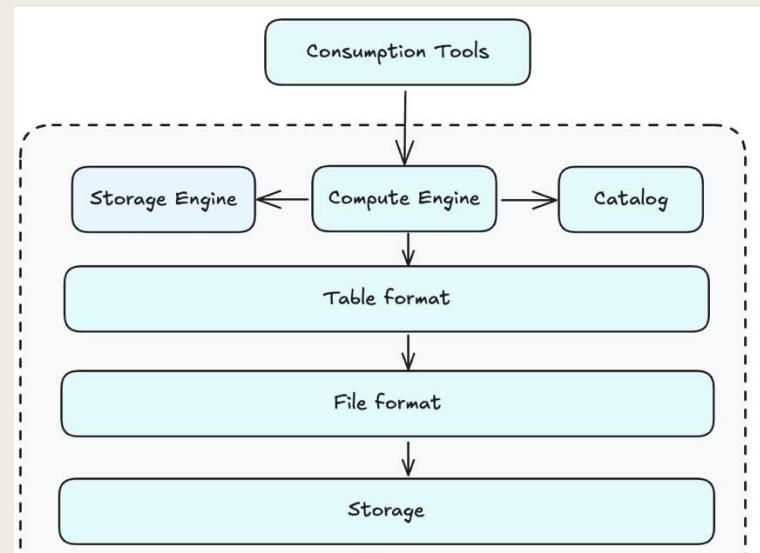
- Layout data in the form specified by table format
- Keep files and data structures up-to-date with new data
- Physical optimization of data
- Index maintenance
- Expiry / delete old data

## ■ Catalog

- Identify dataset needed for analysis using metadata (from table format)
- Users / compute engines go to catalog for existence of a table
  - Get table name, table schema, location of data on storage
- Traditionally, internal to system and only engine can access
- Hive: one of the first systems open for any system to use
- Different from metadata catalog!



# Deconstructing a database



## ■ Catalog

- Identify dataset needed for analysis using metadata (from table format)
- Users / compute engines go to catalog for existence of a table
  - Get table name, table schema, location of data on storage
- Traditionally, internal to system and only engine can access
- Hive: one of the first systems open for any system to use
- Different from metadata catalog!

## ■ Compute Engine

- Run user workloads to process data
- Choose as per data size, compute load, type of workload etc
- Scale? Use distributed compute engines
  - Spark, Dremio, e6Data

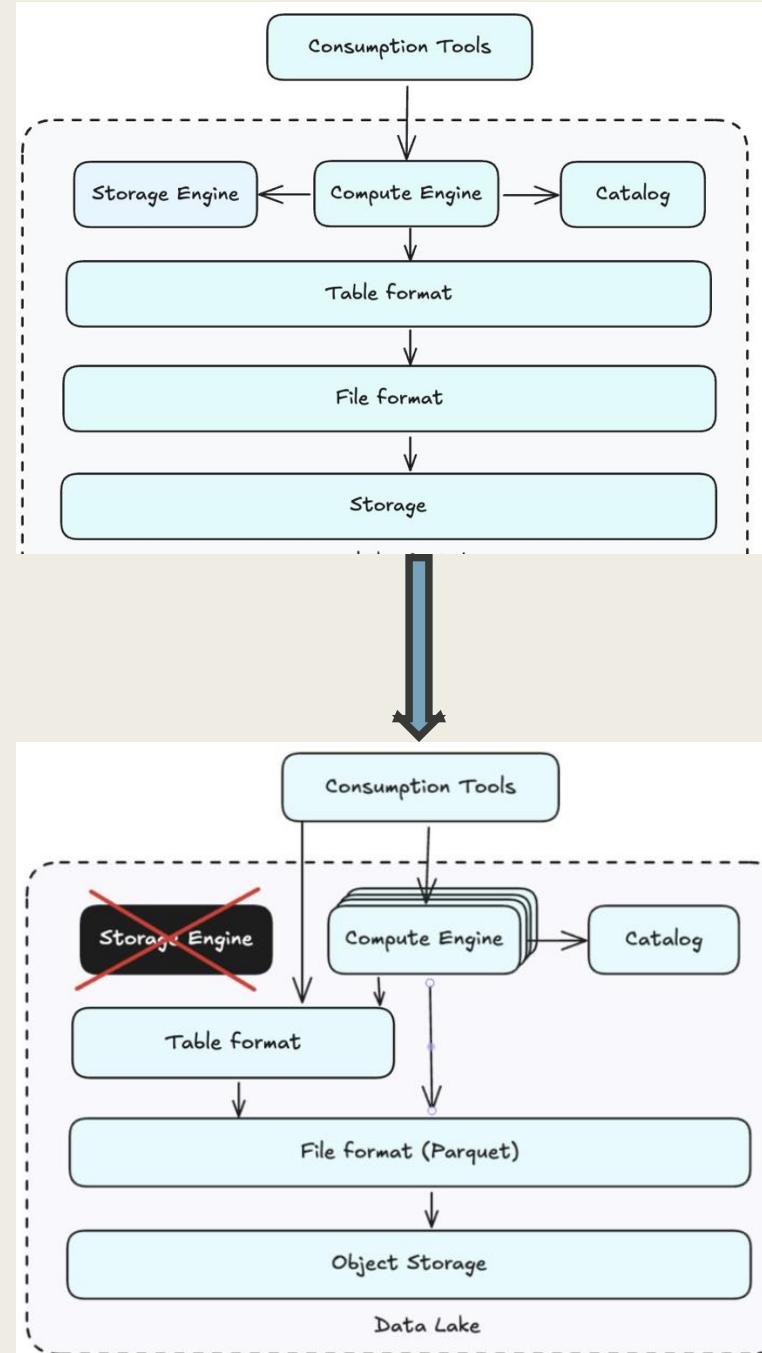
# Warehouse & Lake

## ✓ Pros of a Data Lake

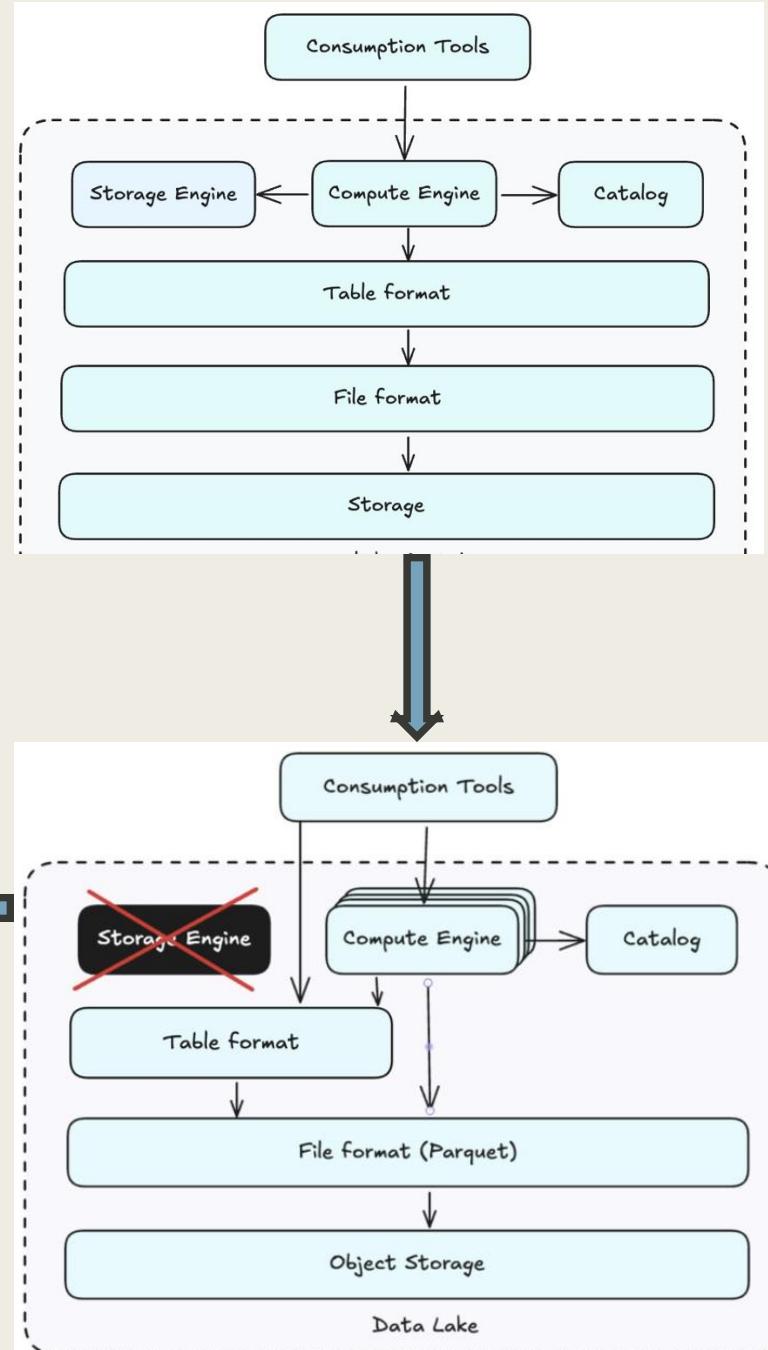
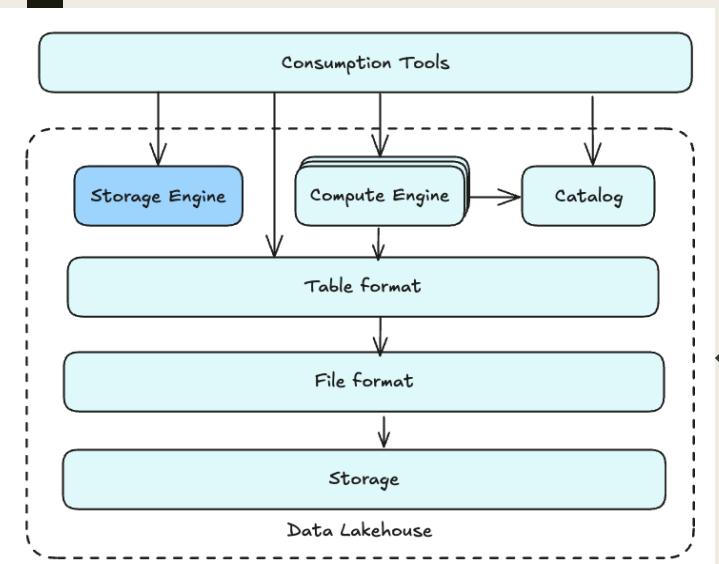
- Lower cost
- Stores data in open formats
- Handles unstructured data
- Supports ML use cases

## ⚠ Cons of a Data Lake

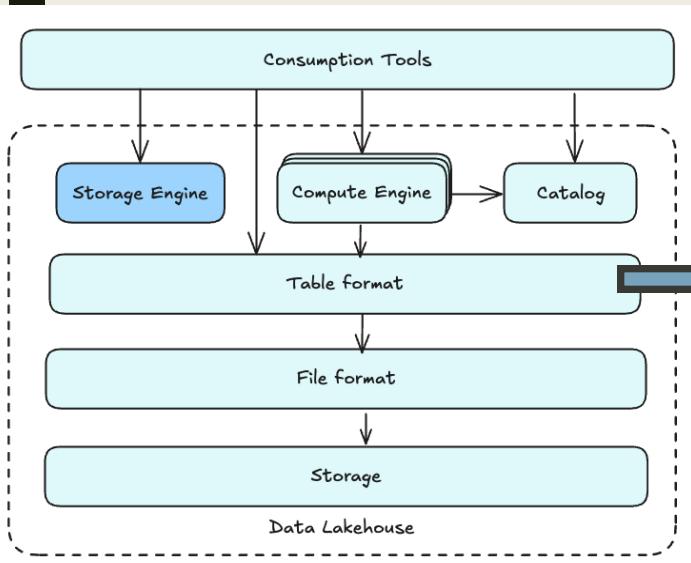
- Performance
- Lack of ACID guarantees
- Lots of configuration required



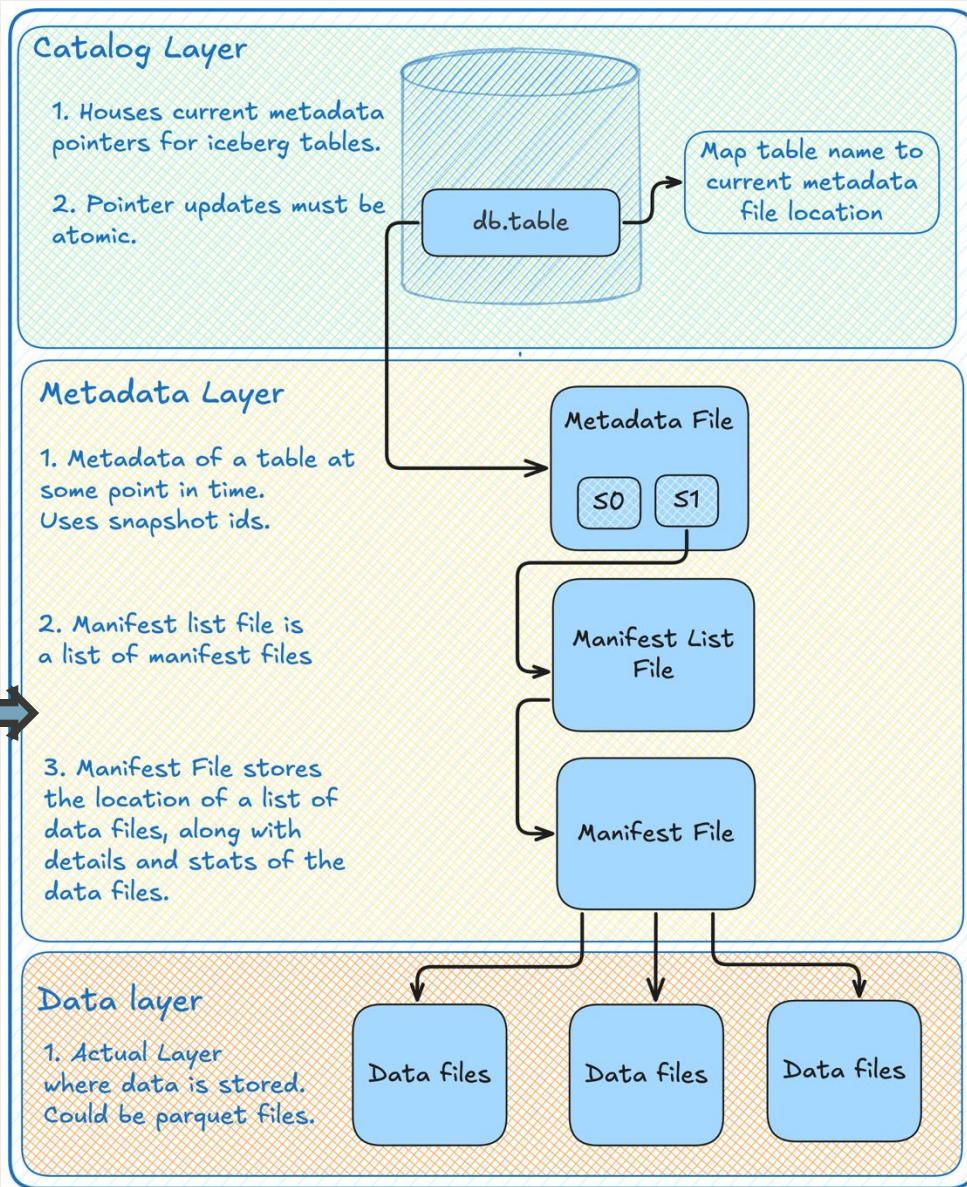
# Data Lake to Lakehouse



# Iceberg Table Format



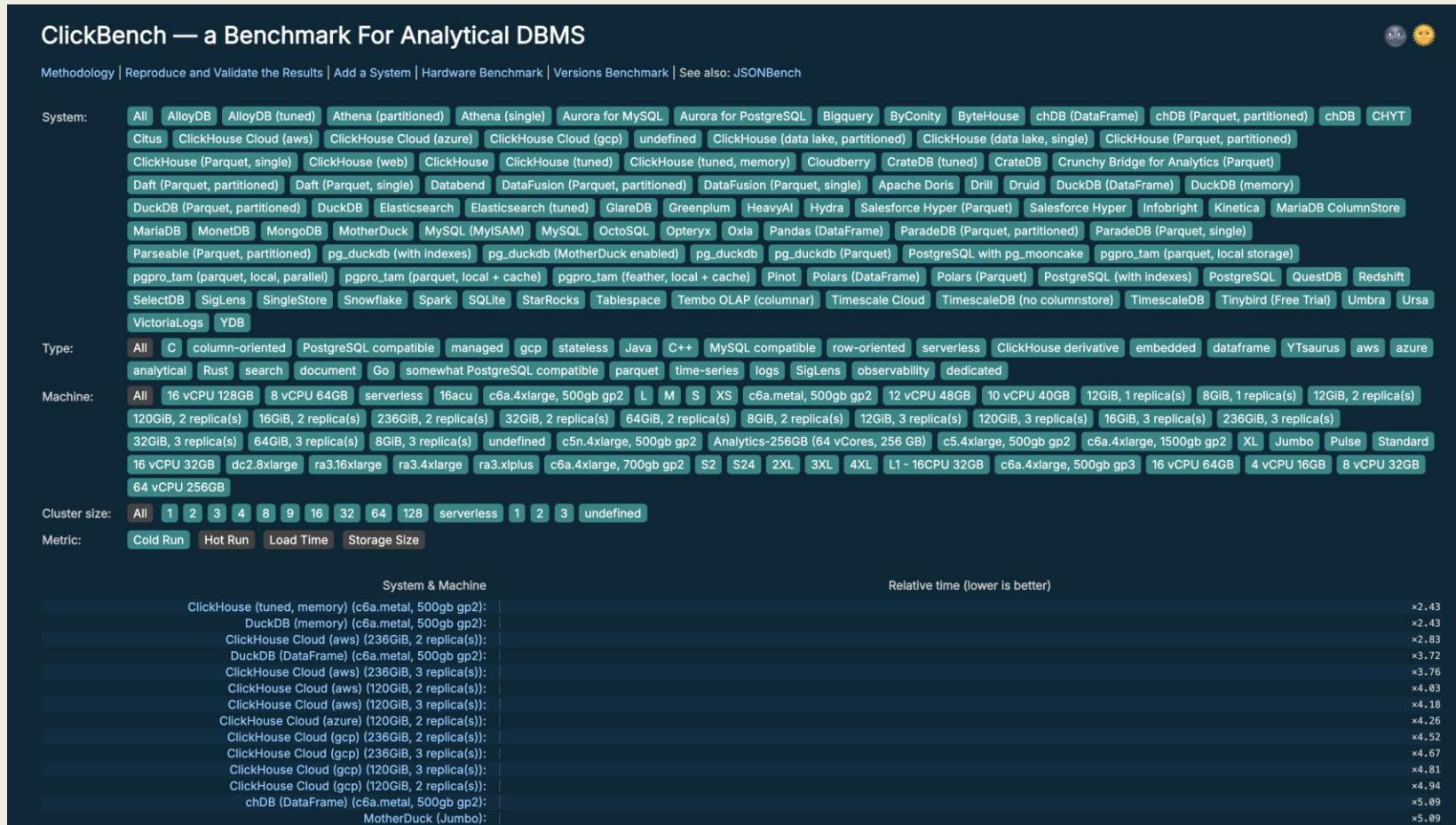
– Apache Iceberg the definitive guide (OREILLY)



CLICKHOUSE &  
ICEBERG

# Introducing ClickHouse!

- Open source\* OLAP database known for speed



# Introducing ClickHouse!

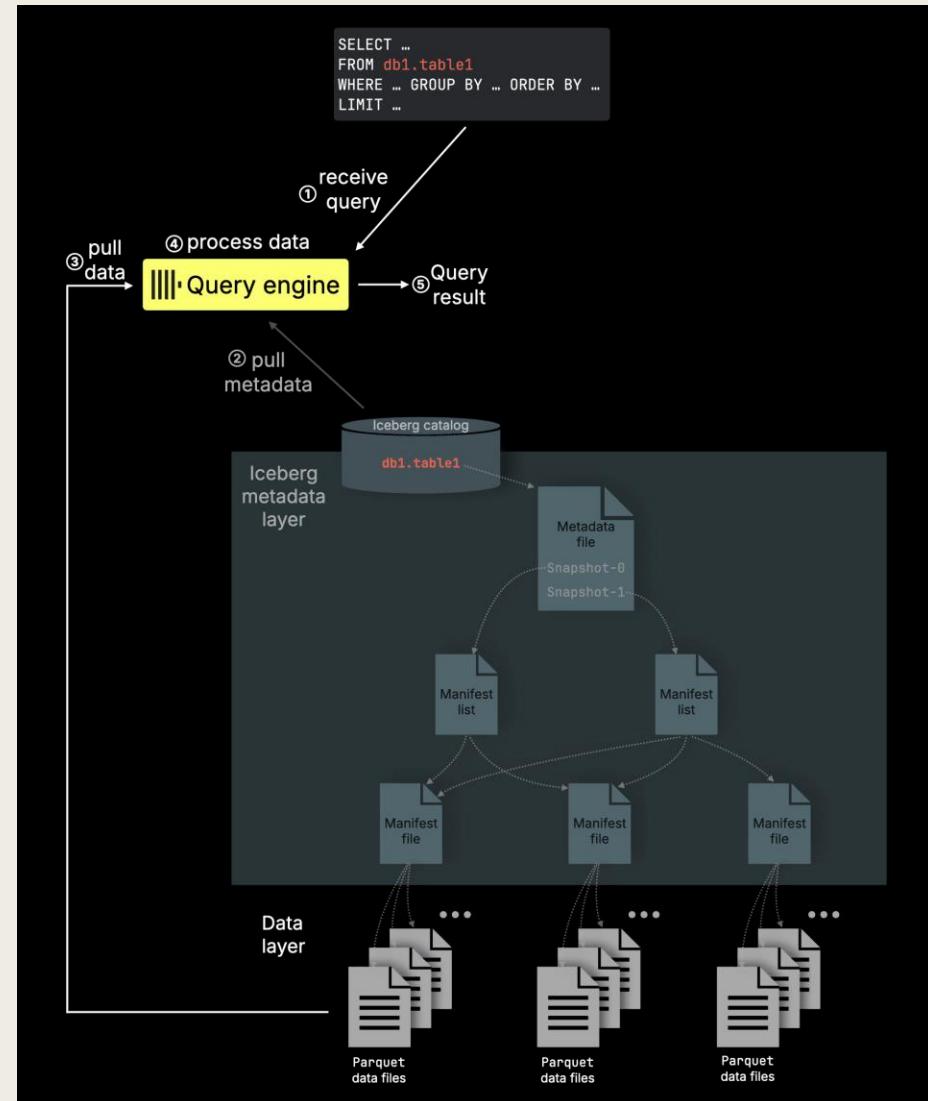
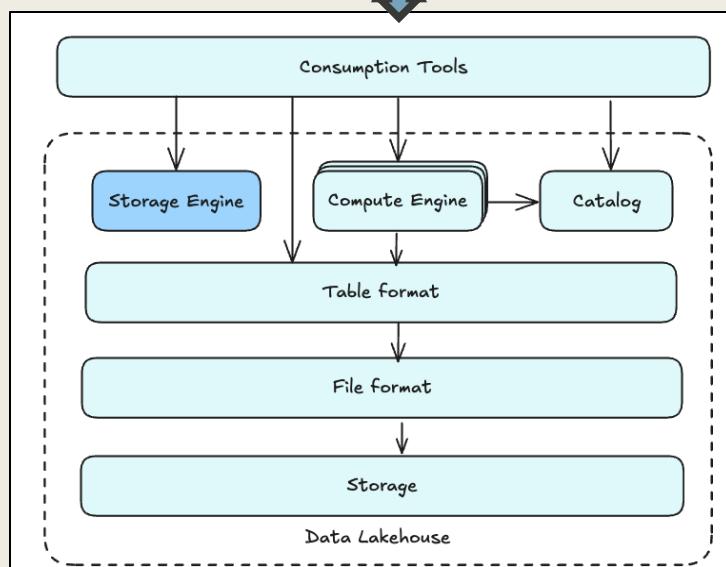
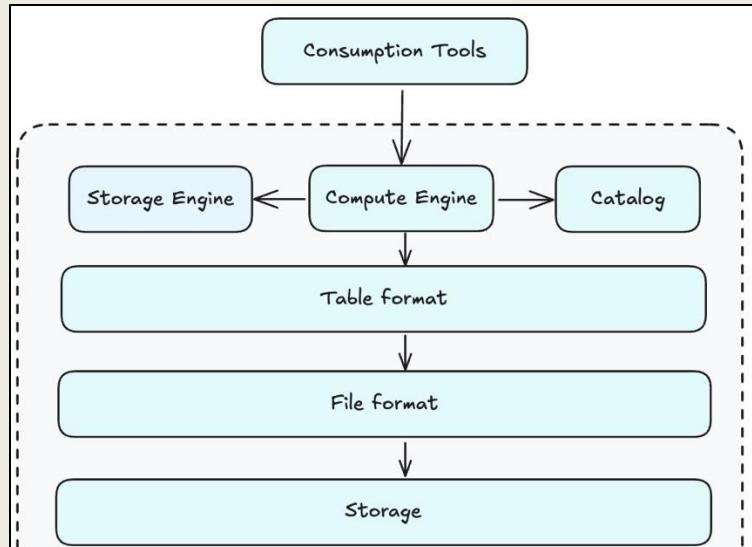
- Open source\* OLAP database known for speed (see ClickBench!)
- Very easy to get started
  - *one binary, regular SQL/Tables*
  - *Very popular, open, well adopted!*

The screenshot shows the GitHub repository page for ClickHouse. At the top, there are navigation links for Issues (4k), Pull requests (532), Discussions, Actions, Projects, Wiki, Security (7), and Insights. Below the header, the repository name "ClickHouse" is shown as public. There are buttons for Watch (690), Fork (7.3k), and Starred (40.5k). The main area displays a timeline of recent commits. A commit by "bharatnc" is highlighted, showing a merge pull request from ClickHouse/zoestenkamp-pa... into master. Other visible commits include merges from Blargian/autogen\_setting..., ClickHouse/prepare-for..., ClickHouse/ci\_compress..., and ClickHouse/lakehouse. On the right side, there's an "About" section with a brief description of ClickHouse as a real-time analytics database management system. Below this, there are links to clickhouse.com and various technology tags: rust, embedded, sql, database, big-data, ai, analytics, cpp, clickhouse, dbms, self-hosted, distributed, olap, cloud-native, mpp, hacktoberfest, and lakehouse. At the bottom, there are links for Readme and Apache-2.0 license.

# Introducing ClickHouse!

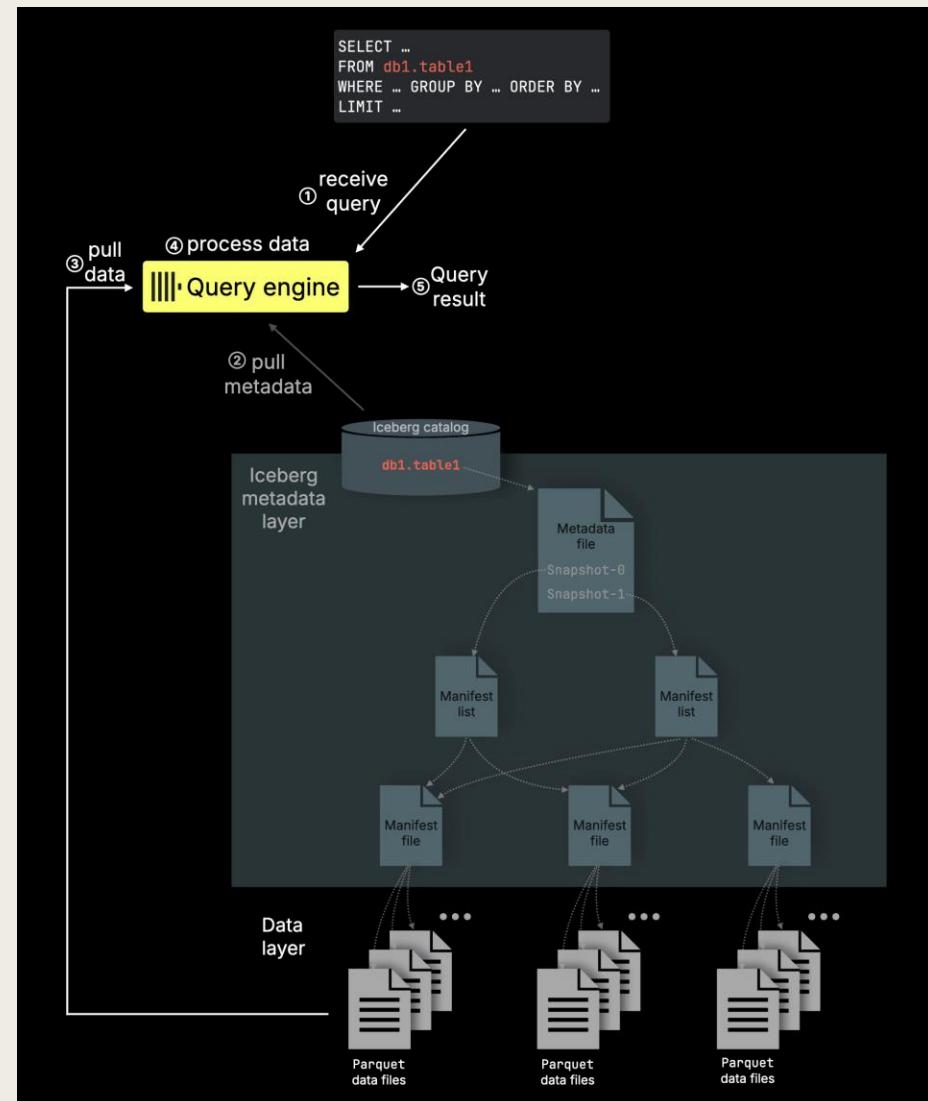
- Open source\* OLAP database known for speed
- Very easy to get started
  - *one binary, regular SQL/Tables like RDBMS*
  - *Very popular, open, well adopted!*
- Note: Very different from an RDBMS (row vs column, oltp vs olap etc)
- Traditionally, loved big fat servers (vertical scaling...)
  - *Now shards & replicas too.*
- Storage Compute segregation(?)
  - *Possible with Open source, not trivial!*
- But a lot of integrations (engines, formatReaders etc)

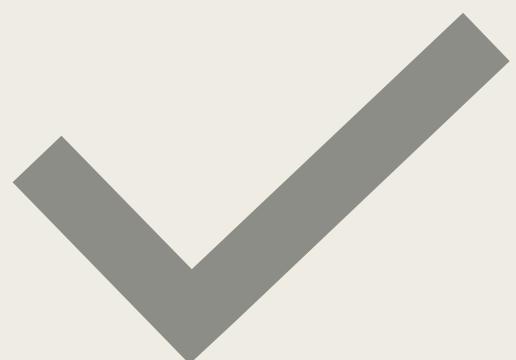
# Query Engine for lakehouse



# Query Engine Requirements

- Read data in object store
  - *Maybe cache too?*
  - *At least an option!*
- Query Parquet
  - *join with other data sources?*
- Data structures for meta files
  - *Caching too maybe?*
- Support Lakehouse features
  - *Catalog integrations*
  - *Time travel,*
  - *Partition Pruning*
  - *Schema Evolution* etc





# INTEGRATIONS IN CLICKHOUSE

# Query file formats

ClickHouse supports 70+ file formats out of the box"

- Parquet
- JSON
- CSV
- Apache Arrow
- ORC
- CSV
- XML
- ... and more
- No ingestion required
  - *Direct SQL queries with dictionaries, joins, window functions, 160+ aggregations*

Query: "undefined" **Save as my query**

```
1 SELECT *  
2 FROM system.formats  
3 WHERE is_input  
4 ORDER BY name;
```

▶ Run

(Ctrl/Cmd + Enter)

## Stats

Query time: 0.002s  
Rows read: 105

## Results

## Charts

#	name
1	
40	Native
41	Npy
42	ORC
43	One
44	Parquet
45	ParquetMetadata
46	Protobuf
47	ProtobufList
48	ProtobufSingle

# Querying Foreign File Formats

The screenshot shows a code editor with the `IInputFormat.h` header file open. The code defines a class `IInputFormat` with various methods like `generate`, `read`, and `resetParser`. A tooltip titled "Derived classes of 'IInputFormat'" lists many subclasses, with `ParquetBlockInputFormat` highlighted.

```
namespace DB

/** Input format is a source, that reads data from ReadBuffer.
 */
class IInputFormat : public SourceWithKeyCondition
{
protected:
    ReadBuffer * in [[maybe_unused]] = nullptr;

public:
    /// ReadBuffer can be null
    IInputFormat(Block header, ReadBuffer & in);

    Chunk generate() override;

    /// All data reading from
    virtual Chunk read() = 0;

    /** In some usecase (hello
     * The recreating of parse
     * resetParser() which all
     * source stream without r
     * That should be called a
     */
    virtual void resetParser();

    virtual void setReadBuffer(ReadBuffer & in);
    virtual void resetReadBuffer() { in = nullptr; }

    virtual const BlockMissingValues * getMissingValues() const { return nullptr; }

    // Must be called from ParallelParsingInputFormat after readSuffix
    ColumnMappingPtr readColumnMapping();
};
```

Derived classes of 'IInputFormat'

- NativeORCBlockInputFormat (in DB)
- OneInputFormat (in DB)
- ValuesBlockInputFormat (in DB)
- ORCBlockInputFormat (in DB)
- ParquetBlockInputFormat (in DB)**
- ParquetMetadataInputFormat (in DB)
- IRowInputFormat (in DB)
- ParallelParsingInputFormat (in DB)
- ArrowBlockInputFormat (in DB)
- RawBLOBRowInputFormat (in DB)
- NpyRowInputFormat (in DB)
- RowInputFormatWithDiagnosticInfo (in DB)
- LineAsStringRowInputFormat (in DB)
- AvroConfluentRowInputFormat (in DB)
- AvroRowInputFormat (in DB)
- JSONEachRowRowInputFormat (in DB)
- MsgPackRowInputFormat (in DB)
- RegexpRowInputFormat (in DB)
- RowInputFormatWithNamesAndTypes<FormatReaderImpl> (in DB)

# Parquet Reader

The screenshot shows a code editor with two tabs open: `ParquetBlockInputFormat.h` and `ParquetBlockInputFormat.cpp`. The `ParquetBlockInputFormat.h` tab is active, displaying the class definition:

```
#if USE_PARQUET
namespace DB
{
    class ParquetBlockInputFormat : public IInputFormat
    {
        public:
            git(
                ReadBuffer & buf,
                const Block & header,
                const FormatSettings & format_settings,
                size_t max_decoding_threads,
                size_t max_io_threads,
                size_t min_bytes_for_seek);

            ~ParquetBlockInputFormat() override;

            void resetParser() override;

            String getName() const override { return "ParquetBlockInputFormat"; }

            const BlockMissingValues * getMissingValues() const override;

            size_t getApproxBytesReadForChunk() const override { return prev... }

        private:
            Chunk read() override;
    };
}
```

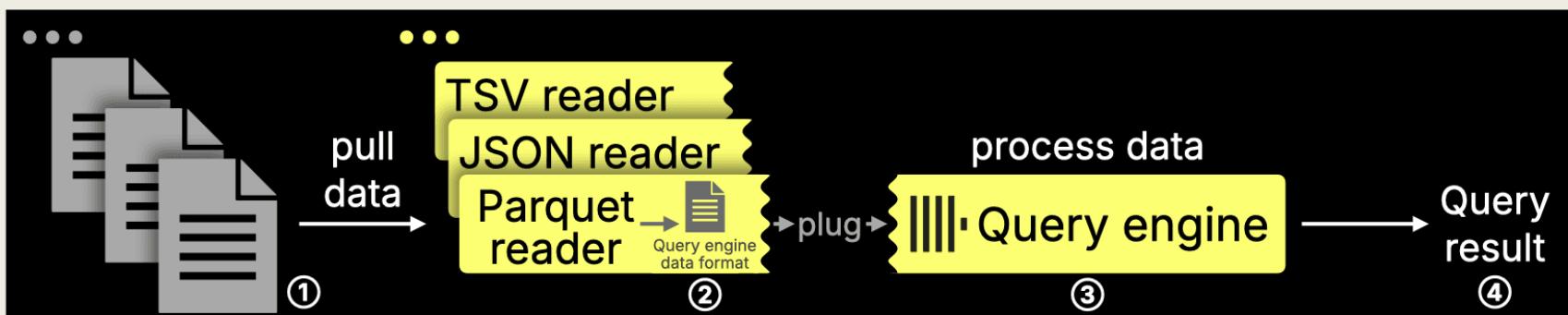
A yellow arrow points from the `ParquetBlockInputFormat` class name in the code to the same class name in the derived classes list. The derived classes list is titled "Derived classes of 'IInputFormat'" and includes the following items:

- NativeORCBlockInputFormat (in DB)
- OneInputFormat (in DB)
- ValuesBlockInputFormat (in DB)
- ORCBlockInputFormat (in DB)
- ParquetBlockInputFormat (in DB) **(highlighted)**
- ParquetMetadataInputFormat (in DB)
- IRowInputFormat (in DB)
- ParallelParsingInputFormat (in DB)
- ArrowBlockInputFormat (in DB)
- RawBLOBRowInputFormat (in DB)
- NpyRowInputFormat (in DB)
- RowInputFormatWithDiagnosticInfo (in DB)
- LineAsStringRowInputFormat (in DB)
- AvroConfluentRowInputFormat (in DB)
- AvroRowInputFormat (in DB)
- JSONEachRowRowInputFormat (in DB)
- MsgPackRowInputFormat (in DB)
- RegexpRowInputFormat (in DB)
- RowInputFormatWithNamesAndTypes<FormatReaderImpl> (in DB)

The `ParquetBlockInputFormat` class definition in `ParquetBlockInputFormat.cpp` is partially visible at the bottom of the screen.

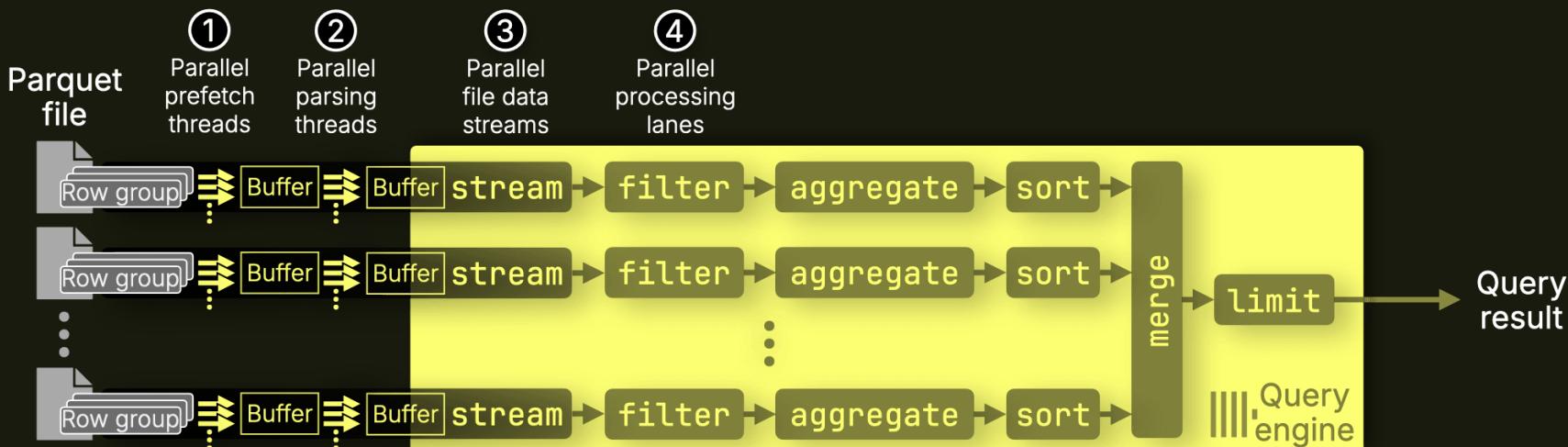
# Reading non-native files

- 1: Format Reader reads and parses external files (eg., Parquet, CSV, JSON...)
- 2: Data converted into ClickHouse's internal columnar in-memory format
- 3: Query engine runs SQL operations (filter, join, aggregate) on in-memory data
- 4: Produces result set for client / downstream pipeline



# Performance with ParquetReader

- 1: Within a Parquet file, the Parquet file reader reads multiple row groups in parallel
- 2: Parsing threads read and parse data from multiple row groups within the same file in parallel
  - *If prefetching is active, they read from the prefetch buffer; otherwise, they read directly from the file*
- 3: Different Parquet files are processed concurrently, each with their own parallel prefetch and parsing threads, to maximize throughput across files
- 4: filtering, aggregation, and sorting happen across independent lanes for maximum concurrency



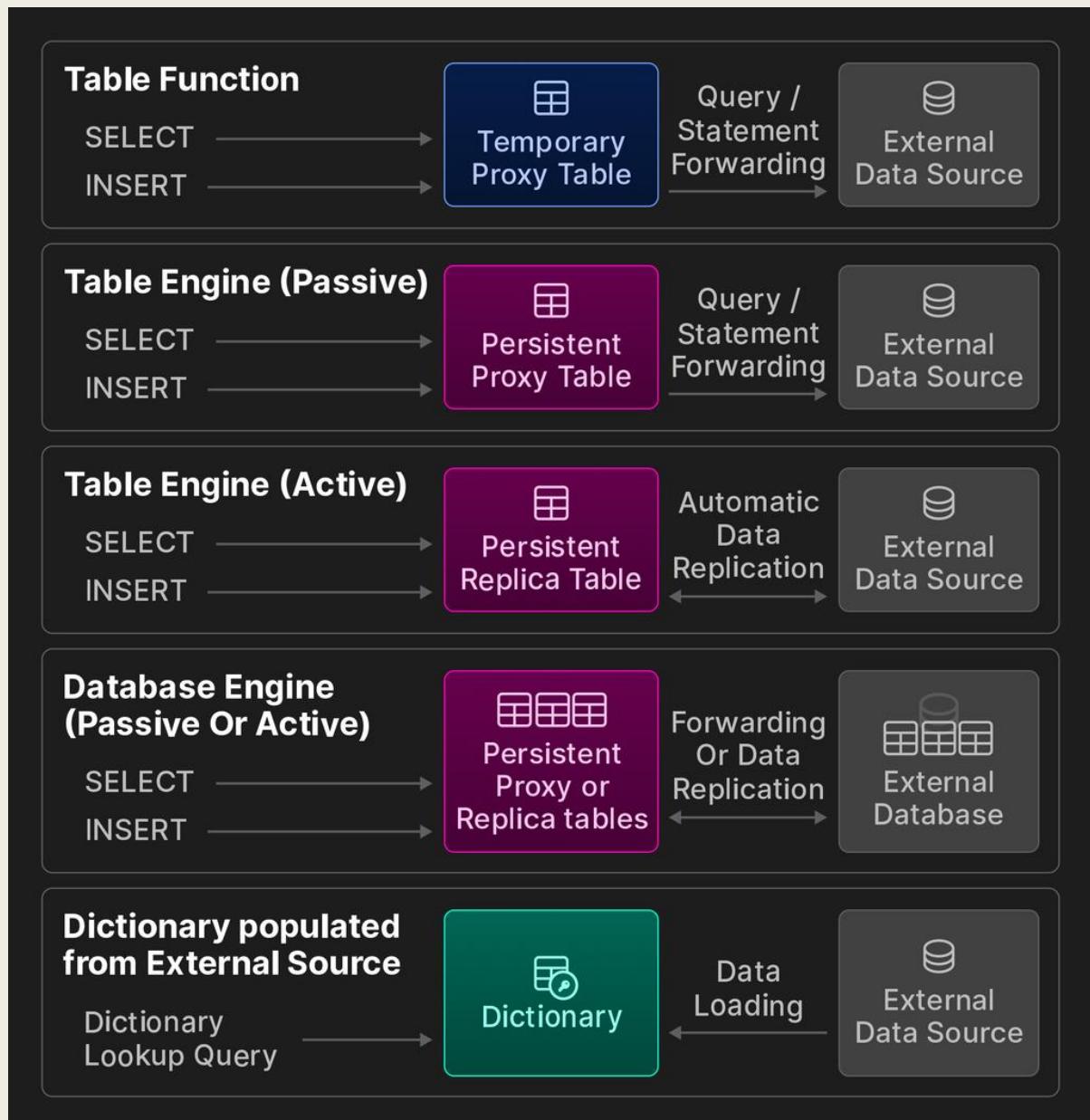
- Awesome that it is open & interoperable though! More room to improve performance
  - Brand new parquet reader loading at <https://github.com/ClickHouse/ClickHouse/pull/78380>

# Query Data Anywhere

80+ built-in integrations with external systems

- S3
- GCS
- Azure Blob
- HDFS
- Kafka
- RabbitMQ
- ODBC/JDBC
- HIVE
- MySQL / PostgreSQL
- MongoDB
- Redis
- Elasticsearch
- ClickHouse-to-ClickHouse Distributed

# Integrate the way you like



# Table Engines

- Istorage.h : Interface for Table Engines
- Implementations => different table engine
  - *StorageMergeTree,*
  - *StoragePostgreSQL,*
  - *StorageKafka2*
  - *StorageObjectStorage (S3, AzureBlob, HDFS)*
  - ... 69 derived classes in total
- Core responsibilities:
  - read(),
  - write(...) , alter(...), drop() etc

The screenshot shows a code editor window with several tabs at the top: main.cpp, IStorage.h (which is the active tab), DatabaseDataLak, and others. The IStorage.h tab displays the following C++ code:

```
namespace DB
{
class IStorage : public std::enable_shared_from_this<IStorage>
{
public:
    /* Storage. Describes the table. Responsible for
     * - storage of the table data;
     * - the definition in which files the data is stored;
     * - data lookups and appends;
     * - data storage structure (compression, etc.)
     * - concurrent access to data (locks, etc.)
     */
protected:
    virtual void setInMemoryFormat(const InMemoryFormat &format) = 0;
    virtual void uniqueKey(const String &key) = 0;
};

class IStorageURLBase : public IStorage
{
public:
    virtual void fromURL(const String &url) = 0;
};

class IStorageSystemZeros : public IStorage
{
public:
    virtual void fromSystemZeros() = 0;
};

class IStorageFromMergeTreeDataPart : public IStorage
{
public:
    virtual void fromMergeTreeDataPart() = 0;
};

class IStorageDummy : public IStorage
{
public:
    virtual void dummy() = 0;
};

class IStorageNATS : public IStorage
{
public:
    virtual void fromNATS() = 0;
};

class IStorageFuzzJSON : public IStorage
{
public:
    virtual void fromFuzzJSON() = 0;
};

class IStorageSystemRemoteDataPaths : public IStorage
{
public:
    virtual void fromSystemRemoteDataPaths() = 0;
};

class IStorageSetOrJoinBase : public IStorage
{
public:
    virtual void fromSetOrJoinBase() = 0;
};

class IStorageObjectStorageQueue : public IStorage
{
public:
    virtual void fromObjectStorageQueue() = 0;
};

class IStorageWindowView : public IStorage
{
public:
    virtual void fromWindowView() = 0;
};
```

A dropdown menu titled "Derived classes of 'IStorage'" is open, listing the following derived classes:

- StorageFuzzQuery (in DB)
- StorageStripeLog (in DB)
- StorageKafka2 (in DB)
- StorageMaterializedView (in DB)
- StorageRedis (in DB)
- StoragePostgreSQL (in DB)
- MergeTreeData (in DB)
- StorageMergeTreeIndex (in DB)
- StorageView (in DB)
- IStorageURLBase (in DB)
- StorageSystemZeros (in DB)
- StorageFromMergeTreeDataPart (in DB)
- StorageDummy (in DB)
- StorageNATS (in DB)
- StorageFuzzJSON (in DB)
- StorageSystemRemoteDataPaths (in DB)
- StorageSetOrJoinBase (in DB)
- StorageObjectStorageQueue (in DB)
- StorageWindowView (in DB)

# Database Engines

- Idatabase.h: interface for all database engines.
- Handles collections of tables
  - not individual rows or columns.
- Core Responsibilities (Table Management)
  - *createTable()*, *dropTable()*, *renameTable()*,
  - *attachTable()*, *detachTable()*,
  - *getTable(name)*,
- Query Support
  - *getTablesIterator()* – for queries like *SHOW TABLES*
  - *getCreateTableQuery()*,
  - *getCreateDatabaseQuery()*

```
main.cpp  IDatabase.h  Database.h

21 namespace DB
157 > /** Database engine. ... */
165
166 class IDatabase : public std::...
167
168 Derived classes of 'IDatabase'
169 DatabaseSQLite (in DB)
170 DatabasePostgreSQL (in DB)
171 DatabasesOverlay (in DB)
172 DatabaseDataLake (in DB)
173 DatabaseS3 (in DB)
174 DatabaseWithOwnTablesBase (in DB)
175 DatabaseFilesystem (in DB)
176 DatabaseMySQL (in DB)
177 DatabaseHDFS (in DB)
178 DatabaseMemory (in DB)
179 DatabaseOnDisk (in DB)
180 DatabaseLazy (in DB)
181 DatabaseOrdinary (in DB)
182 DatabaseAtomic (in DB)
183 DatabaseBackup (in DB)
184 DatabaseReplicated (in DB)
185 DatabaseMaterializedPostgreSQL (in DB)
186
187
188
```

<https://github.com/ClickHouse/ClickHouse/blob/master/src/Databases/IDatabase.h>

# Datalake Engines

```
#if USE_AVRO && USE_PARQUET
namespace DB
{
    class DatabaseDataLake final : public IDatabase, WithContext
    {
        public:
            explicit DatabaseDataLake(
                const std::string & database_name_,
                const std::string & url_,
                const DatabaseDataLakeSettings & settings_,
                ASTPtr database_engine_definition_,
                ASTPtr table_engine_definition_);

            String getEngineName() const override { return "DataLake"; }

            bool canContainMergeTreeTables() const override { return false; }
            bool canContainDistributedTables() const override { return false; }
            bool shouldBeEmptyOnDetach() const override { return false; }

            bool empty() const override;

            bool isTableExist(const String & name, ContextPtr context)
                StoragePtr tryGetTable(const String & name, ContextPtr context);

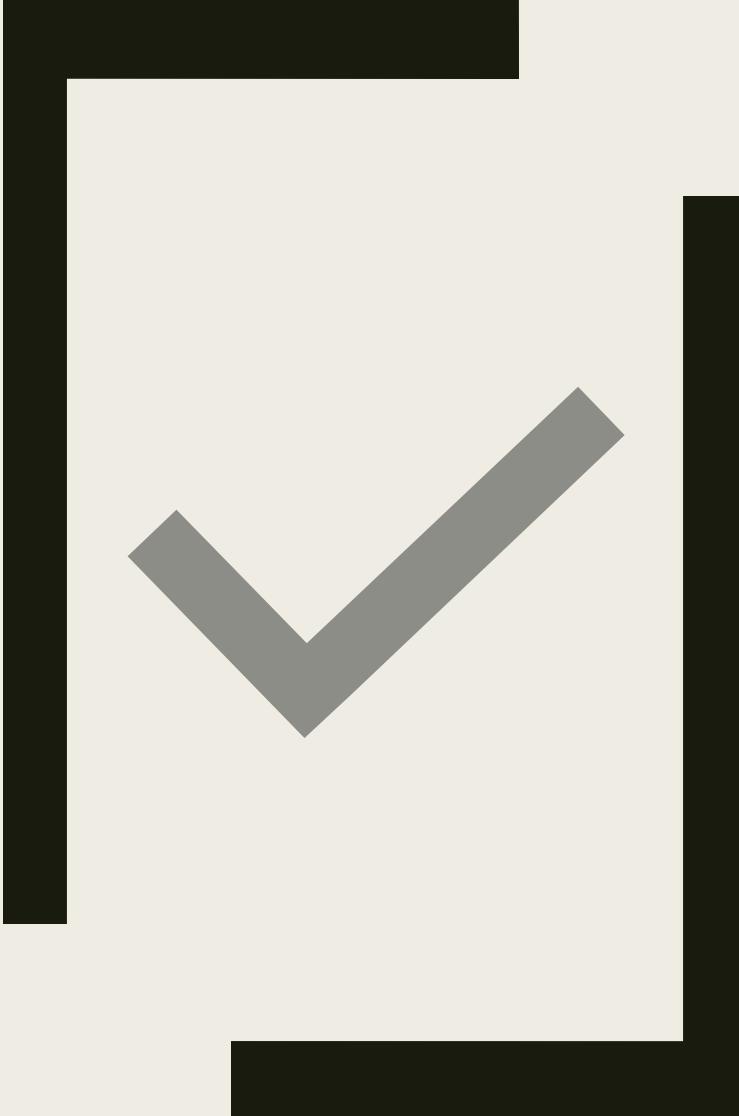
            DatabaseTablesIteratorPtr getTablesIterator();
    };
}
```

Derived classes of 'IDatabase'

- DatabaseSQLite (in DB)
- DatabasePostgreSQL (in DB)
- DatabasesOverlay (in DB)
- DatabaseDataLake (in DB)**
- DatabaseS3 (in DB)
- DatabaseWithOwnTablesBase (in DB)
- DatabaseFilesystem (in DB)
- DatabaseMySQL (in DB)
- DatabaseHDFS (in DB)
- DatabaseMemory (in DB)
- DatabaseOnDisk (in DB)
- DatabaseLazy (in DB)
- DatabaseOrdinary (in DB)
- DatabaseAtomic (in DB)
- DatabaseBackup (in DB)
- DatabaseReplicated (in DB)
- DatabaseMaterializedPostgreSQL (in DB)

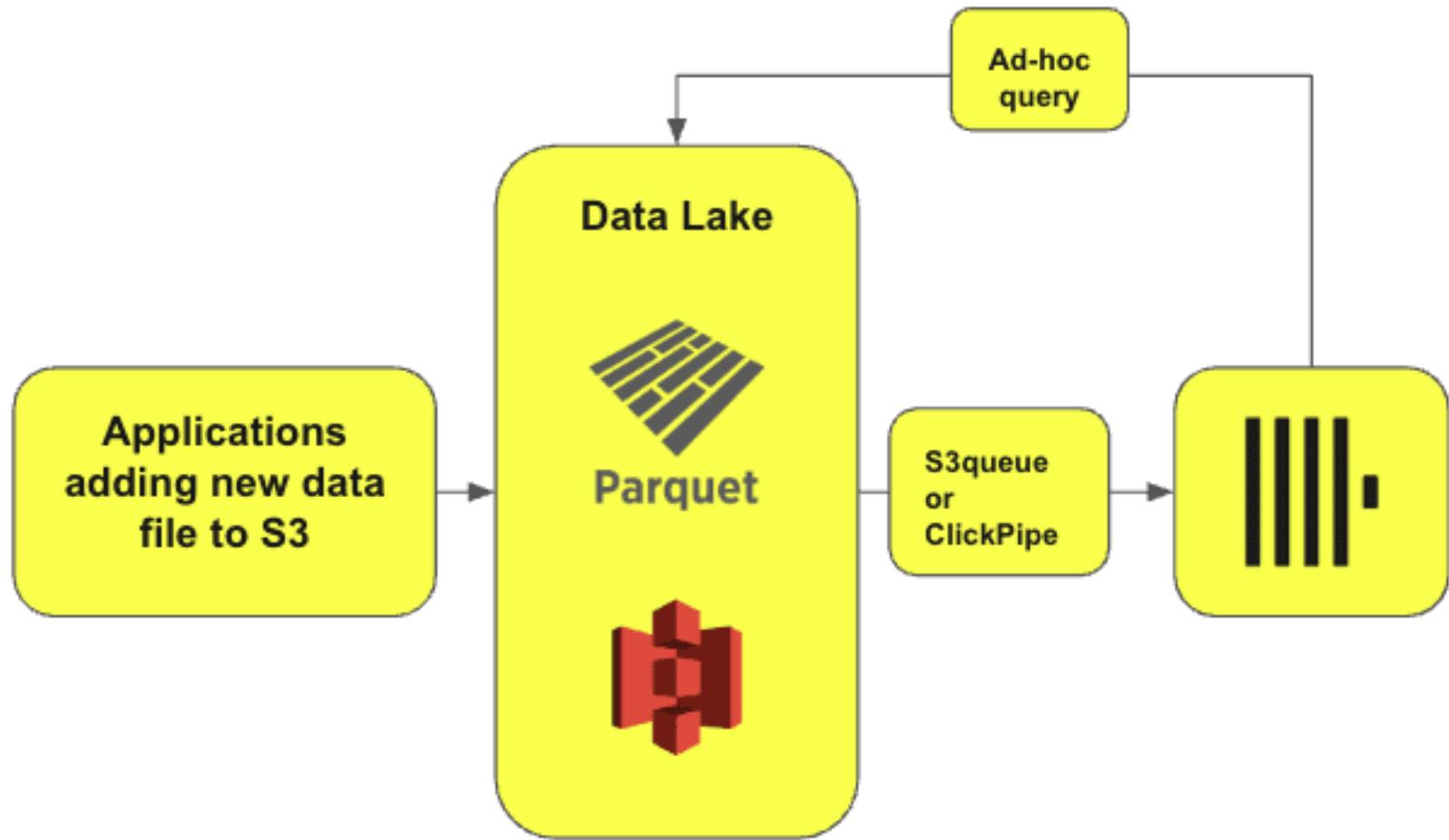
# Why It Matters for Lakehouse

- If you have Parquet,
- if you have Iceberg,
- if you have data on object storage
  - *ClickHouse can query it*

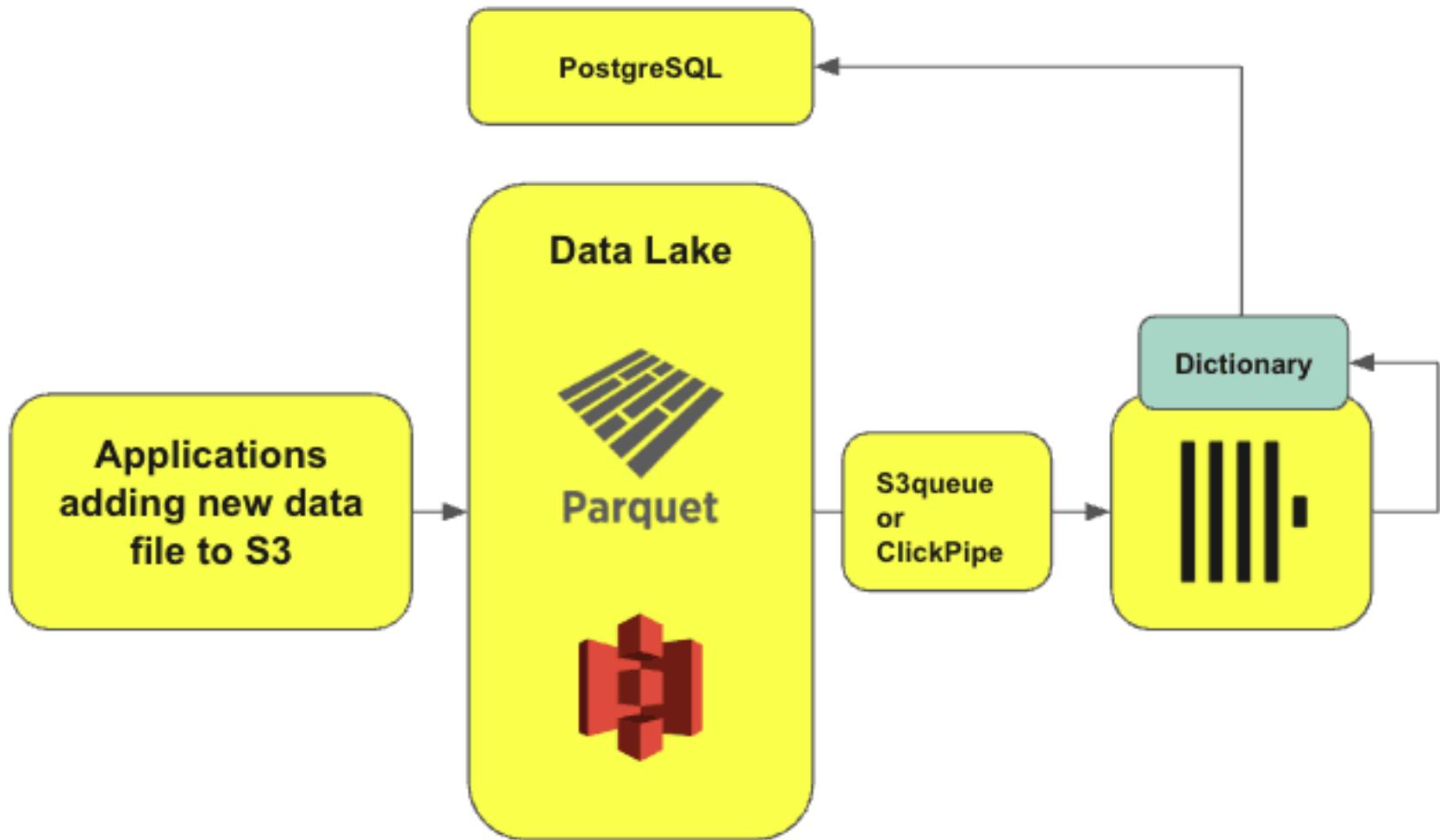


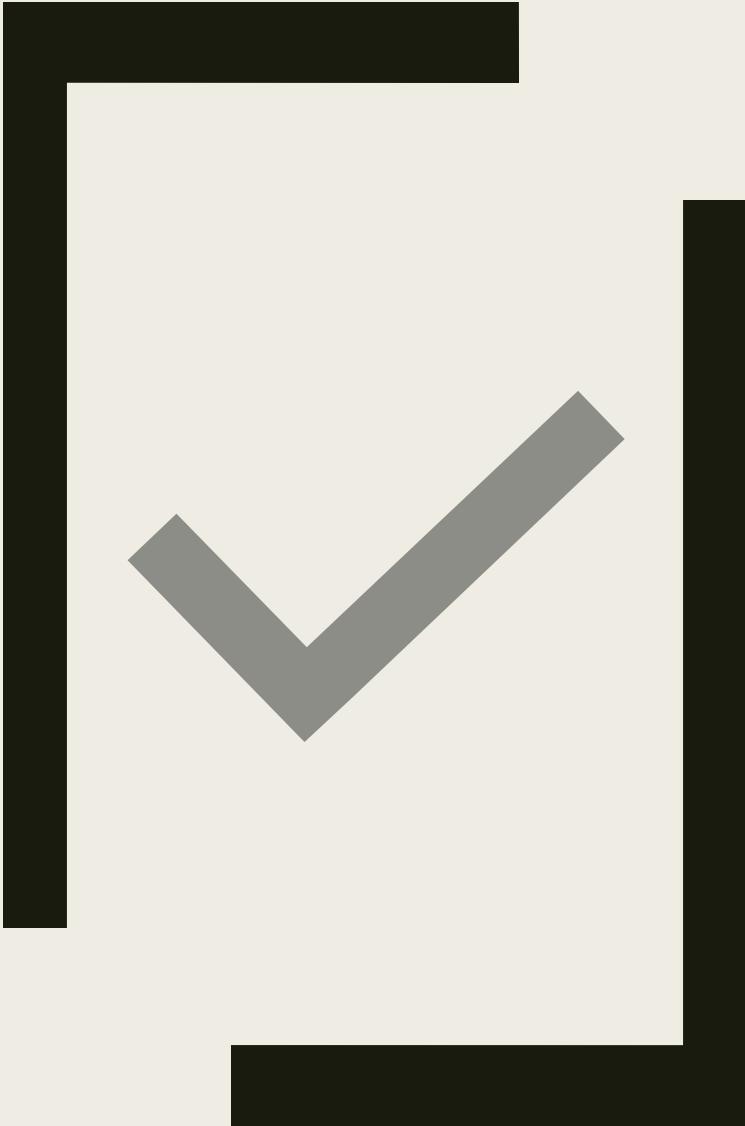
READ  
PARQUET &  
ICEBERG  
FROM  
CLICKHOUSE

# Ad-hoc queries (Trino etal..)



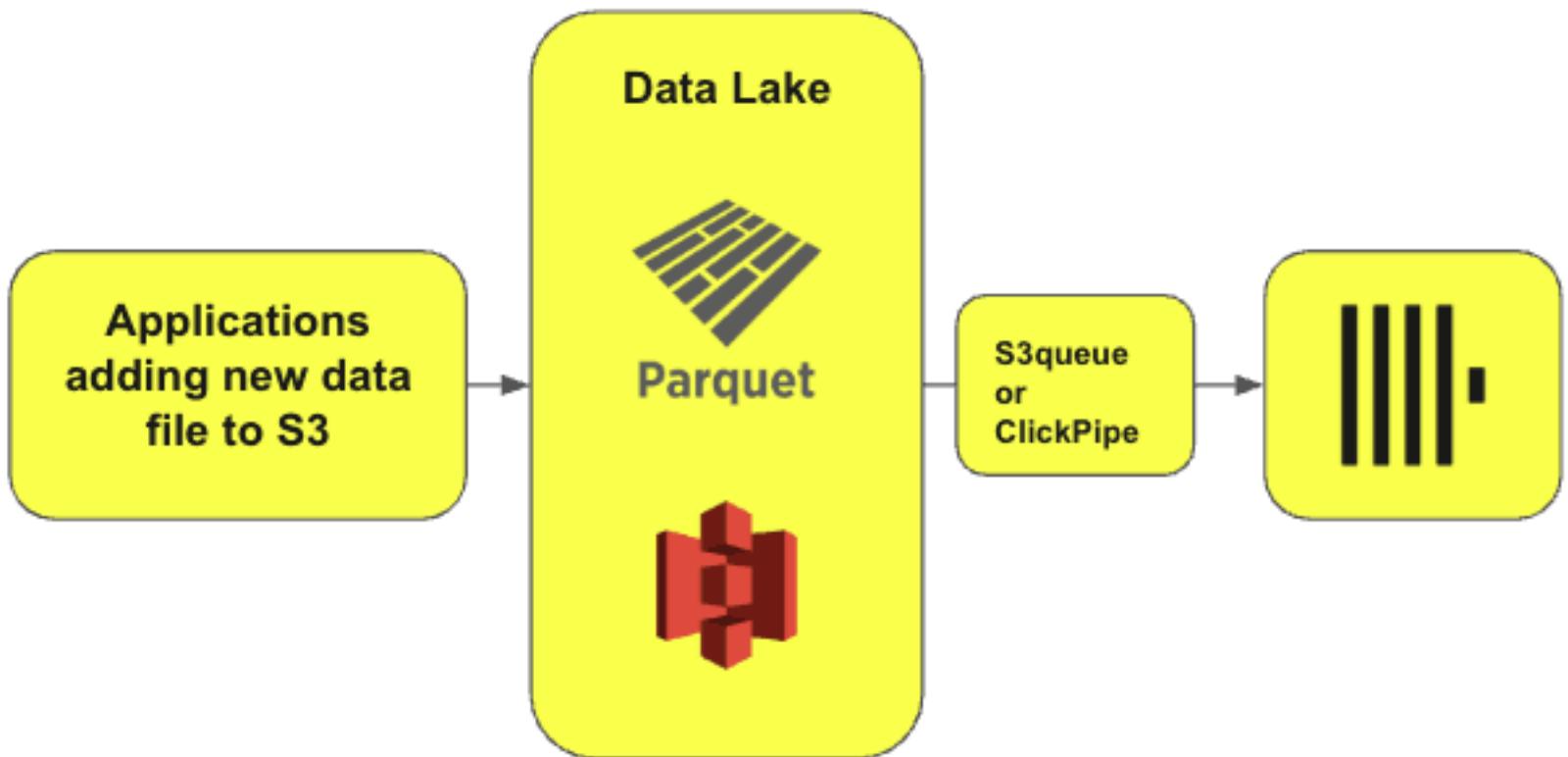
# Querying multiple sources



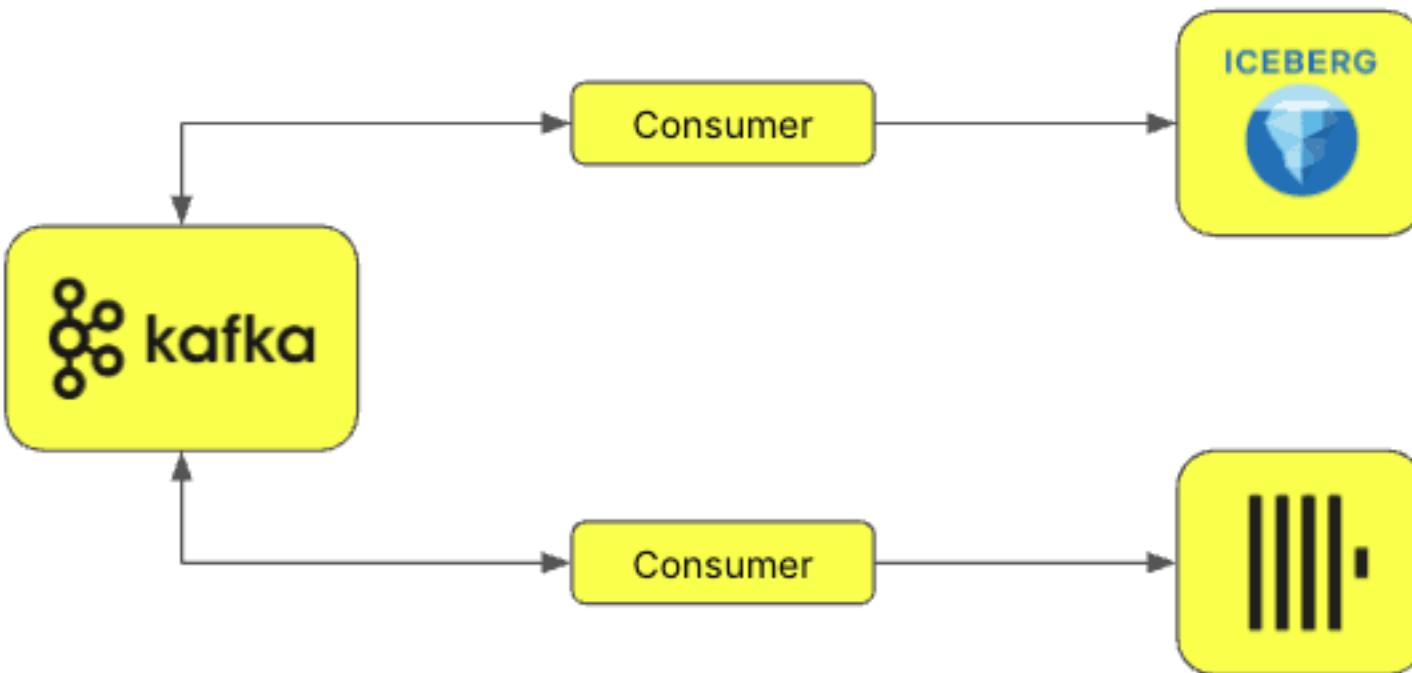


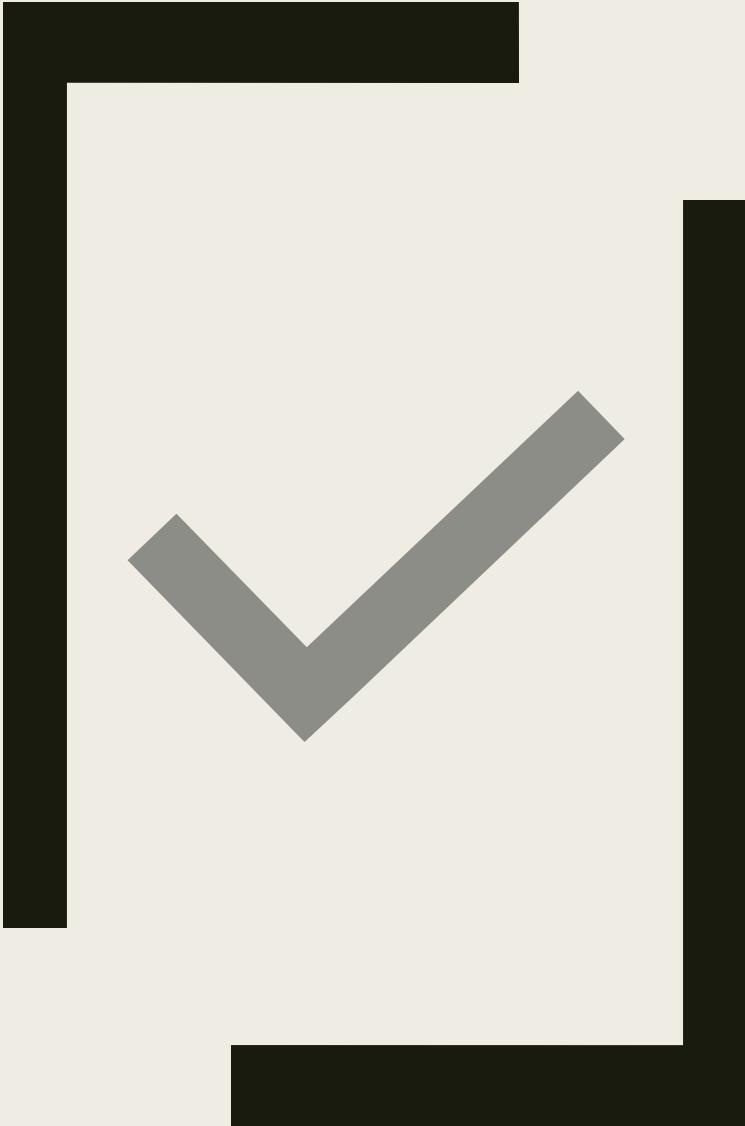
WRITE  
IN  
ICEBERG  
FORMAT

# Writes(1): s3cluster/s3queue



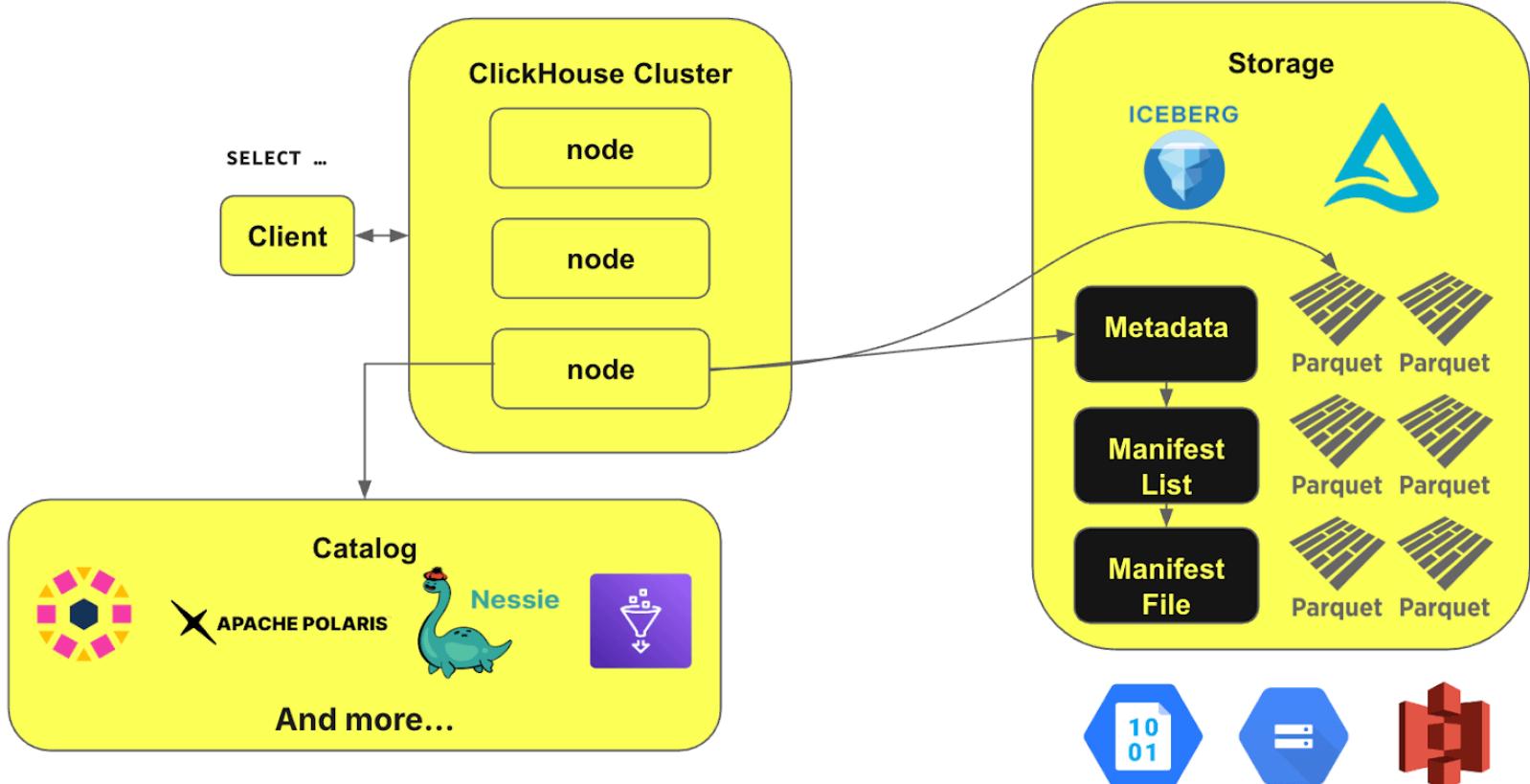
# Writes(2): Double write/ttl/tiering



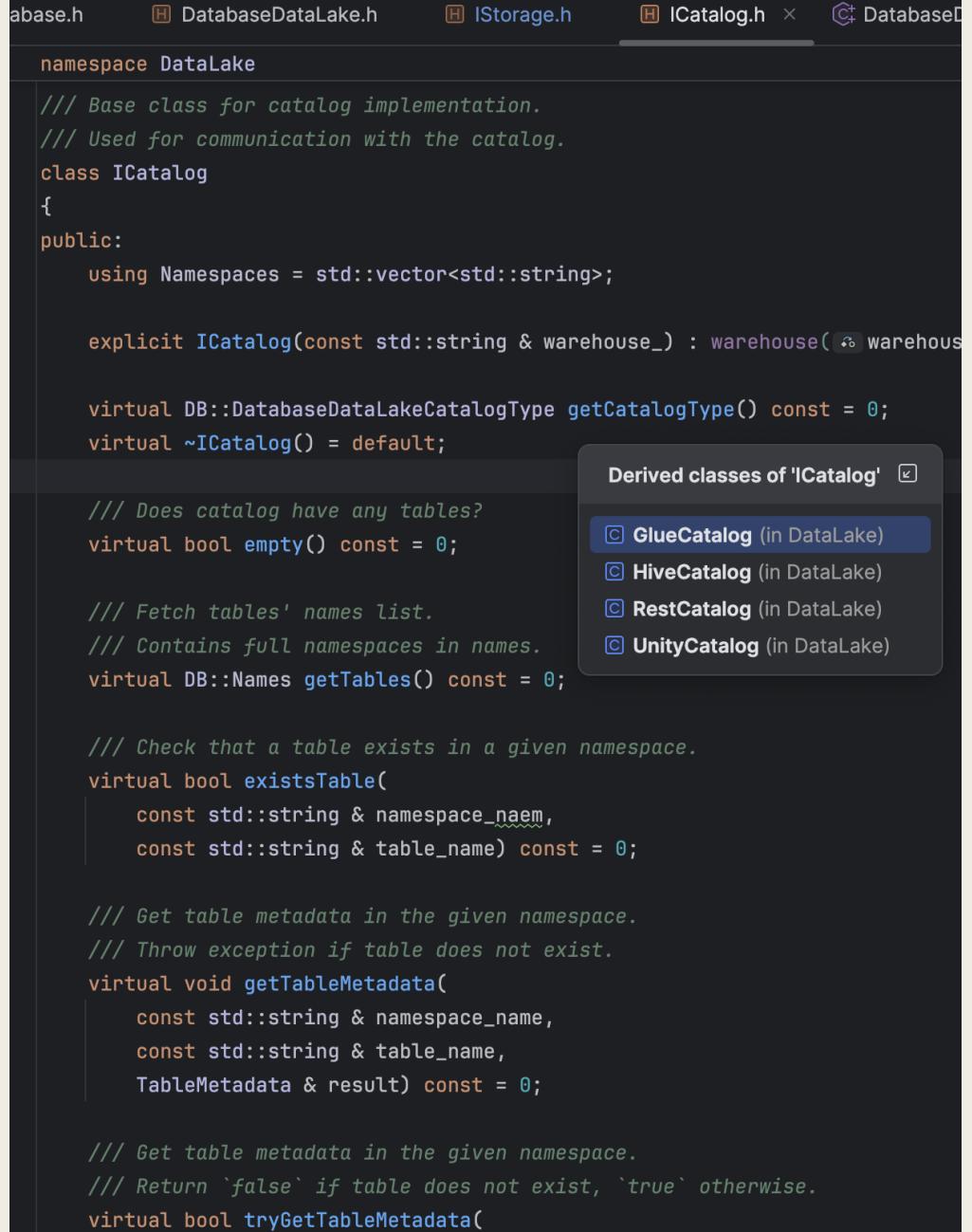


SUPPORT  
FOR  
OTHER  
ICEBERG  
GOODIES!

# Catalog Integration



# Catalog Integration



```
base.h DatabaseDataLake.h IStorage.h ICatalog.h DatabaseD

namespace DataLake
{
    /// Base class for catalog implementation.
    /// Used for communication with the catalog.

    class ICatalog
    {
public:
    using Namespaces = std::vector<std::string>;

    explicit ICatalog(const std::string & warehouse_) : warehouse(warehouse_) {}

    virtual DB::DatabaseDataLakeCatalogType getCatalogType() const = 0;
    virtual ~ICatalog() = default;

    /// Does catalog have any tables?
    virtual bool empty() const = 0;

    /// Fetch tables' names list.
    /// Contains full namespaces in names.
    virtual DB::Names getTables() const = 0;

    /// Check that a table exists in a given namespace.
    virtual bool existsTable(
        const std::string & namespace_name,
        const std::string & table_name) const = 0;

    /// Get table metadata in the given namespace.
    /// Throw exception if table does not exist.
    virtual void getTableMetadata(
        const std::string & namespace_name,
        const std::string & table_name,
        TableMetadata & result) const = 0;

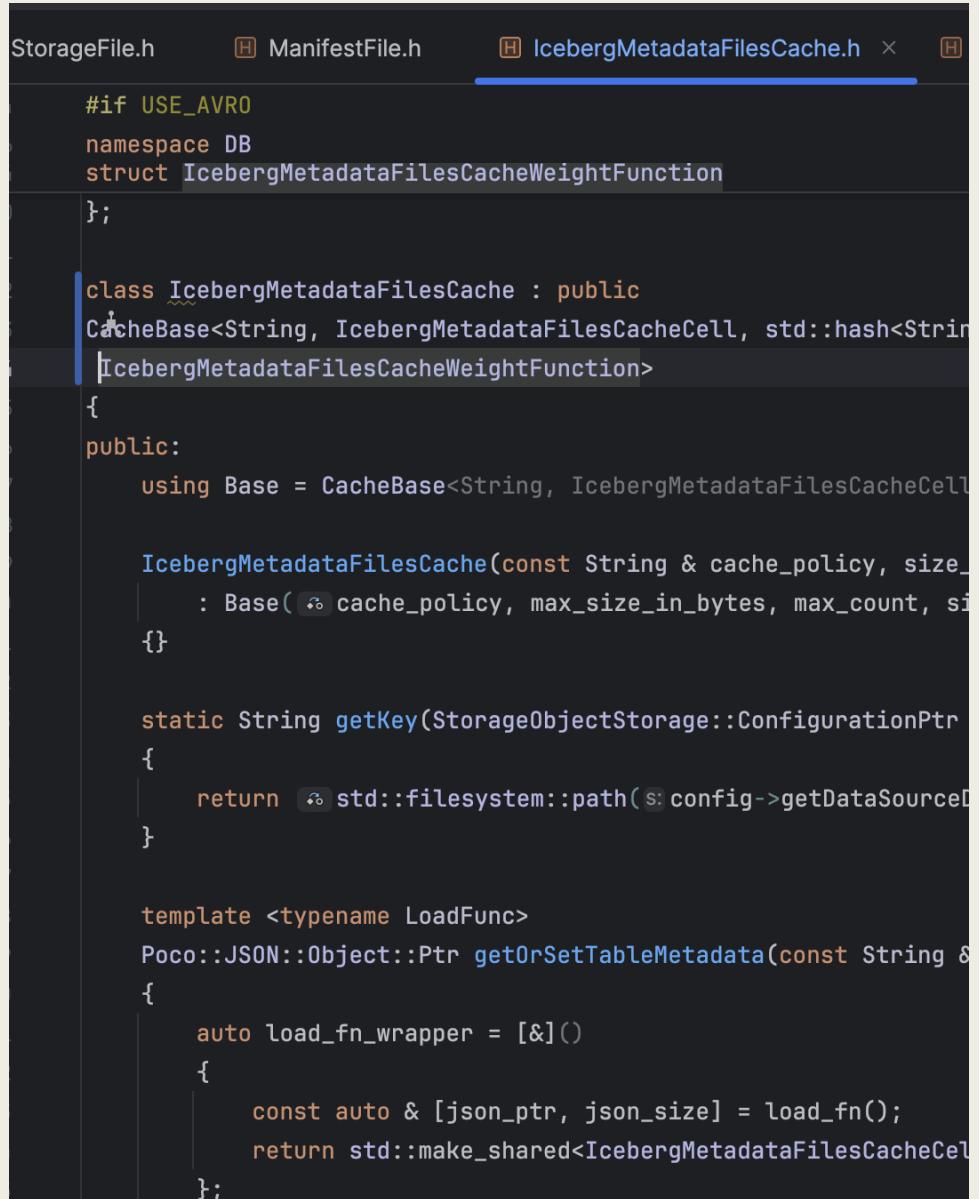
    /// Get table metadata in the given namespace.
    /// Return 'false' if table does not exist, 'true' otherwise.
    virtual bool tryGetTableMetadata(
        const std::string & namespace_name,
        const std::string & table_name,
        TableMetadata & result) const = 0;
};

} // namespace DataLake
```

Derived classes of 'ICatalog'

- GlueCatalog (in DataLake)
- HiveCatalog (in DataLake)
- RestCatalog (in DataLake)
- UnityCatalog (in DataLake)

# Metadata caching



The screenshot shows a code editor with several tabs at the top: StorageFile.h, ManifestFile.h, IcebergMetadataFilesCache.h (which is the active tab), and another partially visible tab. The code in the editor is for a class named IcebergMetadataFilesCache, which inherits from CacheBase<String, IcebergMetadataFilesCacheCell, std::hash<String>, IcebergMetadataFilesCacheWeightFunction>. The class has a constructor taking cache\_policy, max\_size\_in\_bytes, and max\_count. It includes a static method getKey that returns a path based on config->getDataSourceId. There's also a template method getOrSetTableMetadata that uses a lambda to wrap a load function.

```
#if USE_AVRO
namespace DB
struct IcebergMetadataFilesCacheWeightFunction
;

class IcebergMetadataFilesCache : public
CacheBase<String, IcebergMetadataFilesCacheCell, std::hash<String>,
IcebergMetadataFilesCacheWeightFunction>
{
public:
    using Base = CacheBase<String, IcebergMetadataFilesCacheCell>;

    IcebergMetadataFilesCache(const String & cache_policy, size_t max_size_in_bytes, size_t max_count)
        : Base(cache_policy, max_size_in_bytes, max_count, std::hash<String>(),
               IcebergMetadataFilesCacheWeightFunction())
    {}

    static String getKey(StorageObjectStorage::ConfigurationPtr config)
    {
        return std::filesystem::path(config->getDataSourceId());
    }

    template <typename LoadFunc>
    Poco::JSON::Object::Ptr getOrSetTableMetadata(const String & table_name, const String & database_name)
    {
        auto load_fn_wrapper = [&]()
        {
            const auto & [json_ptr, json_size] = load_fn();
            return std::make_shared<IcebergMetadataFilesCacheCell>(std::move(json_ptr));
        };
    }
};
```

# What Works Today

- Experimental (still building...)
- Read-only queries, no writes
  - *In tests/Demo, we use spark for writes.*
- Catalog support (Glue, Hive, Unity, REST)
- Time Travel([PR](#)), Partition Pruning([PR](#)), Schema Evolution([PR](#))
- Metadata caching : in-memory ([PR](#))

# THE ROAD AHEAD

thub.com/ClickHouse/ClickHouse/issues/74046

## Roadmap 2025 #74046

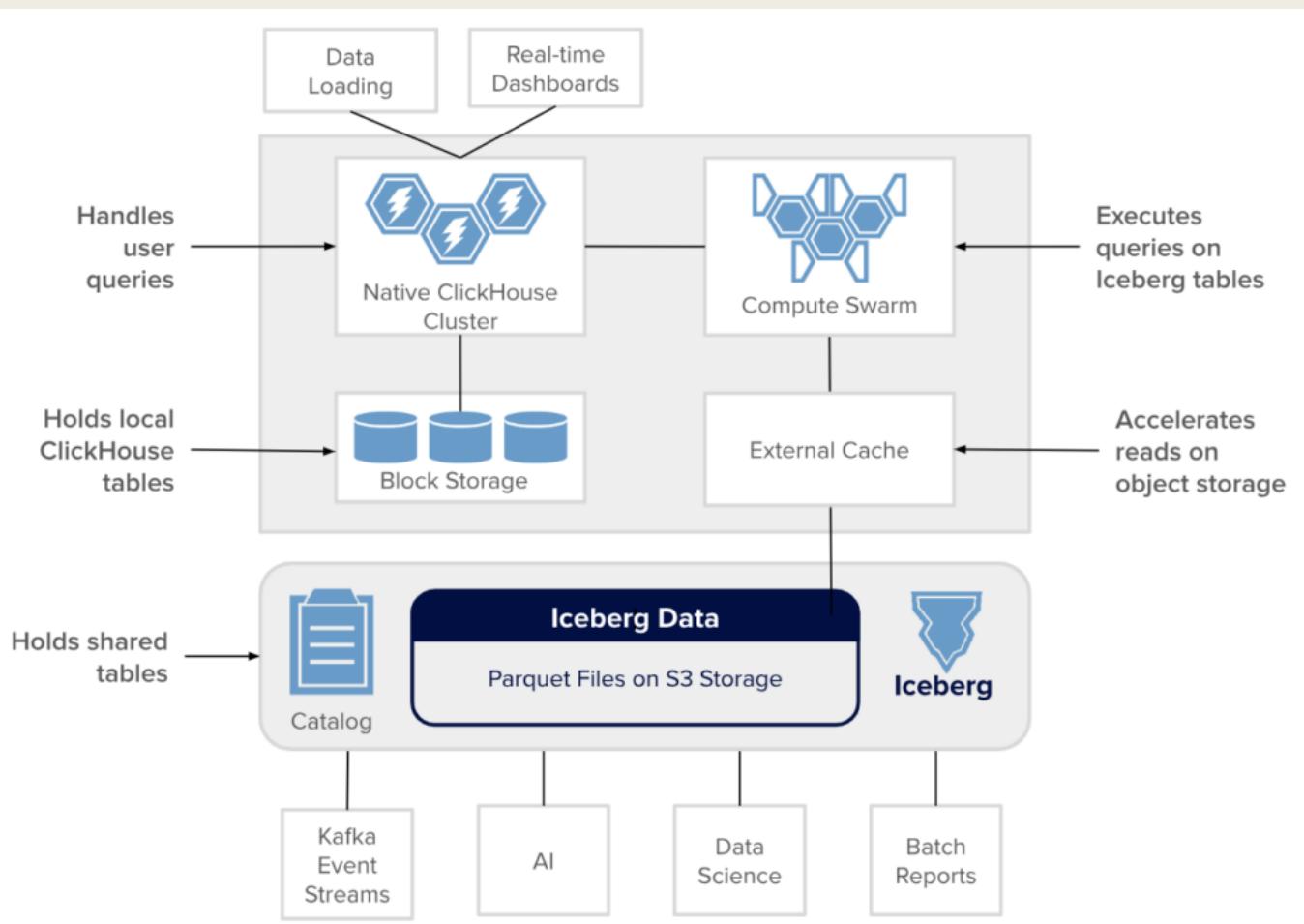
### Data Lakes

- Automatic use of cluster functions ↗ [Use cluster table functions automatically if parallel replicas are enabled #70659](#)
- Parallel distributed INSERT SELECT by default
- Enable Hive-style partitioning by default ↗ [Enable use\\_hive\\_partitioning by default #71636](#)
- Uniform distribution of load across cluster ⓘ [s3Cluster is suboptimal when the number of files is comparable to the number of servers #70190](#)
- Partition pruning for Iceberg ↗ [Iceberg Partition Pruning for time-related partition transforms #72044](#)
- Prewhere support for Parquet ⓘ [\[Feature\] Add support for prewhere in parquet native reader #65527](#)
- Unify different Parquet readers into one ⓘ [A new parquet reader that supports filter push down, which improves query performance and reduces total time on clickbench by 50+% compared to arrow parquet reader #70611](#)
- Writing bloom filters for Parquet ↗ [Write Parquet bloom filters #71681](#)
- Time travel for Iceberg ↗ [Add setting to query Iceberg tables as of a specific timestamp #71072](#)
- Support for deletions in Iceberg ⓘ [Cannot read Iceberg table: positional and equality deletes are not supported #66588](#)
- In-memory metadata cache ⓘ [A cache for data lake's metadata #71579](#)
- Integration with Delta kernel ⓘ [Integrate with delta\\_kernel #75255](#)
- Glue catalog support ↗ [Unity catalog integration #76988](#)
- Unity catalog support ↗ [Add glue catalog integration #77257](#)
- Support for Delta Lake on Azure ↗ [Support deltalake for AzureBlobStorage #74541](#)
- Materialized views on top of data lakes
- Support arbitrary nesting in database and table names ⓘ [Support arbitrary nesting in database and table names. #71171](#)
- Writing into data lakes ⓘ [Support for Writing to Apache Iceberg Tables in ClickHouse #49973](#)
- Background merges for data lakes ⓘ
- Subsets of a large file as a unit of work

# PROJECT ANTALYA

# 1 more thing: Project Antalya

- [Project Antalya Announcement Blog](#)
- [Example and documentation – github repo](#)
- [Antalya Open-source launch webinar \(2025\)](#)
- [Antalya Office Hours Q&A](#)



# Live Demo

- Demo setup can be found [here](#)





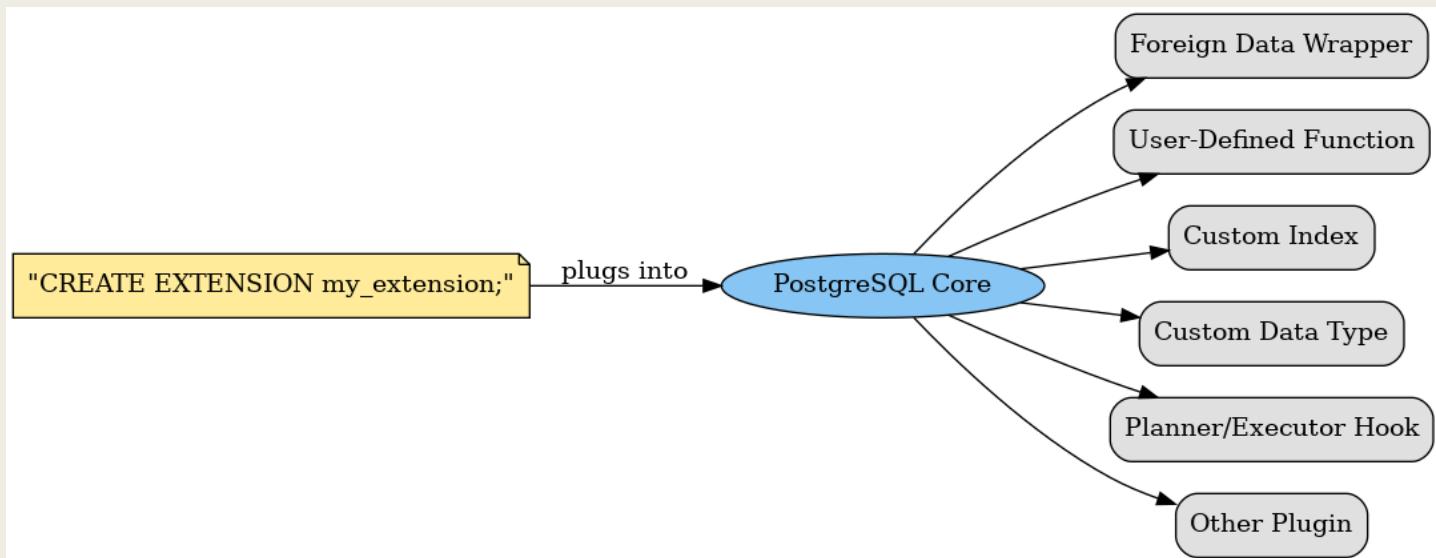
# HACKING POSTGRESQL FOR ICEBERG



# Built to extend

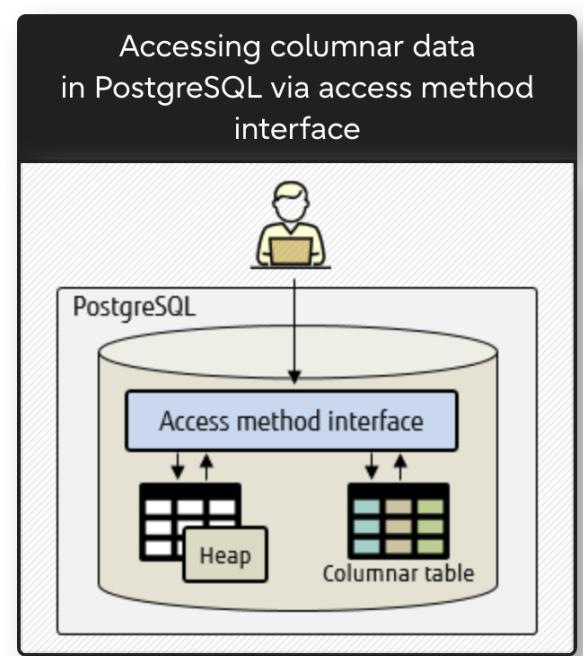
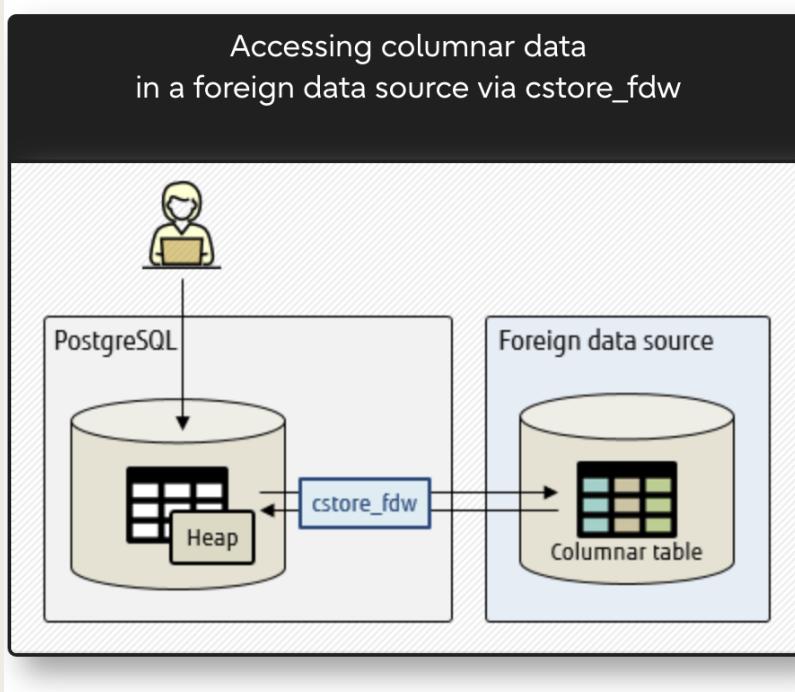
- Integration with Extensions

- *TimescaleDB* (*time-series*),
- *Citus* (*distributed SQL*),
- *PostGIS* (*geospatial data*)
- *pg\_duckdb*



# Extend even more!

- Foreign Data Wrappers (FDW)
  - (*parquet\_fdw*, *clickhouse\_fdw...*)
  - *paradeDB's pg\_analytics*
- Table Access Methods (TAM)
  - *pg\_mooncake* (*columnstore TAM*)
  - *Hydra* (*columnar TAM*)



# DuckDB inside Postgres, what?

- Embeddable, in-memory analytics engine
- Optimized for single-node execution
- Columnar format
- Vectorized processing
- Follows SQL Syntax Closely
- Flexible
  - *Can read (parquet / Iceberg) on S3*
  - *MotherDuck for DBaaS*



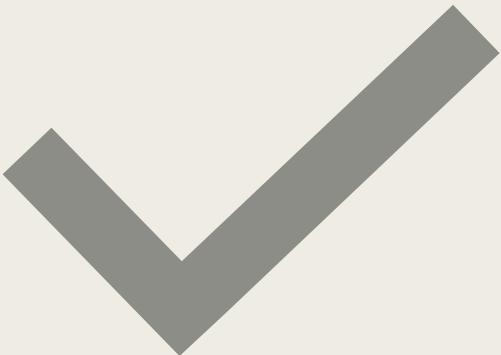
1. [github.com/duckdb/pg\\_duckdb](https://github.com/duckdb/pg_duckdb)
2. [github.com/hydradatabase/columnar](https://github.com/hydradatabase/columnar)
3. [github.com/paradedb/pg\\_analytics](https://github.com/paradedb/pg_analytics)
4. [github.com/Mooncake-Labs/pg\\_mooncake](https://github.com/Mooncake-Labs/pg_mooncake)

# pg\_duckdb == Postgres(PG) + DuckDB



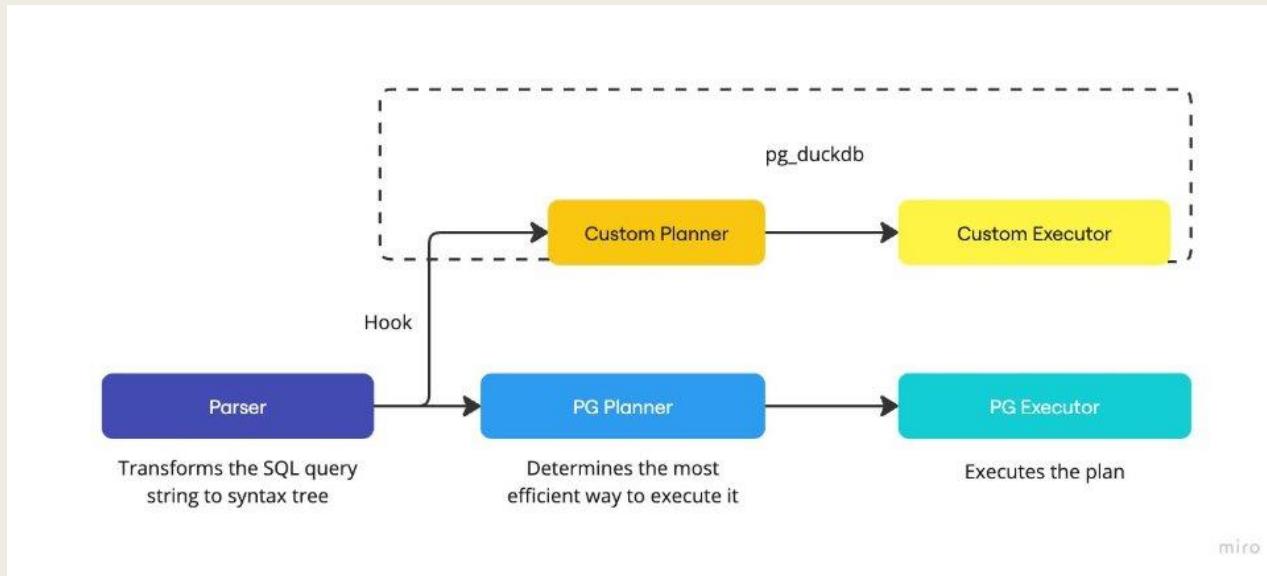
## Data Lake Functions

Name	Description
<a href="#">read_parquet</a>	Read a parquet file
<a href="#">read_csv</a>	Read a CSV file
<a href="#">read_json</a>	Read a JSON file
<a href="#">iceberg_scan</a>	Read an Iceberg dataset
<a href="#">iceberg_metadata</a>	Read Iceberg metadata
<a href="#">iceberg_snapshots</a>	Read Iceberg snapshot information
<a href="#">delta_scan</a>	Read a Delta dataset



LET'S GET  
OUR HANDS  
DIRTY,  
SHALL WE?

# Query Stealing in pg\_duckdb



- **planner\_hook** to create a planned statement that DuckDb can understand and execute.
- **ExecutorStart\_hook** and **ExecutorEnd\_hook** used by `pg_duckdb` for snapshot internals in non SELECT queries.
- **ExplainOneQuery\_hook** overrides the output of **EXPLAIN** query.

# Hooks initialization and Query Stealing

```
void DuckdbInitHooks(void) {
    prev_planner_hook = planner_hook;
    planner_hook = DuckdbPlannerHook;

    prev_executor_start_hook = ExecutorStart_hook ? ExecutorStart_hook : DuckdbExecutorStartHook;

    prev_executor_finish_hook = ExecutorFinish_hook ? ExecutorFinish_hook : DuckdbExecutorFinishHook;

    prev_explain_one_query_hook = ExplainOneQuery_hook;
    ExplainOneQuery_hook = DuckdbExplainOneQueryHook;

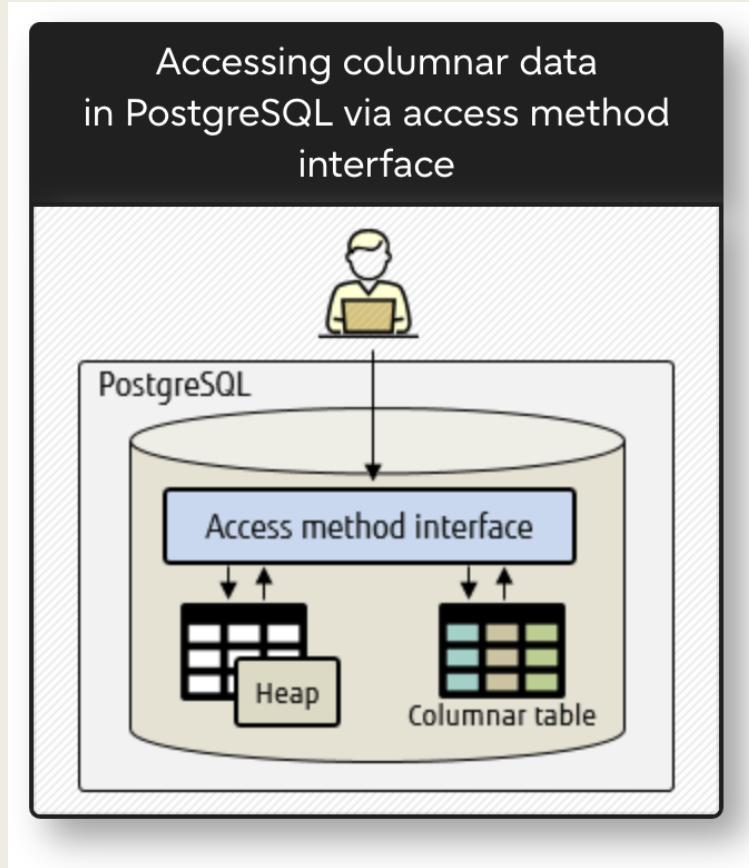
    DuckdbInitUtilityHook();
}
```

[DuckDbInitHooks](#)

```
static PlannedStmt* DuckdbPlannerHook_Cpp(Query *parse,
                                            const char *query_string, int cursor_options,
                                            ParamListInfo bound_params) {
    if (pgduckdb::IsExtensionRegistered()) {
        if (NeedsDuckdbExecution(parse)) {
            IsAllowedStatement(parse, true);
            return DuckdbPlanNode(parse, query_string, cursor_options);
        } else if (duckdb_force_execution && IsAllowedStatement(parse, false)) {
            PlannedStmt *duckdbPlan =
                DuckdbPlanNode(parse, query_string, cursor_options);
            if (duckdbPlan) {
                return duckdbPlan;
            }
        }
    }
    /* If we can't create a plan, we'll fall back to the original hook */
}
```

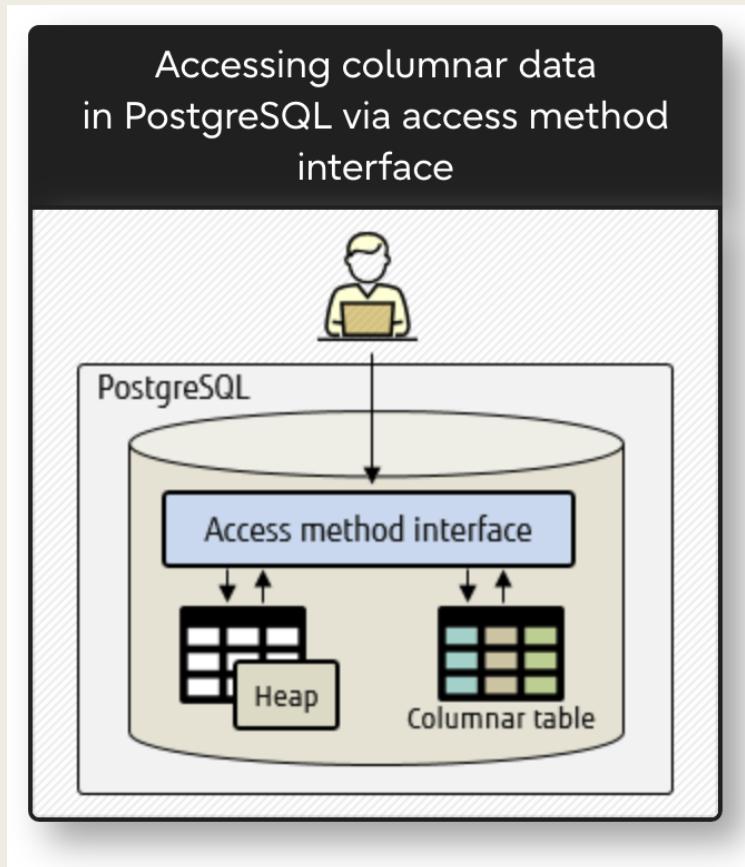
[DuckDbPlannerHooks](#)

# pg\_mooncake (pg\_duckdb++)



- New columnar storage behind TAM
- pg\_mooncake supports loading data from
  - Postgres heap tables,
  - Parquet, CSV, JSON files
  - Iceberg, Delta Lake tables
- Write iceberg\*/ delta tables with parquet
  - On local disk
  - Or object store

# pg\_mooncake (pg\_duckdb++)



## 1. Enable the extension

```
CREATE EXTENSION pg_mooncake;
```

## 2. Create a columnstore table:

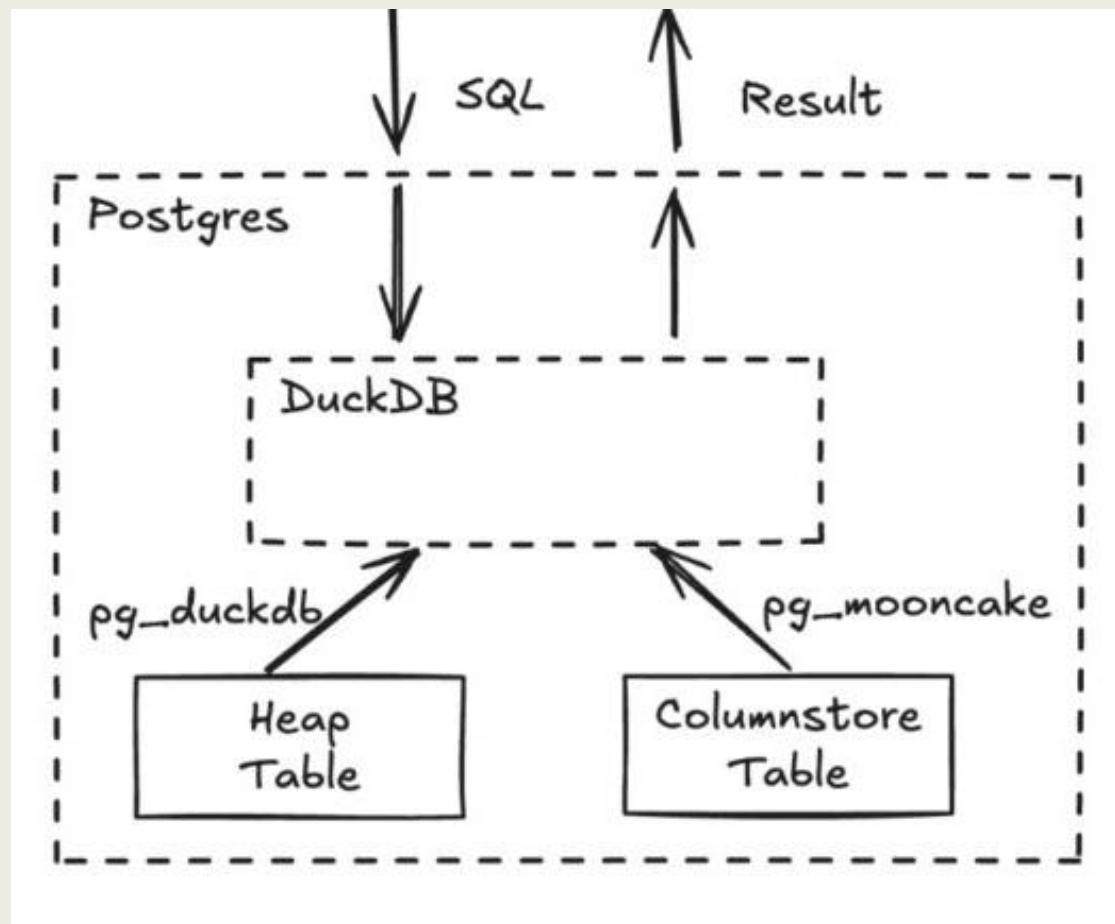
```
CREATE TABLE user_activity(
    user_id BIGINT,
    activity_type TEXT,
    activity_timestamp TIMESTAMP,
    duration INT
) USING columnstore;
```

## 3. Insert data:

```
INSERT INTO user_activity VALUES
(1, 'login', '2024-01-01 08:00:00', 120),
(2, 'page_view', '2024-01-01 08:05:00', 30),
(3, 'logout', '2024-01-01 08:30:00', 60),
(4, 'error', '2024-01-01 08:13:00', 60);
```

```
SELECT * from user_activity;
```

# Query execution flow in pg\_mooncake



<https://www.mooncake.dev/blog/how-we-built-pgmooncake>

# Anatomy of a PostgreSQL Extension

Extensions are modular add-ons that can introduce new capabilities to vanilla PostgreSQL server.

- **SQL file:**
  - Executes SQL commands to create objects (*functions, types, operators, tables*) inside PostgreSQL's system catalogs.
- **Control file:**
  - Declares extension metadata such as *version, description, and installation behavior*. Used by the extension loader to locate and manage the extension lifecycle.
- **Source code:**
  - Dynamically loaded at runtime to register native functions or hooks. User functions can be written in other languages like rust as well.

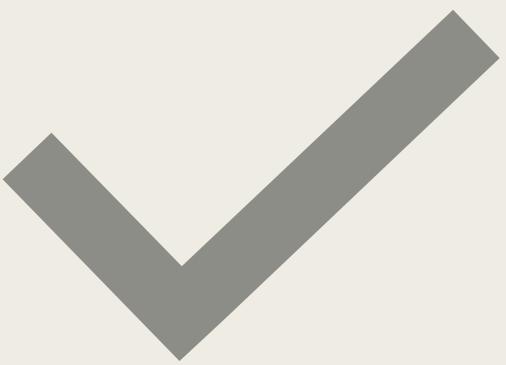
# PostgreSQL Hooks

- Global Function pointers exposed by the backend.
- Extensions can intercept and modify internal behavior at various stages of query processing.
- No core code patch needed!!
- Extensions assign their functions to the hook during `_PG_init()` and remove them using `_PG_fini()`
- **planner hook** allows extensions to intercept and modify the query planning phase, enabling custom plan trees or query routing.
- **Executor Hooks** lets extensions inject logic at the various stages of execution (**ExecutorStart\_hook**, **ExecutorFinish\_hook**), often used for custom data writes or result handling.
- **ExplainOneQuery\_hook** allows overriding the default EXPLAIN procedure for a single query.

# Query stealing

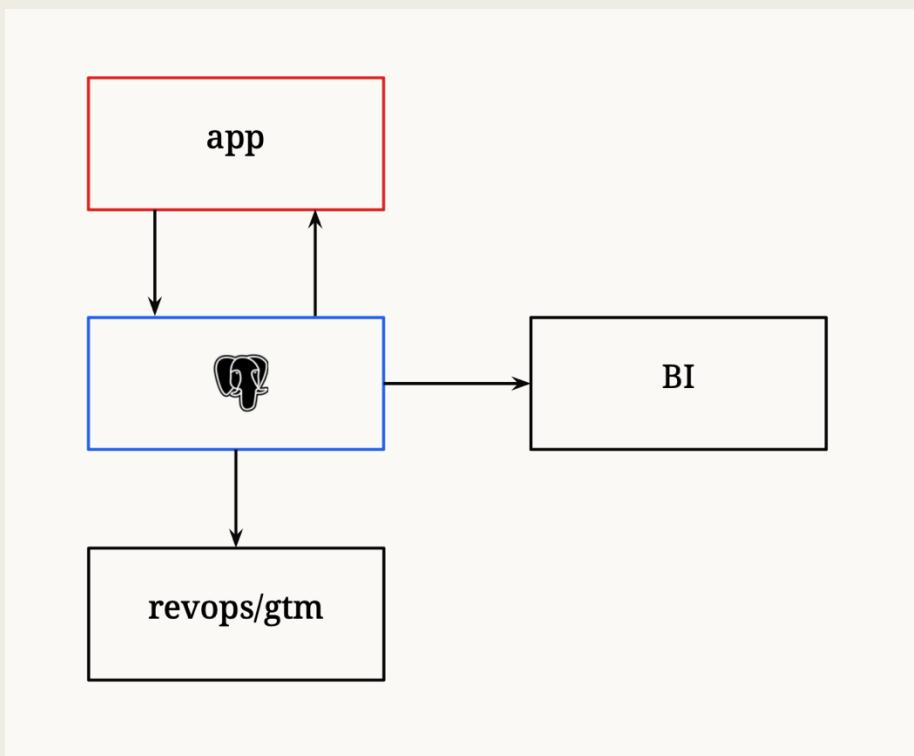
```
static PlannedStmt* DuckdbPlannerHook_Cpp(Query *parse,
    const char *query_string, int cursor_options,
    ParamListInfo bound_params) {
    if (pgduckdb::IsExtensionRegistered()) {
        if (NeedsDuckdbExecution(parse)) {
            IsAllowedStatement(parse, true);
            return DuckdbPlanNode(parse, query_string, cursor_options, bound_params, true);
        } else if (duckdb_force_execution && IsAllowedStatement(parse)) {
            PlannedStmt *duckdbPlan =
                DuckdbPlanNode(parse, query_string, cursor_options, bound_params, false);
            if (duckdbPlan) {
                return duckdbPlan;
            }
            /* If we can't create a plan, we'll fall back to Postgres */
        }
    }
}
```

## DuckdbPlannerHook



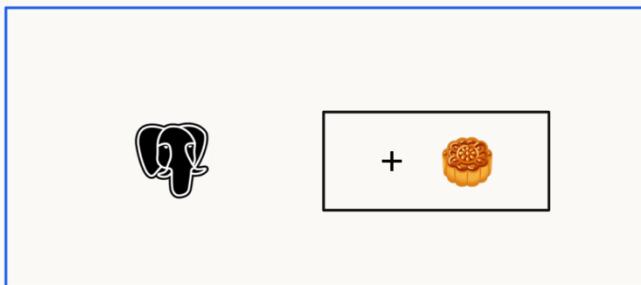
HOW DO I  
USE IT?  
LOADING.. ☺

# Operations : day1



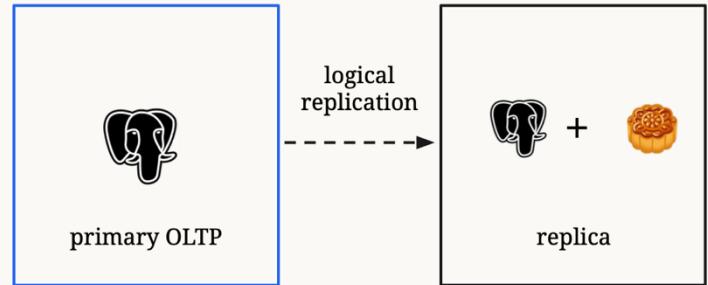
# Operations: Day 2 & 3

as an extension (HTAP)



self-managed, Neon Postgres

as an analytics replica



RDS, Aurora, CloudSQL

in your VPC

# Logical replication from Postgres to Iceberg

Source (any postgres):

```
CREATE PUBLICATION pub  
FOR TABLE orders, customers;
```

Destination (Crunchy Data Warehouse):

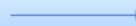
```
CREATE SUBSCRIPTION sub  
CONNECTION 'host=...'  
PUBLICATION pub  
WITH (create_tables_using 'iceberg');
```



Auto-create Iceberg tables, if not exists



Copy initial data



Replicate changes (insert, **update**, delete, truncate)

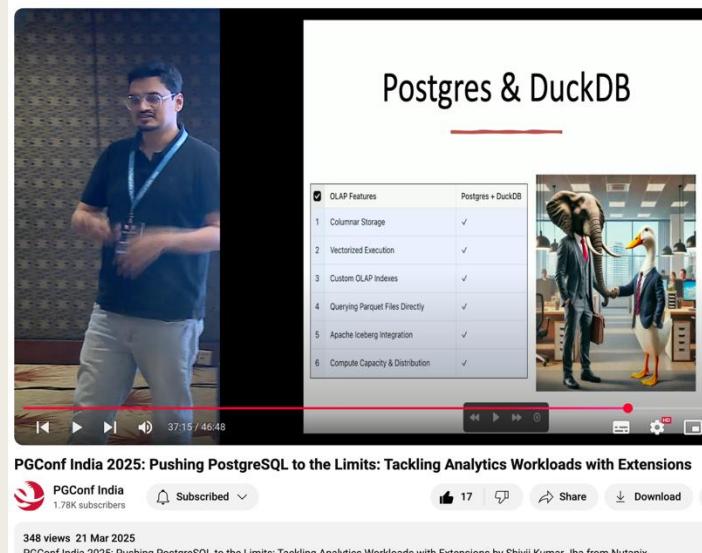
Periodically flush change batches to Iceberg

Automatic compaction

Available today on Crunchy Bridge!

# Reading further!

- See my PGConf India talk for a deep dive (right)
- Signals that Iceberg is
  - *no longer just for distributed compute engines*
  - *travelling far and wide...*



[tinyurl.com/shiv-pgconf-talk](https://tinyurl.com/shiv-pgconf-talk)

[tinyurl.com/shiv-pgcong-slide](https://tinyurl.com/shiv-pgcong-slide)

# Live Demo

- Demo setup can be found [here](#)



# WRAPPING UP

# Why This Matters

- Performance (native format) + openness (Iceberg)
  - *Can we stop dreaming of data migrations finally?*
- If your queries are spread across apps
  - *Committed to current query language*
  - *Maybe you just want to do a PoC first?*
- Great time for builders to engage early!
  - *If you just love hacking open-source code early 😊*
  - *Much easier to become a contributor now!*

# THANK YOU!



[tinyurl.com/ch-lake-slides](https://tinyurl.com/ch-lake-slides)



## Q&A

[github.com/shiv4289/shiv-tech-talks/](https://github.com/shiv4289/shiv-tech-talks/)  
[linkedin.com/in/shivjijha](https://linkedin.com/in/shivjijha)