

PrismDB vs DuckDB: A Comprehensive Technical Comparison

Whitepaper Version: 1.0 Date: December 2025 Authors: PrismDB Team

Executive Summary

This whitepaper provides an in-depth feature-by-feature comparison between **PrismDB**, a modern analytical database written in Rust, and **DuckDB**, the widely-adopted in-process SQL OLAP database. Both systems share similar design philosophies as embedded analytical databases but differ in implementation language, maturity, and specific feature sets.

Key Findings:

- Both databases target **embedded/in-process OLAP workloads** with zero external dependencies
- PrismDB is written in **Rust** while DuckDB is written in **C++**
- Both use **columnar storage** with **vectorized execution** engines
- DuckDB has a more **mature ecosystem** with 6+ million monthly downloads
- PrismDB offers **comparable SQL features** with a Rust-native implementation
- Both provide **Python-first integration** for data science workflows
- PrismDB's roadmap includes **HTAP capabilities** (document store, vector database, graph database)

Table of Contents

- [1. Introduction](#)
- [2. Architecture Comparison](#)
- [3. Storage Engine](#)
- [4. Query Engine](#)
- [5. Data Types](#)
- [6. SQL Features](#)
- [7. Indexing Mechanisms](#)
- [8. Compression](#)
- [9. Transaction Support](#)
- [10. Extensions & Integrations](#)
- [11. Python Integration](#)
- [12. Performance Characteristics](#)
- [13. Use Cases](#)
- [14. Feature Comparison Matrix](#)
- [15. Conclusion](#)

1. Introduction

1.1 PrismDB Overview

PrismDB is a high-performance analytical database written in Rust, designed for OLAP workloads. Inspired by DuckDB's architecture, it emphasizes:

- Rust-native implementation:** Memory safety without garbage collection
- Embedded deployment:** In-process execution with zero dependencies
- Python integration:** First-class bindings via PyO3
- ACID compliance:** Full transaction support with MVCC
- Educational foundation:** Clean, well-documented codebase

Current Version: 0.1.0 (Active Development)

1.2 DuckDB Overview

DuckDB is an open-source in-process SQL OLAP database management system. Originally developed at CWI Amsterdam, it emphasizes:

- **C++ implementation:** High performance with manual memory management
- **Zero dependencies:** Compiles into a single header and implementation file
- **Broad language support:** APIs for Python, R, Java, Node.js, Go, and more
- **Production maturity:** Version 1.0 released June 2024, 6M+ monthly downloads
- **Active ecosystem:** Extensive extensions and community contributions

Current Version: 1.4.x (Production Ready)

1.3 Shared Design Philosophy

Both databases were designed with similar goals:

Design Principle	PrismDB	DuckDB
Embedded execution	Yes	Yes
No external dependencies	Yes	Yes
OLAP-optimized	Yes	Yes
Columnar storage	Yes	Yes
Vectorized execution	Yes	Yes
Single-file database	Yes	Yes
Python-first	Yes	Yes

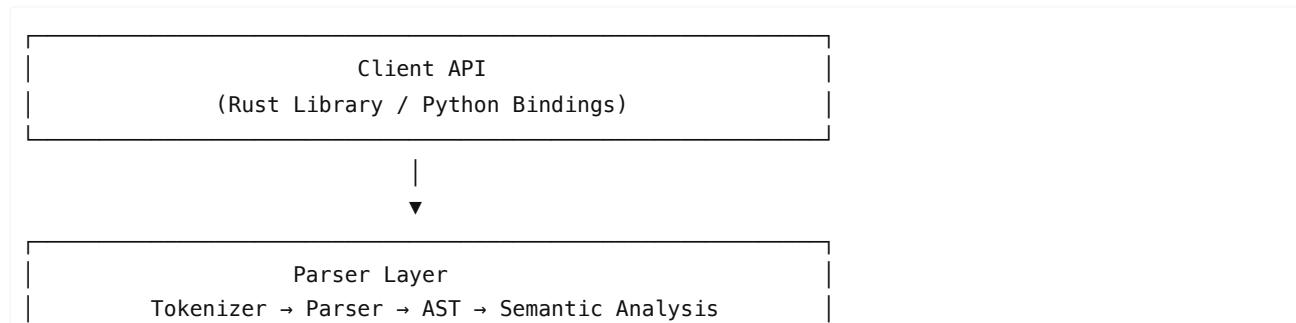
2. Architecture Comparison

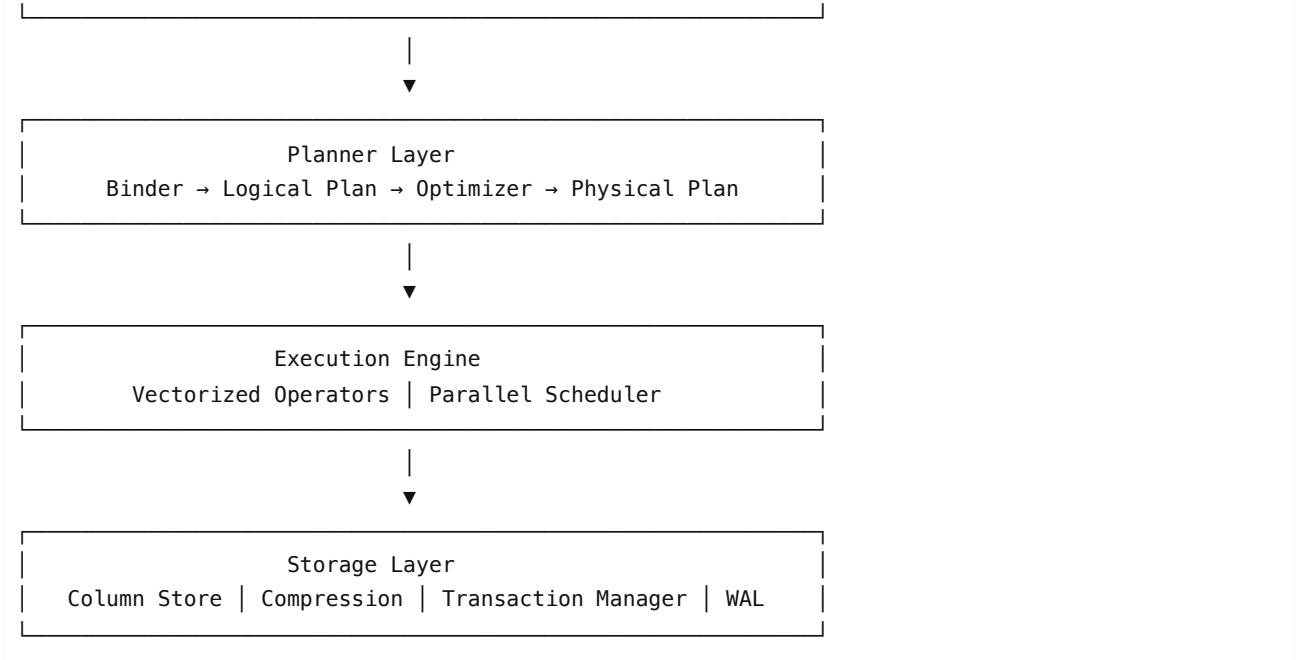
2.1 System Architecture

Aspect	PrismDB	DuckDB
Implementation Language	Rust	C++
Deployment Model	Embedded/in-process	Embedded/in-process
Memory Safety	Compile-time (Rust)	Manual management
Concurrency Model	Multi-threaded (Rayon)	Multi-threaded (custom)
Build Output	Library + Python wheel	Header + implementation file

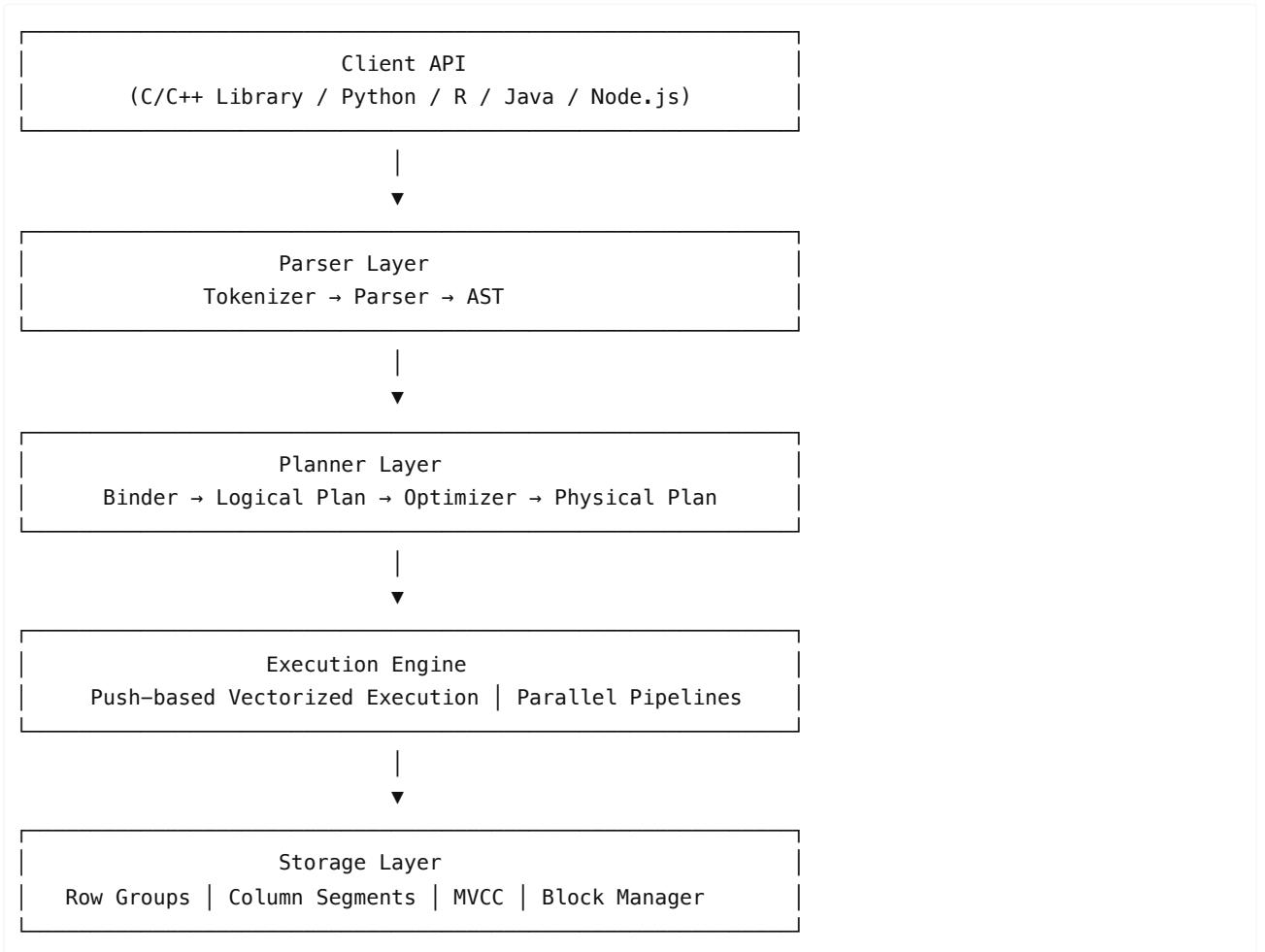
2.2 Component Architecture

PrismDB Architecture:





DuckDB Architecture:



2.3 Academic Foundations

Both systems draw from similar academic research:

Paper	PrismDB	DuckDB
-------	---------	--------

MonetDB/X100 (Vectorized Execution)	Inspired	Directly inspired
Morsel-Driven Parallelism	Yes	Yes (push-based variant)
MVCC for Main-Memory Databases	Yes	Yes
Adaptive Radix Tree	Planned	Yes

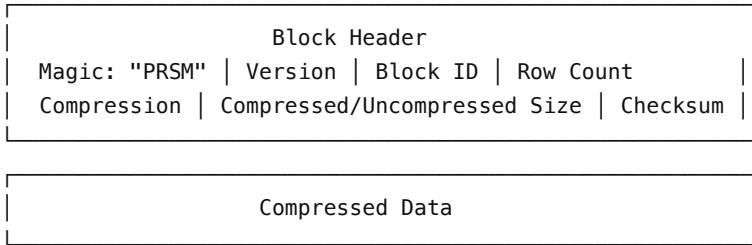
3. Storage Engine

3.1 Storage Format

PrismDB Storage:

- Block-based organization (256KB blocks)
- Separate files per column within blocks
- Validity masks for NULL tracking (64-bit bitmasks)
- Column-level statistics (min, max, null count, distinct count)
- MVCC version chains for row updates

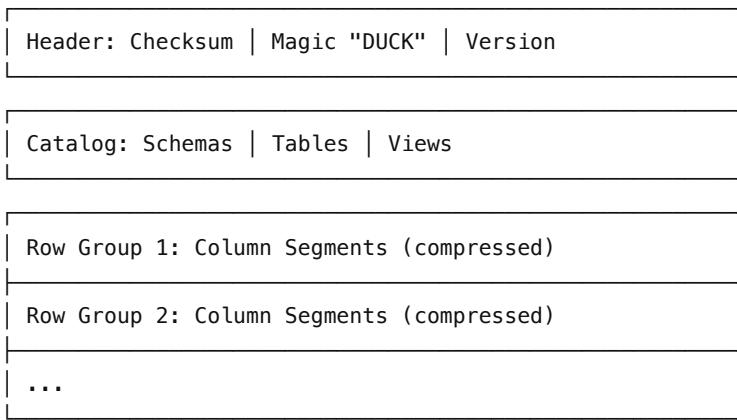
Block Structure:



DuckDB Storage:

- Row group-based organization (120K rows per group)
- Column segments within row groups
- Fixed-size blocks (256KB)
- Sparse metadata for efficient scanning
- Single-file database format

Storage Structure:



3.2 Storage Features Comparison

Feature	PrismDB	DuckDB
---------	---------	--------

Row Group Size	Configurable	120K rows (default)
Block Size	256KB	256KB
Single-file DB	Yes	Yes
In-memory Mode	Yes	Yes
Persistent Mode	Yes	Yes
Partial Loading	Basic	Advanced
Checkpointing	WAL-based	Atomic block writes
Format Versioning	Internal	Public (backward compatible)

3.3 Version Compatibility

DuckDB: Starting from v0.10, DuckDB maintains backward compatibility—newer versions can read files created by older versions. Forward compatibility is provided on a best-effort basis.

PrismDB: Currently in active development; format stability is not yet guaranteed.

4. Query Engine

4.1 Execution Model

Aspect	PrismDB	DuckDB
Execution Style	Pull-based vectorized	Push-based vectorized
Vector Size	2048 tuples	2048 tuples
Parallelism Model	Morsel-driven (Rayon)	Pipeline-based
JIT Compilation	Planned	No (by design)
SIMD Support	Per-function	Automatic

4.2 Vector Types

PrismDB Vector Types:

- Flat vectors (contiguous storage)
- Constant vectors (single value)
- Dictionary vectors (indices + dictionary)
- Selection vectors (filtered indices)

DuckDB Vector Types:

- Flat vectors (contiguous array)
- Constant vectors (single repeated value)
- Dictionary vectors (child + selection vector)
- Sequence vectors (offset + increment)
- Unified Vector Format (generic view)

4.3 Physical Operators

Operator	PrismDB	DuckDB

Table Scan	Yes	Yes
Index Scan	Yes	Yes
Filter	Yes	Yes
Projection	Yes	Yes
Hash Join	Yes	Yes
Sort-Merge Join	Yes	Yes
Nested Loop Join	Yes	Yes
Hash Aggregate	Yes	Yes
Sort	Yes	Yes
Limit/Top-N	Yes	Yes
Window	Yes	Yes
Union/Intersect/Except	Yes	Yes
Pivot/Unpivot	Yes	Yes

4.4 Query Optimization

Optimization	PrismDB	DuckDB
Filter Pushdown	Yes	Yes
Projection Pushdown	Yes	Yes
Constant Folding	Yes	Yes
Join Reordering	Yes	Yes (dynamic programming)
Limit Pushdown	Yes	Yes
Common Subexpression Elimination	Basic	Yes
Predicate Pushdown to Storage	Yes	Yes
Statistics-based Optimization	Column-level	Advanced

5. Data Types

5.1 Primitive Types

Type Category	PrismDB	DuckDB
Boolean	BOOLEAN	BOOLEAN (BOOL)
Signed Integers	TINYINT, SMALLINT, INTEGER, BIGINT, HUGEINT	TINYINT, SMALLINT, INTEGER, BIGINT, HUGEINT
Unsigned Integers	-	UTINYINT, USMALLINT, UINTEGER, UBIGINT, UHUGEINT
FLOATs	FLOAT, DOUBLE	FLOAT, DOUBLE

Decimals	DECIMAL(p,s)	DECIMAL(p,s)
Strings	VARCHAR, CHAR, TEXT	VARCHAR, CHAR, TEXT, STRING
Binary	BLOB	BLOB, BYTEA, VARBINARY
Bit Strings	-	BIT, BITSTRING

5.2 Temporal Types

Type	PrismDB	DuckDB
Date	DATE	DATE
Time	TIME	TIME
Timestamp	TIMESTAMP	TIMESTAMP, DATETIME
Timestamp with TZ	Basic	TIMESTAMPTZ
Interval	INTERVAL	INTERVAL (typed)

5.3 Complex Types

Type	PrismDB	DuckDB
Arrays	LIST(T), ARRAY	ARRAY (fixed), LIST (variable)
Structs	STRUCT(fields)	STRUCT(fields)
Maps	MAP(K, V)	MAP(K, V)
JSON	JSON	JSON
Enums	ENUM(values)	ENUM(values)
UUID	UUID	UUID
Union	UNION(types)	UNION(types)
Big Numbers	-	BIGNUM

5.4 Type System Comparison

DuckDB Advantages:

- Unsigned integer variants
- Typed intervals (INTERVAL YEAR, INTERVAL DAY, etc.)
- BIT/BITSTRING types
- BIGNUM for arbitrary-precision integers
- More mature JSON type implementation

PrismDB Advantages:

- Simpler, more uniform type system
- All types nullable by default
- Consistent naming conventions

6. SQL Features

6.1 DDL (Data Definition Language)

Feature	PrismDB	DuckDB
CREATE TABLE	Yes	Yes
CREATE OR REPLACE TABLE	Yes	Yes
CREATE TABLE AS SELECT	Yes	Yes
ALTER TABLE	Basic	Extensive
DROP TABLE	Yes	Yes
CREATE VIEW	Yes	Yes
MATERIALIZED VIEW	Yes (with refresh)	Yes
CREATE INDEX	Yes	Yes (ART)
TEMPORARY TABLES	Planned	Yes

6.2 DML (Data Manipulation Language)

Feature	PrismDB	DuckDB
SELECT	Yes	Yes
INSERT	Yes	Yes
INSERT BY NAME	Planned	Yes
INSERT OR IGNORE/REPLACE	Planned	Yes
UPDATE	Yes	Yes
DELETE	Yes	Yes
MERGE/UPSERT	Planned	Limited
TRUNCATE	Yes	Yes

6.3 Advanced SQL Features

Feature	PrismDB	DuckDB
CTEs (WITH clause)	Yes	Yes
Recursive CTEs	In progress	Yes
Window Functions	Full support	Full support
PIVOT/UNPIVOT	Yes	Yes
QUALIFY	Yes	Yes
GROUP BY ALL	Planned	Yes
ORDER BY ALL	Planned	Yes
SELECT * EXCLUDE	Planned	Yes
SELECT * REPLACE	Planned	Yes

COLUMNS() expression	Planned	Yes
Lateral Column Aliases	Planned	Yes
FROM-first Syntax	Planned	Yes

6.4 Join Types

Join Type	PrismDB	DuckDB
INNER JOIN	Yes	Yes
LEFT/RIGHT JOIN	Yes	Yes
FULL OUTER JOIN	Yes	Yes
CROSS JOIN	Yes	Yes
SEMI JOIN	Yes	Yes
ANTI JOIN	Yes	Yes
LATERAL JOIN	No	Yes
ASOF JOIN	No	Yes
POSITIONAL JOIN	No	Yes

6.5 Window Functions

Function	PrismDB	DuckDB
ROW_NUMBER	Yes	Yes
RANK	Yes	Yes
DENSE_RANK	Yes	Yes
NTILE	Yes	Yes
PERCENT_RANK	Yes	Yes
CUME_DIST	Yes	Yes
LAG/LEAD	Yes	Yes
FIRST_VALUE/LAST_VALUE	Yes	Yes
NTH_VALUE	Yes	Yes
Frame Types	ROWS, RANGE, GROUPS	ROWS, RANGE, GROUPS

6.6 DuckDB "Friendly SQL" Features

DuckDB offers numerous SQL ergonomic improvements that PrismDB plans to adopt:

Feature	PrismDB	DuckDB
Trailing commas	Planned	Yes
Percentage LIMIT	No	Yes

Prefix aliases (x: 42)	No	Yes
UNION BY NAME	Planned	Yes
Dot operator chaining	No	Yes
Negative array indexing	Planned	Yes
Underscore in numeric literals	Planned	Yes
Case-insensitive identifiers	Yes	Yes

7. Indexing Mechanisms

7.1 Index Types

Index Type	PrismDB	DuckDB
B-Tree	Yes	No
Hash	Yes	No
ART (Adaptive Radix Tree)	Planned	Yes
Min-Max (Zone Maps)	Statistics	Yes
Bloom Filter	Planned	No
GIST	Planned	No
GIN	Planned	No

7.2 Index Implementation

PrismDB:

- Traditional B-tree and hash index support
- Index statistics for cost estimation
- Multi-column index support
- Unique constraints via unique indexes

DuckDB:

- ART (Adaptive Radix Tree) for secondary indexes
- Zone maps (min/max) per row group for predicate filtering
- Automatic index selection by optimizer
- Limited to unique/primary key enforcement

7.3 Predicate Pushdown

Both databases support pushing predicates to the storage layer:

Mechanism	PrismDB	DuckDB
Column statistics filtering	Yes	Yes
Zone map filtering	Via statistics	Yes
Index-based filtering	Yes	Yes (ART)
Parquet predicate pushdown	Yes	Yes

8. Compression

8.1 Compression Algorithms

Algorithm	PrismDB	DuckDB
Constant Encoding	Implicit	Yes
Dictionary Encoding	Yes	Yes
Run-Length Encoding (RLE)	Yes	Yes
Bit Packing	Planned	Yes
Frame of Reference (FOR)	Planned	Yes
Delta Encoding	Planned	Implicit in FOR
FSST (String Compression)	Planned	Yes
Chimp (Floats)	No	Yes
Patas (Floats)	No	Yes
ALP (Floats)	No	Yes
LZ4	Planned	No (lightweight only)
ZSTD	Planned	No (lightweight only)

8.2 Compression Strategy

PrismDB:

- Adaptive compression selection per column segment
- Analyze phase samples data to choose optimal algorithm
- Dictionary encoding for cardinality < 10% unique
- RLE for > 20% consecutive duplicates

DuckDB:

- Lightweight compression only (fast encode/decode)
- Automatic algorithm selection per column segment
- Compression applied at row group level
- No heavy compression (by design, for query speed)

8.3 Compression Characteristics

Aspect	PrismDB	DuckDB
Compression scope	Column segment	Column segment
Analysis phase	Yes	Yes
Heavy compression	Planned (LZ4, ZSTD)	No
Floating-point specialized	No	Yes (Chimp, Patas, ALP)
String specialized	Dictionary	Dictionary + FSST

8.4 Disk Usage Estimates (DuckDB)

- 100 GB uncompressed CSV → ~25 GB DuckDB format
 - 100 GB Parquet → ~120 GB DuckDB format (Parquet already compressed)
-

9. Transaction Support

9.1 ACID Properties

Property	PrismDB	DuckDB
Atomicity	Full	Full
Consistency	Full	Full
Isolation	Multiple levels	Snapshot
Durability	WAL-based	Checkpoint-based

9.2 Isolation Levels

PrismDB:

- Read Uncommitted
- Read Committed
- Repeatable Read (default)
- Serializable

DuckDB:

- Snapshot isolation only
- Single writer, multiple readers
- Optimistic concurrency control

9.3 MVCC Implementation

PrismDB:

- Row-level versioning with Xmin/Xmax transaction IDs
- Version chains for historical reads
- Snapshot isolation via transaction ID comparison
- Rollback support with saved data

DuckDB:

- Bulk-optimized MVCC
- Row group-level versioning
- Optimized for analytical (bulk) operations
- Single writer limitation for simplicity

9.4 Concurrency Model

Aspect	PrismDB	DuckDB
Concurrent Readers	Yes	Yes
Concurrent Writers	Limited (row-level locking)	No (single writer)
Read-Write Concurrency	Yes	Yes
Lock Granularity	Row-level	Database-level writes

10. Extensions & Integrations

10.1 Extension System

PrismDB Extensions:

- CSV reader with auto-detection
- Parquet reader
- JSON reader
- SQLite scanner
- AWS S3 integration (Signature V4)
- HTTP/HTTPS file reading

DuckDB Extensions:

- Core: Parquet, JSON, HTTPFS, AWS
- Data formats: Arrow, Excel, Spatial
- Database connectors: PostgreSQL, MySQL, SQLite
- Cloud: S3, Azure, GCS
- Specialized: Full-text search, ICU, TPC-H/TPC-DS

10.2 Extension Comparison

Extension Category	PrismDB	DuckDB
Parquet	Read	Read/Write
CSV	Read/Write	Read/Write
JSON	Read	Read/Write
Excel	No	Yes
Arrow	Planned	Yes
SQLite	Read	Read
PostgreSQL	Planned	Yes
MySQL	No	Yes
S3/Cloud	Yes	Yes
HTTP(S)	Yes	Yes
Spatial/GIS	Planned	Yes
Full-text Search	No	Yes
Delta Lake	No	Yes
Iceberg	No	Yes

10.3 File Format Support

Format	PrismDB	DuckDB
CSV	Read/Write	Read/Write
Parquet	Read	Read/Write
JSON	Read	Read/Write

Arrow	Planned	Yes
ORC	Planned	No
Avro	Planned	No
Excel	No	Yes
SQLite	Read	Read

11. Python Integration

11.1 Installation & Setup

PrismDB:

```
import prismdb

# In-memory database
db = prismdb.connect()

# File-based database
db = prismdb.connect('mydata.db')
```

DuckDB:

```
import duckdb

# In-memory database
con = duckdb.connect()

# File-based database
con = duckdb.connect('mydata.duckdb')
```

11.2 Query Execution

PrismDB:

```
# Execute and iterate
result = db.execute("SELECT * FROM users")
for row in result:
    print(row)

# Fetch all results
rows = db.execute("SELECT * FROM users").fetchall()

# To dictionary
data = db.to_dict("SELECT * FROM users")
```

DuckDB:

```
# Execute and fetch
result = con.execute("SELECT * FROM users").fetchall()
```

```

# To DataFrame
df = con.execute("SELECT * FROM users").df()

# Query DataFrame directly
import pandas as pd
df = pd.DataFrame({'a': [1, 2, 3]})
result = con.execute("SELECT * FROM df WHERE a > 1").df()

```

11.3 DataFrame Integration

Feature	PrismDB	DuckDB
Pandas integration	Via conversion	Zero-copy query
Polars integration	Planned	Yes
Arrow integration	Planned	Yes
Query DataFrames directly	Planned	Yes
Result to DataFrame	Manual	Native (.df())

11.4 Zero-Copy Operations

DuckDB Advantage: DuckDB can query Pandas DataFrames directly without copying data:

```

import pandas as pd
import duckdb

df = pd.DataFrame({'x': range(1000000)})
# No data copying – queries DataFrame in-place
result = duckdb.query("SELECT * FROM df WHERE x > 500000")

```

PrismDB: Currently requires data import; zero-copy planned for future versions.

12. Performance Characteristics

12.1 Execution Optimizations

Optimization	PrismDB	DuckDB
Vectorized execution	Yes (2048 tuples)	Yes (2048 tuples)
SIMD operations	Per-function	Automatic
Parallel execution	Morsel-driven	Pipeline-based
Work stealing	Yes (Rayon)	Yes (custom)
Late materialization	Yes	Yes
Streaming results	Yes	Yes

12.2 Memory Management

Aspect	PrismDB	DuckDB
Buffer pool	Yes (LRU)	Yes

Memory limits	Configurable	Configurable
Spill to disk	Planned	Yes
Out-of-core processing	Limited	Yes
Memory tracking	Per-operation	Hierarchical

12.3 I/O Optimizations

Optimization	PrismDB	DuckDB
Predicate pushdown	Yes	Yes
Column pruning	Yes	Yes
Parallel I/O	Basic	Advanced
Buffer management	Yes	Yes
Direct file queries	Yes	Yes

12.4 Benchmark Considerations

Direct performance comparisons should consider:

1. **Maturity:** DuckDB has years of optimization; PrismDB is newer
 2. **Use case:** Both optimized for OLAP, but may differ on specific queries
 3. **Data size:** Performance characteristics may vary with scale
 4. **Hardware:** Both utilize modern CPU features
-

13. Use Cases

13.1 Where Both Excel

Use Case	Suitability
Local data analysis	Excellent
Python/data science workflows	Excellent
ETL and data transformation	Excellent
Interactive exploration	Excellent
Embedded analytics	Excellent
Single-file databases	Excellent

13.2 Where DuckDB Has Advantages

1. Production Maturity

- Version 1.0+ with backward compatibility
- 6+ million monthly downloads
- Extensive real-world testing

2. Ecosystem

- More extensions available

- Broader language support
- Community contributions

3. Advanced Features

- Zero-copy DataFrame queries
- ASOF/LATERAL joins
- Delta Lake/Iceberg support
- Friendly SQL syntax extensions

4. Documentation & Support

- Comprehensive documentation
- Active community
- Commercial support (MotherDuck)

13.3 Where PrismDB Has Advantages

1. Rust Implementation

- Memory safety guarantees
- Modern language ecosystem
- Easier contributions for Rust developers

2. Transaction Support

- Multiple isolation levels
- Row-level locking
- More granular concurrency control

3. Index Support

- Traditional B-tree indexes
- Hash indexes
- More familiar to RDBMS users

4. HTAP Roadmap

- Planned document store (MongoDB-like)
- Planned vector database
- Planned graph database

13.4 Decision Matrix

Requirement	PrismDB	DuckDB
Production deployment	Consider	Recommended
Rust ecosystem	Recommended	Consider
Python data science	Good	Excellent
Multiple isolation levels	Recommended	Limited
Zero-copy DataFrame	Limited	Excellent
Traditional indexes	Recommended	Limited
Extension ecosystem	Growing	Excellent
Long-term stability	Developing	Stable

14. Feature Comparison Matrix

14.1 Core Features

Feature	PrismDB	DuckDB
Columnar Storage	✓	✓
Vectorized Execution	✓	✓
Parallel Execution	✓	✓
SIMD Optimization	⚠ Partial	✓
Zero Dependencies	✓	✓
Single-file Database	✓	✓
In-memory Mode	✓	✓
Persistent Storage	✓	✓
ACID Transactions	✓	✓
Multiple Isolation Levels	✓	✗
Backward Compatibility	✗	✓

14.2 SQL Features

Feature	PrismDB	DuckDB
Standard SQL	✓	✓
CTEs	✓	✓
Recursive CTEs	⚠ Partial	✓
Window Functions	✓	✓
PIVOT/UNPIVOT	✓	✓
QUALIFY	✓	✓
GROUP BY ALL	➔ SOON	✓
LATERAL JOIN	✗	✓
ASOF JOIN	✗	✓
Friendly SQL	➔ SOON	✓

14.3 Storage & Compression

Feature	PrismDB	DuckDB
Dictionary Compression	✓	✓
RLE Compression	✓	✓
Bit Packing	➔ SOON	✓

FSST (Strings)	➡ SOON	✓
Float Compression	✗	✓
Zone Maps	⚠ Stats	✓
Row Groups	✓	✓

14.4 Integrations

Feature	PrismDB	DuckDB
Python	✓	✓
R	✗	✓
Java	➡ SOON	✓
Node.js	➡ SOON	✓
Go	➡ SOON	✓
Parquet	⚠ Read	✓
CSV	✓	✓
JSON	✓	✓
Arrow	➡ SOON	✓
S3/Cloud	✓	✓
Zero-copy DataFrame	➡ SOON	✓

14.5 Legend

- ✓ Fully supported
- ⚠ Partially supported
- ➡ SOON Planned/In development
- ✗ Not supported

15. Conclusion

15.1 Summary

PrismDB and DuckDB are both excellent embedded analytical databases with similar core architectures:

Dimension	PrismDB	DuckDB
Implementation	Rust	C++
Maturity	Early (v0.1)	Production (v1.4)
Core Architecture	Columnar + Vectorized	Columnar + Vectorized
Deployment	Embedded	Embedded
Primary Language	Python + Rust	Python (+ many others)
Transaction Model	Full MVCC	Optimized MVCC

Ecosystem	Growing	Mature
-----------	---------	--------

15.2 When to Choose PrismDB

- **Rust ecosystem integration** is a priority
- **Multiple transaction isolation levels** are required
- **Traditional B-tree/hash indexes** are needed
- **Contributing to a Rust codebase** is preferred
- **HTAP capabilities** (document/vector/graph) are on the roadmap

15.3 When to Choose DuckDB

- **Production stability** is critical
- **Broad language support** (R, Java, Node.js) is needed
- **Zero-copy DataFrame operations** are important
- **Extensive extension ecosystem** is required
- **Backward compatibility guarantees** are needed
- **Delta Lake/Iceberg** integration is required

15.4 Complementary Usage

Both databases can be used complementarily:

1. **Development:** Use DuckDB for rapid prototyping with its mature tooling
2. **Rust Projects:** Integrate PrismDB for native Rust applications
3. **Data Exchange:** Both support Parquet for data interchange
4. **Learning:** PrismDB's clean Rust codebase is excellent for education

15.5 Future Outlook

PrismDB Roadmap:

- HTAP capabilities (document store, vector DB, graph DB)
- Enhanced compression (LZ4, ZSTD)
- Expanded language bindings
- Distributed query execution (long-term)

DuckDB Trajectory:

- Continued stability and performance
- MotherDuck cloud integration
- Expanded extension ecosystem
- Community-driven enhancements

References

1. DuckDB Official Documentation - <https://duckdb.org/docs/>
2. DuckDB Why DuckDB - https://duckdb.org/why_duckdb
3. DuckDB Friendly SQL - https://duckdb.org/docs/stable/sql/dialect/friendly_sql
4. DuckDB Lightweight Compression - <https://duckdb.org/2022/10/28/lightweight-compression>
5. DuckDB Storage Format - <https://duckdb.org/docs/stable/internals/storage>
6. DuckDB Vector Execution - <https://duckdb.org/docs/stable/internals/vector>
7. CMU Database Systems Lecture on DuckDB - <https://15721.courses.cs.cmu.edu/spring2024/notes/20-duckdb.pdf>
8. PrismDB Architecture Documentation
9. "MonetDB/X100: Hyper-Pipelining Query Execution" (Boncz et al.)
10. "Morsel-Driven Parallelism" (Leis et al.)

This whitepaper was generated based on analysis of PrismDB source code (version 0.1.0) and publicly available DuckDB documentation as of December 2025.